Identifying reusable functions in code using specification driven techniques

De Lucia, Andrea

University of Durham

Department of Computer Science

Identifying Reusable Functions in Code Using Specification Driven Techniques

Andrea De Lucia

M.Sc.

1995

Abstract

The work described in this thesis addresses the field of *software reuse.* Software reuse is widely considered as a way to increase the productivity and improve the quality and reliability of new software systems. Identifying, extracting and reengineering software components which implement abstractions within existing systems is a promising cost-effective way to create reusable assets. Such a process is referred to as *reuse reengineering.* A reference paradigm has been defined within the RE^2 project which decomposes a reuse reengineering process in five sequential phases. In particular, the first phase of the reference paradigm, called *Candidature phase*, is concerned with the analysis of source code for the identification of software components implementing abstractions and which are therefore candidate to be reused. Different *candidature criteria* exist for the identification of reuse-candidate software components. They can be classified in structural methods (based on structural properties of the software) and specification driven methods (that search for software components implementing a given specification).

In this thesis a new specification driven candidature criterion for the identification and the extraction of code fragments implementing functional abstractions is presented. The method is driven by a formal specification of the function to be isolated (given in terms of a precondition and a postcondition) and is based on the theoretical frameworks of program slicing and symbolic execution. Symbolic execution and theorem proving techniques are used to map the specification of the functional abstractions onto a slicing criterion. Once the slicing criterion has been identified the slice is isolated using algorithms based on dependence graphs. The method has been specialised for programs written in the C language. Both symbolic execution and program slicing are performed by exploiting the Combined C Graph (CCG), a fine-grained dependence based program representation that can be used for several software maintenance tasks.

Acknowledgements

I wish to thank Mr. Malcolm Munro from the University of Durham, for his supervision and guidance throughout the development of the ideas of this research and for his intervention in the correction of this thesis.

Special thanks are due to Prof. Aniello Cimitile from the University of Naples, who helped me to come to Durham. I am also grateful to him for his advices and comments on several drafts of this work, and to all the members of the software engineering research group of the Department of "Informatica e Sistemistica" at the University of Naples for the discussions that gave origin to this research.

I would like to thank Prof. Keith Bennett for all the facilities provided and all the staff of the Department of Computer Science at the University of Durham for all the facilities provided. I also would like to thank the members of the Department of Computer Science and in particular of the Centre for Software Maintenance at the University of Durham for the useful discussions about this work.

Finally, but most important, I thank my fiancée Maddalena who unselfishly encouraged me to come to Durham and supported me throughout this work that has kept me far from her during the past year. This thesis is dedicated to her.

Statement of Copyright

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The introduction of the third generation computers coincides with the *software crisis era.* The power of these new machines and, at the same time their cheapness has produced an increasing demand for software systems of higher and higher complexity. While the field of programming had made tremendous progress (through the systematic study of algorithms and data structures and the invention of *structured programming* [67]) there were still major difficulties in building large software systems. Then in the late 1960s the term *software engineering* was invented [31] to assert that the development of software systems is an engineering discipline that cannot be simply based on a personal activity, such as programming. Parnas and Weiss [141] have defined software engineering as "multi-person construction of multi-version software". Indeed, software engineering deals with systems that are built by teams rather than individual programmers [89]. A system is built using engineering principles: while a programmer writes a complete program, the software engineer writes a software component that will be combined with the components written by other software engineers. The component one writes may be modified by others and may be used to build different versions of the system long after one has left the project.

However, building large software systems is different from building smaller systems. There were fundamental difficulties in scaling up the techniques of small program development to large software development. Indeed, the development of a large software system involves both technical and non-technical aspects [161]. During the first attempt in the production of large software systems, a number of difficulties were discovered: the problems being solved were not well understood by all the people involved in the project, people had to spend a lot of time communicating with each other rather than writing the code, people leaving the project affected the work of other people, replacing an individual required an extensive amount of training about the (poorly defined) project requirements and system design. Many solutions were proposed and tried, but the common consensus was to view the final software system as a complex product and the building of it as an engineering job. The engineering approach involves management, organisation, tools, theories, methodologies, and techniques to be applied for the design and the construction of computer programs and the associated documentation required to develop, operate and maintain them [21].

From the inception of an idea for a software system, until it is implemented and delivered to a customer, and even after that, the system undergoes gradual development and evolution. The software is said to have a *life cycle* composed of several phases. In the traditional *waterfall* life cycle model, first introduced by Royce [152], each phase has well defined starting and ending points, with clearly identifiable deliverables to the next phase. The waterfall life cycle model, currently adopted for software development, comprises five phases [183]:

Requirements analysis and specification. The purpose of this phase is to identify and document the exact requirements for the system, in end-user terms. It follows a *feasibility study* and can be performed by the customer, the developer, a marketing organisation or any combination of the three. Interaction between user and developer is often required.

System and software design. The purpose of this phase is to design a particular software system that will meet the stated requirements. This phase is sometimes split into two subphases: *architectural or high-level design* (which deals with the overall module structure and organisation) and *detailed design* (which refines the high level design by considering each module in detail).

Implementation and unit testing. This phase produces the actual code that will be delivered to the customer as the running system. The system design is coded in a set of modules written in some programming language. The individual modules are also tested (unit testing) in order to verify that each of them meets its specification.

Validation, integration and system testing. All the modules that have been developed and tested individually are integrated in this phase and tested as a whole system to ensure that the software requirements have been met.

Delivery and maintenance. Once the system passes all the tests, it is delivered to the customer and enters the maintenance phase. During this phase modifications can be made to the system to correct faults (that were not discovered in the earlier phases) improve performance, or other attributes, or adapt to a changed environment [184].

Although software engineering has made progress since the 1960s, the field is still far from achieving the status of a classic engineering discipline [89]. Many areas remain in the field that are still being taught and practised on the basis of informal techniques. Moreover, unlike other classical engineering disciplines, the software engineer lack of the mathematical maturity to specify the properties of the product separately from those of the design. Problems are in particular related to the maintenance phase, concerned with the evolution of existing systems to meet ever changing user needs [14, 15]. Changes may be driven by different reasons. Lientz and Swanson [126] subdivided maintenance activities into four categories:

Perfective maintenance. Changes are required to respond to user requests (e.g., request of enhancing the functionalities of the system).

Adaptive maintenance. Changes are needed as a consequence of changes in the data environment (e.g., system input and output formats), or in the processing environment (e.g., hardware, operating systems, etc.)

Corrective maintenance. Changes are made to correct faults identified in the system, which cause incorrect output or abnormal termination of the system.

Preventive maintenance. Changes are made to the software system to improve its quality and reliability in order to anticipate future problems.

Software systems that keep their usefulness in the real world are in continuous evolution [125]. It has been estimated that software maintenance can consume 60-70% of the costs during the life cycle of the system [126]. Moreover, 50-65% of the maintenance phase is taken up with perfective maintenance [126]. Software maintenance is related with several technical and managerial problems [15]. As the system to be changed is already used, the client expects that changes are accomplished quickly, cost-effectively and without degrading the reliability and the maintainability of the system. Unfortunately, these expectations are only rarely met. Usually, user changes are described in terms of behaviour of the software system to be mapped into changes to the source code. Moreover, when a change is made to the code, consequential *ripple effect* may require substantial changes in the documentation, design, test suites, etc.

Although the research in software maintenance has been in progress for the last few years, transforming the field from a practitioner activity to an academic discipline, many of the solutions working for laboratory scale pilots do not scale up to industrial sized software. Many system under maintenance are very large and were developed with obsolete techniques by pioneer software engineers. Software evolution increases the complexity of a software system [125]. Repeated changes often results in a degraded structure and increased entropy of the system, which becomes *legacy*. The code and the documentation are not on line anymore.

Any further change can become a hard task, because of the difficulty of understanding the code and the impact of the change on the rest of the system. It has been estimated that 50-90% of maintenance time is devoted to program comprehension [163], even when there is documentation present [150]. Remedial preventive maintenance actions are required to cope with legacy systems [16].

A way to improve the understandability and maintenability of a legacy system is *modularisation.* Modularising an existing system consists of replacing a single large program or module with a functionally equivalent collection of smaller modules [44]. Modularisation is also useful for downsizing large applications from mainframes to distributed client/server platforms [158]. Indeed, a changing hardware platform is becoming a question of vital importance for the economy and the competitiveness of many companies. Therefore, software systems developed for the old platform have to be available on the new one. In many cases reengineering the existing systems and adapting them to the new platform is cost-effective and can be preferable to new development [159]. Another reason for modularising a system is the possibility to reuse its single modules in the development of new software systems [7].

Software reuse has long been recognised as a way to improve both the productivity and the quality of new software projects [17, 83, 84, 127, 168]. The reuse of software components that have been already tested may reduce the costs of software development and increase software productivity. Moreover, reuse-oriented software development can reduce the maintenance costs [11], because maintenance operations on a modularised system can be better localised.

The concept of "software reuse" is not new, but as old as the name "software engineering" [133]. Software components should be massed produced and stored into repositories in order to be used during to compose more complex components and systems during software development. This view is in line with other engineering disciplines, such as electronic and electrical engineering, where the production of hardware systems is accomplished in this way, often exceeding the demand. Nevertheless, at the state of the art, software productivity is still far to close the gap between the demands placed on software industry and what the state of the practice can deliver [22]. Frakes and Isoda [83] state that the goal of software-reuse research is to discover systematic procedures for engineering new systems from existing assets. Reuse approaches raise a number of issues that may be divided into issues related to *developing reusable assets* and issues related to *developing with reusable assets* [134]. In particular, the most immediate problem in reuse is the building up of a repository of reusable software components [7]. The lack of reusable assets prevents reuse from being widely used in industrial environment [42].

A typical solution for the problem of obtaining reusable assets is to develop them *ex-novo* for reuse. Unfortunately, this solution requires a huge initial investment of money, people and time that gives reuse a long lead time before it starts to pay off in a significant way. One of the most promising ways to make the population of a repository of reusable assets cost effective and to obtain useful results in the short time is by extracting and reengineering them from existing software [7, 32, 42, 76, 91, 145]. Existing systems record in various forms (requirements or design documents, code, test cases, user manuals) a large amount of knowledge and expertise. Therefore, they can become a main source of reusable components.

In this thesis the term reuse reengineering will refer to the process of obtaining reusable assets from existing system [42]. Reuse reengineering processes involve three types of activities:

1. accessing the existing systems to identify reuse candidate components;

2. modifying and packaging them independently;

3. understanding their meaning and producing the related specifications.

The correct planning and execution of these activities greatly depends on the availability of reverse engineering and reengineering techniques and tools. Reverse engineering has been defined [50] as the process of analysing a subject system to

- identify the system components and their relationships;

- create representations of the system in another form at higher levels of abstraction.

On the other side, reengineering is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation in the new form [50]. Reverse engineering techniques are mainly required at points 1 and 3 of a reuse reengineering process, while point 1 is mainly related to reengineering software components according to a predefined template.

1.1 Criteria for Success

This thesis deals with the identification and extraction from existing systems of *reuse-candidate* [42] software components. In particular, the use of code scavenging techniques [44] to search existing software systems for source code components implementing software abstractions is investigated. The criteria for success, to be judged in the final chapter, are as follows:

- review of existing program representations;

- description and evaluation of existing methods for the identification of components driven by structural properties of the software;
- description and evaluation of existing methods for the identification of components driven by the specification of the abstraction sought;
- development of a new specification driven method for the identification of code fragments implementing functional abstractions;
- formalisation of the new method;
- prototype implementation of the new method to show that it is automatable;
- evaluation of the new method by the use of a case study.

1.2 Outline of the Thesis

The remainder of the thesis is organised as follows.

Chapter 2 describes a reference paradigm for setting up reuse reengineering processes. The paradigm has been defined within RE^2 [42], a research project jointly carried out by DIS (Department of 'Informatica e Sistemistica') at the University of Naples and CSM (Centre for Software Maintenance) at the University of Durham and funded by CNR (Italian National Research Council). The key role of the paradigm is to define a framework where relevant methodologies and tools can be used, allowing partial solutions to different problems to be linked together, and experiments can be repeated. The chapter focus on the first phase of the RE^2 reference paradigm, called Candidature phase. In this phase, existing software systems are searched for sets of components candidate to implement reusable modules. A set of different code scavenging methods proposed in the literature and looking for code components implementing abstractions are presented. These methods are classified in structural methods and specification driven methods, depending on the way type of search: structural methods search for code components according to structural properties of the software, while specification driven methods search for code components implementing the specification of a given abstraction.

Chapter 3 presents the new specification driven method looking for code fragments implementing the specification of functional abstractions. The method is based on program slicing as a program decomposition technique for isolating code fragments, and uses symbolic execution [60, 64, 111] and theorem proving [6, 29, 143] techniques in order to map the specification of the functional abstraction onto the slicing criterion to be used for isolating the slice implementing the required function. A different definition of slicing criterion is given in order to consider a program slice as a procedure. The specification driven program slicing process is then specialised to programs written in the C language [110]. First some problems involving the symbolic execution of C programs are outlined and then we show how both symbolic execution and program slicing can be performed by exploiting an intermediate representation for C programs called Combined C Graph [112, 113, 114].

Chapter 4 describes a prototype implementation of a tool for isolating reusable functions written in C language [110] using the specification driven program slicing technique. The language chosen for implementing the prototype tool is Prolog [164]. The architecture of the prototype system is described together with the single module components.

Chapter 5 shows the validity of the proposed method through its application to a small demonstrative examples and a case study. The system chosen to validate the method consists of over 5000 LOC (not including comments) of C code. Four large functions implementing more than one functionality have been identified and reengineered by decomposing them into smaller functions each of which implements one functionality. The case study demonstrated the applicability of the method even to programming languages, like C, that provide primitives for the implementation of functional abstractions. The evaluation of the method is also discussed.

Some conclusions and further directions are finally given in chapter 6.

Chapter 2

Literature Review

The reuse of software components can considerably reduce the development costs and improve the quality and the reliability of new software systems [127]. The most immediate problem in reuse is the building up of a repository of reusable software components [7]. Although reusable software components can be designed and implemented during the development of new software projects, existing software is widely considered to be the main source for the extraction of reusable assets [7, 32, 42, 76, 91, 145].

A reuse reengineering process consists of the set of activities for identifying software components implementing abstractions, reengineering them according to a predefined template, associating them with their interface and functional specification and populating a repository with the reusable assets so obtained. In particular, the RE^2 project [42] outlines the role that reverse engineering and reengineering techniques have in a reuse reengineering process involving existing source code.

In this chapter the RE^2 reference paradigm for setting up reuse reengineering processes is described. In particular we focus on the first phase of the paradigm, called the Candidature phase, which is related to the identification of software components implementing abstractions in code. Several program representations proposed in the literature for software maintenance and in particular useful for reverse engineering and reengineering are described. A survey of both structural candidature criteria and specification driven candidature criteria is also presented.

2.1 The RE^2 Reference Paradigm

A reuse reengineering process is a complex process which may entail using methodos and tools from different fields as well as defining new methods and new tools. Each of these methods and tools often only provides a partial solution for a particular problem. Canfora

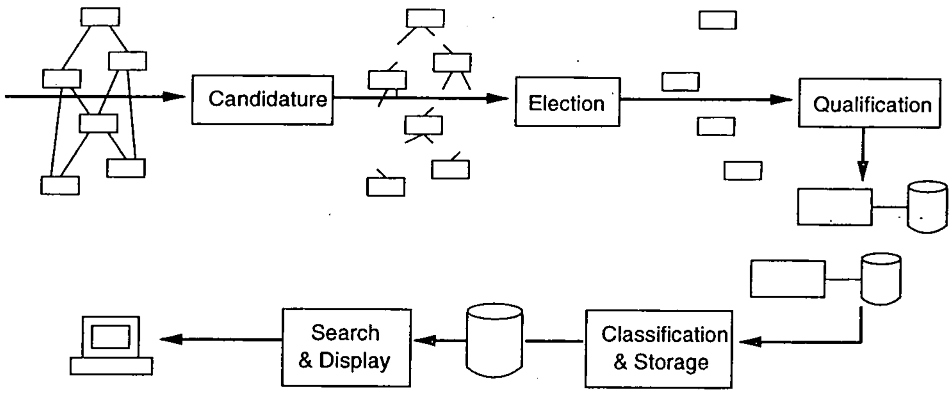


Figure 2.1: The RE^2 Reference Paradigm

et al. [42] show the need for a reference paradigm to set up reuse reengineering processes. The key role of the paradigm is to define a framework where relevant methods and tools can be used allowing partial solutions to be linked together. A reference paradigm is also necessary to allow the repetition of the experiments. Figure 2.1 shows the reference paradigm of the RE^2 research project jointly carried out by DIS (Department of 'Informatica e Sistemistica') at the University of Naples and CSM (Centre for Software Maintenance) at the University of Durham. The RE^2 paradigm decomposes a reuse reengineering process in five sequential phases called *Candidature*, *Election*, *Qualification*, *Classification and Storage* and *Search and Display*. In particular, the first three phases are related to the identification, extraction and reengineering of software components for the production of reusable modules, while the latter two phases populate the repository and set up the environment for the retrieval and the reuse of modules during the development of new systems. The RE^2 project is currently involved with the first three phases of its reference paradigm.

The *Candidature* phase produces sets of software components by using source code analysis techniques. According to the taxonomy presented by Chikofsky and Cross [50], reverse engineering techniques are required to identify software components and the relations existing among them. Each set of software components is candidate to make up a reusable module when suitably de-coupled, re-engineered and possibly generalised. The Candidature phase is organised in three steps:

1. Defining a *candidature criterion* to produce a first approximation of the set of the reuse-candidate modules. This also entails the definition of the model of the system (typically a program representation) to apply the criterion and the information needed to make up an instance of the model.

2. Defining and setting up a *reverse engineering* process to extract a set of software components from code and make up an instance of the model defined in the previous

step.

3. *Applying the candidature criterion* to the particular model instance to produce the set of reuse-candidate modules.

The *Election* phase transforms reuse-candidate modules in reusable modules by de-coupling each set of software components from the external environment and clustering them into a module according to a predefined template. This phase consists of three steps:

1. Defining a *module template* (according to the primitives provided by the particular programming language considered and the concepts of abstraction and information hiding) for reengineering the reuse-candidate modules.

2. Defining and setting up a reengineering process for *de-coupling* the components from the external environment.

3. Defining and setting up a reengineering process for *clustering* the components in the template.

Not all the meaningful candidate sets will be included in the reusable modules. Some of them will be discarded during the election phase because of the complexity and the costs of the reengineering operations needed to de-couple and cluster them [72]. Moreover, a *Concept Assignment* [18] process[1] can be required before the Election phase in order to select the subset of modules that can be associated with human-oriented concepts. Only such reuse-candidate modules will be de-coupled and reengineered. This also allows the validation of a candidature criterion [53].

Reverse engineering activities are also required in the *Qualification* phase which produces the interface and functional specifications of the reusable modules obtained in the election phase. The Qualification phase is organised in three steps:

1. Defining or choosing a *specification formalism* expressing the semantics of a reusable module and how it should be used.

2. Defining and setting up of the *functional reverse engineering* process to produce the functional and the interface specifications from the source code module and to express them according to the defined formalism.

3. *Testing and fixing* the specifications produced to ensure their correctness and consistency with the code.

[1] In agreement with Biggerstaff *et al.* [18] we call the problem of associating software components and related relationships with human oriented concepts the concept assignment problem.

The *Classification and Storage* phase groups together the activities that classify the reusable modules and related specifications according to a reference taxonomy. The aim is to define a repository system and populate it with the reusable modules produced.

The *Search and Display* phase groups together the activities that set up a front end user interface to interact with the repository system. The aim is to make finding the modules the user needs as simple as possible, for example by giving them visual supports to navigate trough the repository system.

2.1.1 Candidature Criteria

The definition of a candidature criterion and the model of the system to apply the criterion are a major problem in a reuse reengineering process. In the RE^2 project [42] the software engineering principles of abstraction [67] and information hiding [140] are assumed as a guideline to define the reuse reengineering process. As a consequence, a candidature criterion should automatically produce a first approximation to the sets of components to extract from a software system, each set being a candidate to constitute a reusable module that implements one abstraction.

There are three search primitives from which candidature criteria can be defined [44]: *Isolation*, *Aggregation*, and *Generalisation.*

Isolation consists of breaking a large monolithic system component that implements more than one abstraction into a set of components each of which implements one abstraction.

Aggregation consists of linking together several low-level system components (i.e., routines, functions, paragraphs, data types, variables, chunks of code) whenever these contribute to implement a higher level abstraction.

Generalisation consists of assigning a module with a higher generality, thus extending the class of problems it can solve. Generalising a module involves storing it in a "generic" form from which several specific modules can be instantiated.

Based on one or more search primitives, many candidature criteria (in the following they are also simply called methods) have been proposed that exploit a large set of technologies and tools. Following the taxonomy proposed by Canfora *et al.* [44], these methods can be grouped into three families: METhods driven by a metric MODel (METMOD), METhods driven by the TYPe of the abstraction to be searched (METTYP), and METhods driven by a full or partial SPECification of the abstraction to be recovered (METSPEC).

METMOD candidature criteria entail the selection of a set of metrics [80] and the definition, for each metric, of a value range that may be considered characteristic of code implementing a reusable abstraction. Typical METMOD methods are those which search pieces of code that exhibit high cohesion [182] and low coupling [182] with the rest of the system [76]. In these cases, metrics capturing data-binding [70, 105], control-binding [76] and similarity measures [156] are the main components of the metric model. Generally, METMOD methods are founded on simple metric models, which consists of very popular metrics such as the McCabe cyclomatic number [132], the Halstead volume [93], the number of lines of code and statements. In these cases the value range of each metric is defined heuristically, for example by analysing the values that metric assumes when applied to software components with high frequency of reuse [32, 76].

METTYP candidature criteria are specialised to search for only one type of abstraction. Examples of abstractions that can be recovered include external and internal functionalities [45, 65, 124, 136, 177], I/O and error handlers [158], front-end user interfaces [131], data base transactions [119, 157], data structures [36, 76, 129], abstract data types [37, 43, 128, 129], domain applications objects and classes [39, 47, 85, 181]. Based on a descriptions of the type of abstractions to be identified, candidature criteria partition the system into a set of components, each of which is candidate to implement an abstraction. Candidature criteria are generally defined in terms of *summary relations* [35] obtained by combining relations directly produced through static analysis of code (for example, call relations, data definitions, data uses). Typical METTYP methods are those which re-arrange the components of a system into a hierarchy which may be mapped onto a possible functional decomposition of the system [46, 55]. METTYP methods also identify clusters of data items, data types, and routines, each cluster being a candidate to implement an object or an abstract data type [167].

METSPEC candidature criteria presuppose the existence of a full or partial specification of an abstraction. Existing systems are searched for code fragments (also delocalised) which satisfy the specification. Typical METSPEC methods are those that, starting from a function partially specified by means of a characteristic subset of output (and possibly input) data, analyse a system for identifying the minimum number of components that contribute to the values of the output data [124]. The specification can be a formal one [41, 59] (in this case tools such as symbolic executors [64] and theorem provers [6, 29, 143] can be required to map the specification onto the code), or can be given in term of a set of test cases capturing the behaviour of the abstraction [92, 177, 178], or can be encapsulated in a knowledge base [3, 115, 120, 142, 146, 180] in term of a library of programming *plans* [160] or *clichés* [148].

The first two families of methods are also called mass methods, because they are applied to one or more systems and produce a large set of candidate modules. In the following we will refer to METMOD and METTYP methods as *structural candidature criteria* [59], because they only analyse structural characteristics of a system (based on software metrics or programming languages primitives used for implementing abstractions). A concept assignment process is required to be applied to the candidate modules produced by a structural candidature criterion, in order to associate them with human oriented meanings. Modules that cannot be associated with any human oriented meaning will be discarded. METSPEC methods are also called spot methods, because they are applied to only one system, or system component, and produce only one module. This is then a candidate to implement the specified abstraction and no concept assignment process has to be applied before the Election phase.

2.2 Existing Program Representations

Program representations play a key role in a reuse reengineering process and in particular during the candidature phase. It is very common to use static analysis techniques for translating the source code into an intermediate representation, that can be used as the model for the application of the candidature criterion. In this section we describe some of the program representations mainly used in software maintenance and software engineering and in particular in reverse engineering and reengineering.

2.2.1 Control Flow Graph and Call Graph

The simplest program representations used in software maintenance are the *control flow graph* [5, 100] and the *call graph*. Both the representation are based on the concept of *flow graph*[2]

Definition 2.1 A flow graph $G = (s, N, E)$ is a directed graph where N is the set of nodes, $s \in N$ and E is the set of edges. For each node $n \in N$ there exists a path from s to n.

[2] In [100], a control flow graph is simply called a flow graph.

Control Flow Graph

In a control flow graph (s, N, E), nodes in N represent single-entry/single-exit regions of executable code (called *basic blocks*) and edges in E represent possible execution flow between code regions. The node s represents the program's entry. Although for compiler optimisation [5] a basic block consists of a maximal single-entry/single-exit sequence of statements, usually in software maintenance each node of a control flow graph correspond to an individual statement. A control flow graph is suitable to perform intraprocedural data flow analysis [5, 100] and program slicing [176].

Call Graph

In a call graph (s, N, E), nodes in N represent procedures[3], edges in E represent call relationships between procedures and edges labels represent actual parameters. The node s represents the *main* procedure. Since one procedure may call another at many points, a call graph may be a multigraph with more than one edge connecting two nodes. A program's call graph can be constructed efficiently [154] and used for different applications, like *flow-insensitive* interprocedural data flow analysis, e.g. [63, 166], and software salvaging [53, 55].

Interprocedural Control Flow Graph

The features of these two representations have been combined in the *interprocedural control flow graph* [123]. An interprocedural control flow graph is the union of the control flow graphs of the individual procedures in the program; each control flow graph has unique *entry* and *exit* nodes and *call sites* are split into *call* and *return* nodes. Each call node is connected to the entry node of the procedure it invokes, while each exit node is connected to return nodes of all call sites that invoke the procedure. The interprocedural control flow graph has been used to solve the *reaching definitions* problem (i.e., the problem of determining the set of variable definitions that reach each program point) in presence of pointers [139]. A similar representation has been used by Myers [135] for flow-sensitive interprocedural data flow analysis.

2.2.2 Program Representations for Data Flow Analysis

The control flow graph and the call graph can be enriched with information which allow *flow-sensitive* data flow analysis. To this aim a variety of representations have been proposed in the literature.

[3] We will refer to the primitives programming languages provide to implement functional abstractions, e.g. *program*, *procedure*, *function*, *subroutine*, with the generic name "procedure".

The Program Summary Graph

A variant of the call graph which provides information for flow-sensitive data flow analysis is the *program summary graph* [33]. The program summary graph represents programs written in a procedural language with call by reference parameters. For each procedure, there are *entry* and *exit* nodes for each formal parameter, while at each call site there are *call* and *return* nodes for each actual parameter. Binding edges relate call nodes with the corresponding entry nodes and exit nodes with the corresponding return nodes. Data flow within each procedure is summarised by *reaching edges* between the procedure control points such as entry, exit, call and return for formal and actual parameters. Callahan [33] presents iterative algorithms to solve a variety of data flow problems such as whether the values of actual reference parameters *may-be-preserved* over a procedure call. The program summary graph and the algorithms that use it for flow-sensitive data flow analysis were developed to reengineer existing Fortran programs for parallel environments.

The Interprocedural Flow Graph

An extension of the program summary graph, the *interprocedural flow graph* [96] allows the calculation of interprocedural definition-use pairs [5], by providing information at each node about the locations of definitions and uses of reference parameters and global variables that can be reached across procedure boundaries. Two algorithms compute the *reaching definitions* and *reachable uses* (i.e., the problem of determining the set of variable uses that can be reached from each program point), by propagating the intraprocedural data flow information throughout the program guided by the edges in the graph. To ensure that data flow information is propagated only over possible execution paths in the program, new edges connecting call and return nodes (*interprocedural reaching edges*) are used to preserve the calling context of the called procedures during the interprocedural data flow analysis. The resulting sets of reaching definitions and reachable uses can be used to compute definition-use pairs for integration testing [97].

The Program Dependence Graph

The *program dependence graph* [81] is an intraprocedural program representation consisting of two superimposed subgraphs, the *control dependence subgraph* and the *data dependence subgraph*. In the control dependence subgraph nodes represent program's statements and edges represent control dependencies[4] between statements. An algorithm to construct the

[4] Given a control flow graph of a program, a node m is *control dependent* on a node n if and only if n has two outgoing edges, where one of them always results in m being reached, while the other edge may result

control dependence graph from the control flow graph has been shown in [81]; the time complexity of the algorithm in $O(n^2)$, where n is the number of nodes in the control flow graph. For structured programs, the control dependence graph corresponds to the control structure *nesting tree* [51], a program representation where internal nodes represent control structures and leaves represent elementary statements. The data dependence subgraph contains several types of edges representing different data dependence relations [122], enclosing definition-use pairs [5]. The program dependence graph was first introduced as an internal representation for optimising and parallelising compilers [81]. Its role in a software development and maintenance environment has been outlined by Ottenstein and Ottenstein [138].

The System Dependence Graph

An extension of the program dependence graph, the *system dependence graph* [103] combines dependence graphs for individual procedures with additional nodes and edges making up the call interfaces. Nodes are added to a procedure's dependence graph to model parameter passing by value-result. Each call-site node is connected to the entry node of the called procedure. Moreover, the call-site contains *actual-in* and *actual-out* nodes for each actual parameter, while *formal-in* and *formal-out* nodes are linked to the procedure's entry node for each formal parameter. Binding edges connect actual-in with formal-in nodes and formal-out with actual-out nodes. The system dependence graph has been introduced by Horwitz *et al.* [103] for interprocedural slicing. The original interprocedural slicing algorithm proposed by Weiser [176] suffers of lack of precision, because it fails to account for the calling context of a called procedure. *Interprocedural data flow edges* modelling transitive data dependencies between actual parameters across a procedure call are then added to the system dependence graph and exploited, in order to preserve the procedure calling context during interprocedural slicing. This edges permit a more precise computation of a slice across procedure boundaries.

2.2.3 Combining Features of Different Representations

In a reuse reengineering environment and in general in a software maintenance environment, different program representations can be necessary in order to solve different problems. Program representations have been proposed in the literature that combine the features of several representations.

in m not being reached [81].

The Unified Interprocedural Graph

Harrold and Malloy [98] identified the problem that a maintenance environment needs information contained in different intermediate program representations. Instead of incorporating each of the existing representations and using the associated algorithms to develop program maintenance tools, they propose to use the *unified interprocedural graph*, an interprocedural program representation that integrates the features of four different representations: the call graph, the program summary graph, the interprocedural flow graph and the system dependence graph. Algorithms developed for each of these program representations are applicable to the unified interprocedural graph by simply considering subsets of nodes and edges. The main benefits of this approach are the reduction in storage space, deriving from the elimination of redundant information contained in the different representations, and the convenience of accessing a single program representation. Moreover, the construction of the unified interprocedural graph can be obtained incrementally.

The Web Structure

A similar approach has been followed in the *Web structure oriented Software Development Workbench* (WSDW) [71]. In WSDW, software development and maintenance tools are integrated through sharing the same program representation, called *web structure* [130]. A web structure is a particular *relational structure* [77] originally designed as internal program representation for compilers and interpreters. It consists of a supporting tree structure with labelled nodes which represents the syntax of the program. A second structure is superimposed which consists of labelled edges, called *web edges*, connecting nodes of the underlying tree. Web edges provide semantic information, e.g., about control flow, call interface, scope of variables, and other properties that the tree alone cannot express. Program analysis and transformations are accomplished by rewriting the web representation according to rewriting rules called productions [130]. Software engineering tools in WSDW are written using the tool development language TDL [73] which allows web productions to be expressed graphically. Tools based on the algebraic framework of web transformations can enrich the web representation with new information or transform the program structure. For example, in WSDW a *data dependency analyser* adds data dependence edges [122] to the web representation, while a *program recoder* is used for program restructuring and a *program paralleliser* [74] transforms the web representation of a sequential program into the web representation of an equivalent parallel program. The web structure has also been used as intermediate representation within a reverse engineering environment [69].

2.3 Structural Candidature Criteria

Structural candidature criteria aim to recover software components that satisfy particular structural properties based on a metric model (METMOD methods) or on the type of abstraction to be searched for (METTYP methods). In particular, several candidature criteria for the identification of different types of abstractions have been defined and used within the RE^2 project [42]. These methods make use of structural reverse engineering techniques [50] to extract a set of software components from code and make up instances of a predefined model of the abstraction's type. In some cases the type of abstraction to be recovered and a metric model are combined in order to achieve more accurate modules [38, 39]. Usually, a concept assignment process [18] has to be applied in order to associate the extracted components and the related relationships with human oriented concepts. Such a process allows the selection of the set of meaningful software components (the components that actually implement software abstractions) among the set of the recovered components, for the following reengineering phase. This is also useful to validate the candidature criterion [53].

In general a structural candidature criterion for identifying reuse-candidate software components consists in instantiating a model

$$(P, CF, sf : P \to 2^{CF}, M, caf : 2^{CF} \to 2^{CF \times M})$$

where:

- P is a set of *programs.*
- CF is a set of *code fragments* of programs in P.
- sf is a *selection function.* Given a program $p \in P$ it selects the subset $cf \subseteq CF$ of code fragments of P which satisfy particular structural properties. These code fragments are candidate to be reengineered and reused.
- M is a set of human-oriented meanings about the domain of the programs in P.
- caf is the *concept assignment function.* Given a program $p \in P$, let $cf = sf(p)$ be the set of reuse-candidate code fragments of P. The function caf selects the subset of fragments $cf' \subseteq cf$ that can be associated with a meaning in M. These code fragments only will be reengineered and reused.

In the following we illustrate some structural candidature criteria proposed in the literature.

2.3.1 Metric Based Candidature Criteria

Metric based candidature criteria are based on a software *reusability attributes model* which attempts to characterise those attributes directly through measures of an attribute, or indirectly through automatable measures of evidence of an attribute's existence [32]. A set of acceptable values are defined for each metrics. These values can be either simple ranges of values (e.g., measure α is acceptable between α_1 and α_2) or more sophisticated relationships among different metrics (e.g., measure α is acceptable between α_1 and α_2, provided that measure β is less than β_0). The extremes of each measure depend on the application, the environment, the programming language, and many other factors not easily quantifiable. Therefore, the ranges of acceptable values for the measures are usually experimentally determined.

Caldiera and Basili [32] propose a reusability model for identifying reuse-candidate components based on the attributes of *reuse costs* (they includes costs for extracting the component from the old system, packaging it into a reusable components, finding and modifying the component, and integrating it into the new system), *usefulness* (functional usefulness is affected by both the commonality and the variety of the functions performed by the component), and *quality* (such as correctness, readability, testability, ease of modification, and performance of a component). The model uses four metrics:

Halstead volume. The first metric used in the reusability model is the Halstead volume [93] defined as:

$$V = (N_1 + N_2)\log_2(\eta_1 + \eta_2)$$

where η_1 is the number of the different *operators* used in the program (e.g., arithmetic operators, decisional operators, assignment operators, functions, etc.), N_1 is the total number of occurrences of operators in the program, η_2 is the number of different *operands* (e.g., constants, variables, etc.) defined and used in the program, and N_2 is the total number of occurrences of operands in the program. The component volume affects both reuse costs and quality. Both a lower and an upper bound are needed in the reusability model to discard too small components (the reuse costs of such a component exceed its intrinsic value) and too large components (such a component is more error prone and has lower quality).

Cyclomatic complexity. The McCabe cyclomatic number [132] is used to measure the complexity of the control flow of a component and is defined as:

$$v(G) = e - n + 2$$

where e and n are the number of edges and nodes, respectively, in the control flow graph G. The component complexity affects reuse cost and quality. As for the Halstead volume, the reusability model needs both a lower and an upper bound: the reuse of components low complexity may mot repay the reuse cost, whereas high component complexity may indicate poor quality. Moreover, high complexity with regularity of implementation suggests high functional usefulness.

Regularity. The economy of a component's implementation, or the use of correct programming practices can be measured based on some regularity assumptions and using the Halstead Software Science Indicators [93]. If the actual length of a component is:

$$N = N_1 + N_2$$

and the estimated length is:

$$\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$$

the closeness of the estimate is a measure of the regularity of the component's coding:

$$r = 1 - \frac{N-\hat{N}}{N} = \frac{\hat{N}}{N}$$

Component regularity measures the readability and the non-redundancy of a component's implementation. Therefore, components whose regularity is in the neighbourhood of 1 are candidate to be reused.

Reuse frequency. The reuse frequency of a component can be estimated by comparing the number of static calls addressed to the component with the number of calls addressed to a class of components assumed to be reusable. For example, let $n(X)$ denote the number of static calls addressed to a component X in a system. If $S_1, \ldots, S_N$ are the standard components (i.e., predefined in the standard environment) and C is a user-defined component used in the system, the reuse-specific frequency of the component C can be defined as:

$$v_\sigma(C) = \frac{n(C)}{\frac{1}{N}\sum_{i=0}^{N} n(S_i)}$$

The reuse-specific frequency is an indirect measure of the functional usefulness of a component.

2.3.2 Functional Abstraction Candidature Criteria

Functional abstraction is the first form of abstraction implemented in high level languages. Traditional programming languages (in which the majority of the existing software is written) provide primitives (that we call with the generic name of procedures) for the implementation of functional abstractions. The search for code implementing functional abstractions can be based on the primitives of isolation, aggregation and generalisation [42].

Isolating Reusable Functions Using Program Slicing

The need for searching for functional abstractions by means of isolation is related to the lack of reusability as a quality attribute to be taken into account when developing a software system. Very often this lack of reusability fails to respect the fundamental principle of software engineering, according to which every procedure must implement one and only one abstraction.

Two different techniques have been proposed in the literature [42] to isolate functionalities within a procedure: horizontal isolation (i.e., each piece of isolated code is a block of the procedure's text) and vertical isolation (i.e., each piece of isolated code is a set of statements that lie on a same dynamic path of the procedure). The horizontal isolation may be founded on primes [10, 79] and assumes the original procedure to be structured. Although methodologies and tools are available for restructuring software with respect to a set of one-in/one-out primes [8, 23, 48, 144], very often horizontal decomposition is not suitable for isolating functions because of the *interleaving* of code fragments responsible for accomplishing more than one purpose in the same section [153]. This can arise either intentionally (for example, in optimising a program, a programmer may use some intermediate result for several purposes) or unintentionally, due to patches, quick fixes, or other hasty maintenance practices. In such cases vertical isolation can be used and supported by techniques such as *program slicing* [176].

Program slicing has been introduced by Weiser [174, 176] as a powerful method for automatically decomposing a program by analysing its control and data flow. Program slices help programmers to better understand programs while debugging [175] and can be used to parallelise sequential programs [176]. In the Weiser's definition a *slicing criterion* of a program P is a pair $\langle s_{out}, V_{out}\rangle$ where s_{out} is a statement in P and V_{out} is a subset of variables in P. A *slice* of a program P on a slicing criterion $\langle s_{out}, V_{out}\rangle$ consists of all statements and predicates of P that might affect the values of the variables in V_{out} just before the statement s_{out} is executed. The program dependence graph [81] can be used as representation for implementing efficient slicing algorithms both at intraprocedural [138] and interprocedural [103] level. A slice is computed by backward traversing the control and data dependence edges of the graph and corresponds to the subgraph containing the reached vertices and edges. In order to exploit these algorithms, all the variables in the set V_{out} of a slicing criterion $\langle s_{out}, V_{out}\rangle$ are required to be used at s_{out}.

Horwitz *et al.* [103] also defined *forward slicing*: given a a program point p and a variable v, a *forward slice* with respect to the slicing criterion $\langle p, v\rangle$ consists of all statements and predicates of the program that might be affected by the value of v at point p. Gallagher and Lyle [86] introduced the concept of *decomposition slice* and outlined its role in software maintenance and testing. The decomposition slice with respect to a variable v consists of all the statement and predicates of the program that might affect the values of the variable v, regardless of any program point. Program slicing has been exploited in data-flow testing, thanks to the definition of the *dynamic slicing* [117, 118]. A *dynamic slice* on a slicing criterion $\langle X, s^q_{out}, V_{out}\rangle$ is an executable part of a program whose behaviour is identical, for the same set of input data X, to that of the original program with respect to the subset of variables of interest V_{out}, at some execution position q[5]. Moreover, program slicing has been used for integration testing [97] and regression testing [90]. Finally, Venkatesh [169] proposes a hybrid approach to program slicing, called *quasi static slicing.* Quasi static slices fall between static slices [176] (which preserve a subset of the behaviour of the original program for all possible inputs to the slice) and dynamic slices [118] (which are constructed for a particular execution of the program and are required to preserve the subset of the behaviour of the original program for just one specific input state). Indeed, in quasi static slices some of the input values are fixed, while the behaviour of the program must be analysed and understood when other input values vary. Quasi static slicing has been generalised and used in a process of semantic equivalent program transformations for program comprehension [95].

Due to its feature of program decomposition technique, program slicing has been used as structural method for recovering reusable components from existing software by isolation. A slicing based structural model for isolating functional abstractions can be instantiated with reference to Ottenstein's definition of slicing criterion. In this case the slicing algorithm is the selection function, the variables in the set V_{out} of the slicing criterion correspond to the output data of the function to be isolated and the selected code fragments (program slices) satisfy the transitive closure on the control and data-flow dependencies from the program point s_{out}. For example, Ning *et al.* [136] use program slicing within the tool COBOL/SRE to segment large legacy systems written in COBOL. In particular, they use both backward and forward slicing techniques.

Lanubile and Visaggio [124] give two definitions of slicing for identifying and extracting

[5] The statement s_{out} holds the position q on the program execution path obtained with input X.

two main kinds of components in legacy systems written in COBOL: environment-dependent operations and domain-dependent functionalities. The former, environment-dependent components, depend on the technological environment which holds a system and usually consist of basic operations on the database, report production, displaying of interface maps, or user-machine dialogue. On the other hand, domain-dependent components characterise a class of problems in the same application domain and typically consist of computational formulas or business rules.

Environment-dependent components are identified and extracted by searching for *direct slices.* A direct slice of a program P on a slicing criterion $\langle s_{out}, V_{out}\rangle$ consists of all statements and predicates of P that *directly* contribute to the values of V_{out} just before the statement s_{out} is executed (this includes the set SS of all statements that define the *living* [5] value of any variable in V_{out} at the program point s_{out} and all predicates that give rise to the alternative control flow paths on which the statements in SS lie). Direct slices can be used to identify structural information distributed throughout the program and to extract a *source* (input) or *sink* (output) *module* [124] encapsulating that information. Human interaction is required both for identifying a slicing criterion, and refining the slice extracted, whenever this is too complex to be clustered in one module [65].

Domain-dependent components are identified and extracted by searching for *transform slices.* A transform slice of a program P on a slicing criterion $\langle s_{out}, V_{in}, V_{out}\rangle$, where s_{out} is a program statement and V_{in} and V_{out} are subsets of program variables, consists of all statements and predicates of P that might affect the values of the variables in V_{out} starting from the values of V_{in}, just before the statement s_{out} is executed. The computation of a transform slice terminates as soon as the definition of each of the variables in V_{in} is found. Transform slices can be packaged in *transform modules* [124], i.e., components whose main purpose is to transform data into some other form. A different approach to the extraction of domain-dependent functions has been proposed by Cutillo et al. [66]. Given a slicing criterion $\langle s_{out}, V_{in}, V_{out}\rangle$, the slice to be clustered into the transform module is computed as $S(V_{out}) \setminus S(V_{in})$, where $S(V)$ denotes the decomposition slice [86] on the set of program variables V. However, the accuracy of this technique is lower with respect to the transform slice algorithm presented in [124]. Indeed, the absence of a program point in the slicing criterion [86] may cause the addition of extraneous statements in the transform module or the deletion of necessary instructions.

Canfora *et al.* [45] use program slicing to isolate the external (user) functionalities of large monolithic programs. Slicing criteria of type $\langle p, v\rangle$, where p is an output statement and v is a program variable referenced at p, are used to extract *output-including slices* [45]. A *functional slice* [45] corresponding to an external functionality is obtained as the union of more output-including slices. When a functional slice has been obtained, a comparative analysis of the software documentation and of the slice must be performed in order to associate the latter with a meaningful user functionality. Program slicing and concept assignment process are alternated into an iterative process in order to refine the functional slice. A functional slice can be decomposed in source, transform and sink modules [124], by isolating the statements that get the external input (to be clustered in the source module) and return the external output (to be clustered in the sink module) from the statements that compute the function (these statements will compose the transform module). However, the transform module can still be too complex, because it embeds several internal functionalities. In this case functional slices can be decomposed into more elementary internal subfunctionalities by a process of slice's intersection [45]. Moreover, a further decomposition of a functional slice can be obtained by intersecting the component output-including slices. The type of result obtained by intersecting slices is an index of the degree of cohesiveness of a module [137]. A case study [45] showed that only modules that exhibit communicational or sequential cohesion [182] can be decomposed into potentially reusable components.

Aggregating Procedures on the Call Graph

The need for searching for functional abstractions by means of aggregation is related to the poor quality of the high level design. Very often this poor quality of design means that it is impossible to recognise the high level functions that a set of procedures implements and, thus, to create a software system as a set of potentially reusable modules [42]. The aggregation entails the identification of sets of procedures with high functional cohesion [182], each set being a candidate to create a reusable module.

The most elementary form of aggregation consists of searching the call graph for particular sub-graphs, such us strongly connected sub-graphs, trees, one-in/one-out sub-graphs. More sophisticated forms of aggregation can be obtained by transforming a call graph into a tree. The transformation of a call graph into a tree is a reverse engineering process to abstract a high level design document, essentially a structure chart, from a low level representation of the calls in a software system [13].

Cimitile and Visaggio [55] propose a candidature criterion for the identification of functional abstractions based on the transformation of the call graph of the system into a tree by exploiting the dominance relation [100] among the nodes of the call graph. In [55] a program's call graph $(s, PP, CALL)$[6] is referred to as *Call Directed Graph* CDG. The *Call Directed Acyclic Graph* (CDAG) is obtained from the program CDG by collapsing each

[6] PP is the set of procedures, $s \in PP$ is the main procedure and $CALL$ describes the activation relation on $PP \times PP \setminus \{s\}$.

strongly connected subgraph[7] into a single node.

In a CDAG a procedure p_x *dominates* a procedure p_y if and only if each path from s to p_y contains p_x. The reflexive and transitive closure of the dominance relation on the CDAG is the direct dominance relation. A procedure p_x *directly dominates* a procedure p_y if and only if p_x dominates p_y and all the procedures that dominate p_y dominate p_x too. A procedure p_x *strongly and directly dominates* a procedure p_y if and only if p_x directly dominates p_y and p_x is the only procedure that calls p_y.

The direct dominance relation can be represented as a tree, called the *Direct Dominance Tree* (DDT), whose root is the main procedure s. The *Strong and Direct Dominance Tree* (SDDT) is obtained from DDT by marking all the edges representing the strong and direct dominance relation. The set of the subtrees of a SDDT can be divided in two subsets, the subset MET of the subtrees containing only marked edges and the subset $UMET$ of the subtrees containing at least an unmarked edge. The *Reduction of the Strong Direct Dominance Tree* (RSDDT) is a tree obtained from the SDDT by collapsing each subtree in MET into a unique node.

Four rules have been proposed to aggregate procedures into reuse-candidate modules and to identify the *uses* and *is composed of* relationships [89] between them:

1. The set of procedures represented by the nodes of a strongly connected subgraph of a CDG is a candidate to constitute a reusable module.

2. The set of procedures represented by the nodes of a subtree $t \in MET$ is a candidate to constitute a reusable module represented by the root of t.

3. The set of procedures represented by nodes of a subtree $t \in UMET$ linked to the root of t by a marked edge is a candidate to constitute a reusable module. This module *uses* the modules represented by the nodes in t which are linked to the root by an unmarked edge.

4. Each of the marked (unmarked) edges of RSDDT is a candidate to constitute an *is composed of* (*uses*) relationship between the modules represented by the nodes that the edge links.

The criterion has been validated both in Pascal [53] and COBOL [46] environment.

[7] A strongly connected subgraph of a CDG contains at least one cycle involving all its nodes. This cycle is due to the presence of recursion between the procedures of the program.

Generalising Functions

The generalisation of a function implemented by one or more procedures increases the likelihood of the function to be reused. The generality of a function is often viewed as an index of its reusability [25].

A first elementary form of generalisation of a function consists of parametrising it. For example, a procedure for array sorting parametrised with respect to the length of the array is more general than one for arrays with a fixed length. However, excessive parametrisation should be avoided because a function with too many parameters may be difficult to use and to maintain [42]. Often a function that can be generalised is identified by locating instances of *near-duplication* in a software system [9, 58, 104], i.e., sections of code that are textually identical except for systematic substitution one set of variable names and constants for another. An experiment showed that 12% of the X Window System [155] was composed of duplicated code that could be removed by rewriting the system [9].

A more interesting form of generalisation consists of generalising the type of information that a function handles [42]. To obtain such a type of generalisation it is required to record the procedure that implements the generic function as a skeleton; the designer who is going to use the component must first instantiate it to the required type. Modern languages, such as Ada [24], have syntactic primitives to write generic functions and instantiate them. However, most of the existing software systems are written in traditional that do not allow procedures to be written in a generic way. For these languages a formalism to express function skeletons can be defined and a library of procedures with generic parameters can be created [52]. The instances of one of the procedures in the library can be obtained by editing operations. To make sure that errors are not made during the instantiation process an environment should be created that accepts the description of the instantiation operations in a high-level user-oriented formalism and performs the corresponding low level operations [42].

2.3.3 Data Abstraction Candidature Criteria

Several structural candidature criteria for searching data abstractions in code written using procedural languages have been proposed in the literature [36, 37, 38, 43, 47, 76, 85, 99, 128, 129, 181]. A survey can be found in [167]. In the following we will illustrate some of the criteria proposed within the RE^2 project [42].

Identifying Objects

A logic based method for identifying objects (or data structures) has been proposed by Canfora and Cimitile [36]. This approach looks for the set of procedures PP, the set of global variable DD and the set of references of procedures to global variables, which can be expressed by a relation $DAT \subseteq PP \times DD$. It is worth noting that this relation can also be represented as a bipartite graph called *variable-reference* graph [34]. The criterion is based on two rules:

1. Two global variables d_1 and d_2 contribute to define the same candidate object if they belong to a reference cobweb, i.e. if and only if[8]

$$(d_1, d_2) \in DSDAT = (trans(DAT)\ DAT)^* \subseteq DD \times DD$$

2. A procedure c defines one of the methods of the object to which a global variable d belongs if c directly references one of the variables in the cobweb around d, i.e. if and only if

$$(d, c) \in PPDAT = (trans(DAT)\ DAT)^*\ trans(DAT) \subseteq DD \times PP.$$

A different criterion has been proposed [38] and validated in C environment [39] which treats undesired pairs in DAT, called *coincidental* and *spurious* connections[9], that produce clusters of procedures and functions implementing more than one object. The candidature criterion considers for each $p \in PP$ the subgraph generated by clustering together the set $DD(p)$ of global variables p references and the set $PP(p)$ of procedures that only access these data items. These sets can be defined as:

$$\begin{aligned} DD(p) &= PostSet(p) \\ PP(p) &= \bigcup_{d \in DD(p)} P(d,p) \end{aligned}$$

where,

$$\begin{aligned} P(d,p) &= \{p_i \in PP \mid p_i \in PreSet(d) \wedge PostSet(p_i) \subseteq PostSet(p)\} \\ PreSet(d) &= \{p \in PP \mid (p,d) \in DAT\} \\ PostSet(p) &= \{d \in DD \mid (p,d) \in DAT\} \end{aligned}$$

[8] $trans(R)$ and R^* denote the transpose and reflexive transitive closure of the relation R, respectively.

[9] A procedure which implements more than one function, each function logically belonging to a different object, generates coincidental connections. A procedure which implements system specific operations by directly accessing the supporting data structure of more than one object generates spurious connections. Coincidental connections can be eliminated by slicing the procedure and isolating the different functions. Spurious connections can be eliminated by deleting the procedure from the set PP.

For such a subgraph the index $IC(p)$ defining its *internal connectivity* and the variation $\Delta IC(p)$ in the internal connectivity, due to the possible clustering with respect to procedures in $PP(p)$, are calculated:

$$\begin{aligned} IC(p) &= \frac{\sum_{d \in PostSet(p)} \#P(d,p)}{\sum_{d \in PostSet(p)} \#PreSet(d)} \\ \Delta IC(p) &= IC(p) - \sum_{d \in PostSet(p)} \frac{\#\{p_i \mid PostSet(p_i) = \{d\}\}}{\#PreSet(d)} \end{aligned}$$

where $\#A$ denotes the number of elements in the set A.

The procedures whose associated ΔIC is greater than a given threshold are used to generate clusters. All the other routines are considered to introduce coincidental or spurious connections and are sliced or deleted, respectively. Moreover, some of the data items are merged into a unique item. These operations generate a new *variable-reference* graph on which the indexes are recalculated and the operations reexecuted. The process ends when the graph is partitioned into a set of isolated sub-graphs, each of which consists of one data node (corresponding to a set of global variables) and a collection of procedures that only access it. Each one of these isolated sub-graphs defines a candidate object.

Abstract Data Types Candidature Criteria

A logic based method for identifying abstract data types has been proposed by Canfora *et al.* [37]. This approach looks for the set of procedures PP, the set of user defined data types TT and the set of uses of user defined data types in the headings of the procedures[10], which can be expressed by the relation $TYP \subseteq PP \times TT$. To take into account the relationships possibly existing among the different user defined data types used in the heading of a procedure, this relation is refined by the relation $STYP \subseteq TYP$ containing the pairs (p, t) such that p does not use any super-type[11] of t in the interface. The criterion is based on two rules:

1. Two user defined data types t_1 and t_2 contribute to define the same candidate abstract data type if they belong to a cobweb of formal parameters declarations, i.e. if and only if

$$(t_1, t_2) \in ABTYP = (trans(STYP)\ \ STYP)^* \subseteq TT \times TT$$

[10] A procedure p uses a user defined type t if and only if t is used to define the type of a formal parameter or a return value of p [37]

[11] The user defined data type t_1 is a super-type of the user defined data type t_2 if t_2 is used to define t_1 [129].

2. A procedure p defines one of the operators of the abstract data types to which a user defined data type p belongs if p uses one of the user defined data types in the cobweb around t to declare a formal parameter, i.e. if and only if

$$(t,p) \in PPTYP = (trans(STYP)\ STYP)^{*}\ trans(STYP) \subseteq TT \times PP.$$

The criterion above has been applied to a set of case studies written in Pascal [40]. Although the experiment gave satisfactory results, some modules were too large and difficult to understand and associate with data abstractions. This was in particular caused by a extensive use of user defined sub-range or enumeration types in the headings of procedures. Such user defined types were used in some procedures that could not be associated with operators of an abstract data type and procedures that could be associated with operators of different abstract data types. The criterion was also affected by another problem which caused lack of precision: procedures contributing to the implementation of an abstract data type, but that did not use any user defined data type (for example, procedures implementing a subfunction of an operator of an abstract data type), were not selected for the candidate module. To overcome these problems, an improved criterion has been proposed by Canfora *et al.* [43].

The problem of complex modules has been solved by identifying and deleting the user defined sub-range or enumeration types that caused large clusters of procedures and user defined data types. In this way complex modules can be divided in simpler ones that can be associated with human oriented concepts. To include all the procedures involved in the implementation of an ADT the previous method has been combined with an iterative process based on the SDDT of the system. At each step, the process deletes from the CDAG the procedures that do not belong to some candidate modules and that are strongly and directly dominated by the main procedure. Moreover, new call edges are inserted between the main procedures and procedures that do not have any incoming edge and the SDDT is reconstructed. The last SDDT, so obtained, is only constituted by procedures that implement candidate ADTs. Moreover, the dominance relationships on this SDDT can be used to identify the structure of a module implementing an ADT (the procedures exported, i.e., the operators of the ADT, and those only used in the body for implementation purpose) and the *uses* relationships between modules.

The details of the improved criterion can be found in [167]. The criterion has been applied to the same set of case studies used to validate the former criterion [54]. The results of the experiments were characterised by a greater number of reusable modules of higher quality.

2.4 Specification Driven Candidature Criteria

The concept assignment function caf is very important to validate a structural method [53], because it selects the subset cf' of reuse candidate code fragments from the set cf produced by the selection function sf. The adequacy of a structural method can therefore be measured as the percentage of elements in cf which also are in cf' [53].

On the other hand, specification driven methods presuppose the existence of information about the specification of the abstraction to be recovered and their application does not need to be followed by a concept assignment process. The specification of the abstraction can be given in different ways. For example, it can be expressed in a formal way [41, 59] or as a set of test cases capturing the behaviour of the abstraction [92, 177, 178]. A widely used approach is to build up a library of programming *plans* [160] or *clichés* [148] (each of which corresponds to an abstraction) and to search for their instances in code [3, 115, 120, 142, 146, 180].

In general a specification driven method consists in instantiating a model

$$(P, CF, S, sf : P \times S \rightarrow 2^{CF})$$

where:

- P is a set of *programs.*
- CF is a set of *code fragments* of programs in P.
- S is a set of specifications.
- sf is a *selection function.* Given a program $p \in P$ and a specification $s \in S$ it selects the subset $cf \subseteq CF$ of code fragments of P which satisfy particular structural properties and implement the specification s. These code fragments are candidate to be reengineered and reused.

The adequacy of a specification driven method is very high [44]. In the following we will illustrate some of the methods proposed in the literature.

2.4.1 Formal Specifications for Candidature Criteria

Formal specifications are generally beneficial because a formal language makes specifications more concise and explicit [27]. These techniques help the software engineer acquire greater insights into the system design, dispel ambiguities, maintain abstraction levels, and determine both his approach to the problem as well as its implementation. Formal specification languages such as Z [162] and VDM [108] have been used in several software development projects [27, 28]. Reverse engineering techniques have also been exploited for recovering a formal specification from code [2, 3, 38, 57, 87, 171].

Canfora et al. [41] propose three specification driven candidature criteria for isolating code fragments implementing functional abstractions in large programs. The function to be isolated is partially specified in term of its sets of *input* and *output* data and first order logic formulas, called *precondition* and *postcondition* [101]. The precondition expresses the constraint which must hold on the input data to allow the execution of the function, while the postcondition expresses the condition which will hold on the output and input data after the execution of the function (i.e., it relates the output data to the input data). Other conditions that bind the execution of a function in the context of a program (for example, conditions on a variable carrying a selection from a menu of several functions that can be executed) are also considered. The formal specification of the function is used together with structural techniques based on the program dependence graph [81]. Human interaction is required to trace the data and conditions of the specification into the program variables and predicates.

Three type of *conditioned software components* are identified and extracted: *conditioned functions.*, *conditioned programs* and *conditioned slices.*

Conditioned function. Given a program P and a condition C, the conditioned function $F(C)$ consists of all the statements and predicates in P that are control dependent [81] on a program predicate implementing the condition C. A control dependence graph traversal algorithm is presented which searches for a predicate node p corresponding to the precondition or a binding condition of the required functional abstraction. The nodes which are control dependent on p are candidate to implement the functional abstraction and are isolated and extracted. Human interaction is required to identify the program predicate implementing the condition.

Conditioned program. Given a program P and a condition C, the conditioned program $P(C)$ is an executable program containing all the statements and predicates of P that can be executed when the condition C holds true. An algorithm traversing the control dependence graph is used to identify all the nodes corresponding to statements and predicates of P that can be executed whenever a given condition holds true. This method is useful to isolate different behaviours of a program modules. These behaviours can be specified by a postcondition composed of the disjunction of several conditions, each of which corresponds to one behaviour. The software engineering is asked for proving implications between the condition and the program predicates, in order to discard execution paths.

Conditioned slice. Conditioned slicing is a generalisation of the *quasi static* slicing paradigm [169], where a general condition substitutes the set of value assigned to a subset of input data. A conditioned slicing criterion is of the form $\langle s_{out}, V_{out}, C\rangle$, where s_{out} is a program statement, V_{out} is a set of program variables and C is a condition. A conditioned slice of a program P on a conditioned slicing criterion $\langle s_{out}, V_{out}, C\rangle$ consists of all the statements and predicates of P that might affect the values of the variable in V_{out} just before the statement s_{out} is executed, when the condition C holds true. As program slicing can be used to identify reusable functions in code, conditioned slicing can be used for identifying different behaviour of a functional abstraction implemented by a program slice. An two phase algorithm is exploited which first compute the conditioned program $P(C)$ and then the conditioned slice by exploiting the program dependence graph of $P(C)$.

These methods can be improved by using formal method tools [28], such as a symbolic executor [64] and a theorem prover [29].

2.4.2 Candidature Criteria Based on Test Cases

A way to provide the specification of a functionality is by carefully designing a set of test cases for a program. The set of test cases expresses a behaviour of the program corresponding to an external functionality.

Mapping test cases to code

Wilde et al. [177, 178] considers the problem of locating functionalities in code as the identification of the relation existing between the ways the user and the programmer see the program. From the user point of view, a program consists of a collection of, possibly overlapping, functionalities:

$$FUNCS = \{f_1, f_2, ..., f_N\}$$

while the programmer's view consists of a collection of program components:

$$COMPS = \{c_1, c_2, ..., c_M\}$$

The problem is the identification of the components in $COMPS$ which contribute to implement a functionality in $FUNCS$, i.e., the construction of a relation $IMPL \subseteq COMPS \times FUNCS$. The link between components and functionalities may be provided by test cases. A test case T_i exhibits a set of functionalities $F(T_i) = \{f_{i,1}, f_{i,2}, ...\}$ which can be identified by a system user. On the other hand, a test case also exercises a set of program components $C(T_i) = \{c_{i,1}, c_{1,2}, ...\}$ which can be identified by instrumenting the code and monitoring its execution.

Both a *deterministic* and a *probabilistic* techniques have been proposed to analyse the traces resulting from the program execution [177].

Probabilistic formulation. This approach is motivated by a statistical view of the problem. Given from a sample of test cases extracted from a population composed of every possible test sequences for the program, the approach tries to identify the *best indicators* of a given functionality. Let $p_{f,c}$ indicates the conditional probability that a test case that exercises the component c also exhibits the functionality f, i.e.:

$$p_{f,c} \quad = \quad P(f \mid c) \quad = \quad P(f,c)/P(c)$$

The best indicator components for a given functionality f are those having the greatest $p_{f,c}$. If $T_1, T_2, ...,$ are a random sample of test cases, an estimator of $p_{f,c}$ is

$$\hat{p}_{f,c} \quad = \quad Freq(f,c)/Freq(c)$$

The *implements* relation can be constructed as:

$$IMPL_z \quad = \quad \{(c,f) \in COMPS \times FUNCS \mid \hat{p}_{f,c} \geq z\}$$

for some threshold $0 \leq z \leq 1$.

Deterministic formulation. In the deterministic approach the *implements* relation for a given functionality f is builded by taking all the components exercised in tests cases exhibiting f and subtracting out those components exercised in the remaining test cases, i.e., the resulting relation would satisfy, for some T:

$$IMPL'(c,f) \quad \Longrightarrow \quad c \in C(T) \quad \text{and} \quad f \in F(T)$$

and as well there would be no T' such that:

$$IMPL'(c,f) \quad \Longrightarrow \quad c \in C(T') \quad \text{and} \quad \neg f \in F(T)$$

where the symbol $\neg$ denotes the logical *not*. The relation $IMPL'$ is simply a limiting case of the $IMPL_z$, as it can be easily proved [177] the equivalence of:

$$IMPL_{1.0}(c.f) \quad \Longleftrightarrow \quad IMPL'(c,f)$$

The method was experimented in a set of case studies [178] conducted on software systems written in C language [110]. The probabilistic approach achieved slightly better results than the deterministic one.

While this candidature criterion is cost-effective, very practical and easy to implement and use, it is only good to find components that are unique to a particular functionality. In general, the method lacks in precision, because the software component identified could be too large and include more functionalities than the one sought. In some cases, the set of statements exercised by a set of test cases might enclose the whole program.

Combining test cases and program slicing

A more precise method by Hall [92] combines the use of a set of test cases with program slicing. The method is called *simultaneous dynamic program slicing* because it extends and simultaneously applies to a set of test cases the *dynamic slicing* technique [117, 118] which produces executable slices that are correct on only one input. The basic idea of this approach is that the end-user specifies which test cases illustrate the desired behaviour; the system then computes a simultaneous dynamic program slice that is guaranteed to be correct on the indicated test cases. This method takes into account the data flow of the program and then allows the reduction of the set of selected statements. Indeed, only the statements that might affect the values of the output variables of the function on the exercised paths will be considered.

A simultaneous program slice on a set of test cases is not simply given by the union of the dynamic slices on the component test cases. Indeed, simply unioning dynamic slices is unsound, in that the union does not maintain simultaneous correctness on all the inputs [92]. An iterative algorithm is presented that, starting from an initial set of statements, incrementally constructs the simultaneous dynamic slice, by computing at each iteration a larger dynamic slice. The algorithm has been implemented in ISAT (Interactive Specification Acquisition Tool) [92], a multi-functional apprentice system to aid the human developer in acquiring, validating, implementing, maintaining, and reusing a rule-based reactive system model consistent with an evolving set of requirements. Experiments conducted on a set of six case studies indicate that the method produces significantly smaller subsets than three competing approaches: *static slicing* [176], *all-executed-code* on the set of test cases, and the intersection of the outputs of the first two techniques.

However, the method does not consider the problem of finding a slicing criterion and then it can only be used to identify external functionality. Lack of precision can result in identifying internal functions when dealing with large programs. Moreover, whenever a slicing criterion has not been adequately selected, the method might produce slices containing more functionalities than the one expected or even not containing all the statements of the searched functional abstraction. In the next chapter we will show how the use of a formal specification and of symbolic execution helps in defining a suitable slicing criterion, allowing a better precision in identifying expected functions.

2.4.3 Knowledge Based Candidature Criteria

Several specification driven methods for identifying abstractions in code are knowledge-based approaches. They encode the knowledge about the functions to be identified in the form of programming plans and make use of an internal representation of the program for mapping program actions to these plans. The final result is a kind of tree, with program instructions at the leaves, programming plans in the internal nodes, and goals the program achieves as the root. Knowledge based methods can be classified as either top-down or bottom-up [146].

Top-Down Knowledge Based Methods

Top-down methods [115, 120, 121] use the knowledge about the goals the program is assumed to achieve and some heuristics to locate what plans from the library can achieve these goals; then program statements and the plan from the library that can achieve these goals; then they attempt to connect these plans to the actual program statements. Typically, the program representation used consists of an abstract syntax tree [5] annotated with semantic information [115, 121]. Once a plan achieving a program goal has been identified, it is translated into a lower abstraction level representation (usually the same representation as the program). The process includes using matching rules to detect how the statements achieve the various subgoals within a plan, and difference rules to recognise how they differ from the statements expected in a plan.

Pattern matching techniques have been developed by Kontogiannis *et al.* [116] both for *code-to-code* and *concept-to-code* matching. The code-to-code matching uses a dynamic programming algorithm to calculate the distance between two code fragments. The concept-to-code matching uses Markov models to compute similarity measures between an abstract description, expressed in a *concept language*, and a code fragment.

Bottom-Up Knowledge Based Methods

Bottom-up methods [3, 142, 146, 179, 180] are more precise but also more expensive. They start from the program statements and try to identify the plan which can have these statements as components; then they attempt to infer program goals from these plans. Such methods are too expensive because they exhaustively search for plans in a program. For example, Wills [179, 180] encodes the program into a flow-graph and uses a flow-graph grammar as library of clichés. Graph parsing is then exploited to identify all the notable subgraphs in the program's flow-graph. This approach has also been used for recognising a program's design [149]. A different approach proposed by Ross [151] uses symbolic execution to derive an *effects-based* representation of a program, in term of a set of effect-trees. Each effect-tree describes the value a variable can assume through the possible feasible and non-redundant paths. The program representation is then matched against a library of known effect-trees associated with high level abstraction descriptions. Paul and Prakash [142] use a finite state machine-based tool for searching patterns in C code. Source code is compiled into an *attributed syntax tree* (AST), while patterns (expressed by means of a pattern language) are compiled into an extended nondeterministic finite state automata, called *code pattern automata* (CPA). A CPA *interpreter* runs the CPA with the AST as input. A match occurs whenever the CPA reaches a final state.

The time complexity of these methods is exponential because of the NP-completeness of the problem. A solution to limit the number of candidate plans considered during program understanding can be provided by using hierarchical [3] or indexed [146] organisations of the plan library. For example, Quilici and Chin [147] propose a cooperative environment where an automated program understander based on an indexed plan library is augmented with facilities for aiding programmers in extracting and recording additional program design information.

However, the main limitation of knowledge-based methods is that they can require a large library of plans. Moreover, while this approach can be effective for recognising stereotypical domain independent plans, it can be too expensive for dealing with domain dependent functions and then not convenient for a reuse reengineering process. Indeed, to apply a knowledge-based method to software written for a particular application domain we need to design and develop a new plan library for this domain and the cost of such a task might be comparable to the cost of designing and developing a library of reusable modules.

2.5 Summary

In this chapter the background for the work described in this thesis has been discussed. The reference paradigm of the RE^2 project [42] has been outlined. The reference paradigm proposes a systematic approach for setting up a reuse reengineering process. It divides a reuse reengineering process into five sequential phases, each phase being characterised by the objects it produces. The first three phases produce the reusable modules from existing systems, while the latter two phases populate the repository and set up the environment for the retrieval and the reuse of modules during the development of new systems.

The work presented in this thesis is part of the first phase of the RE^2 reference paradigm, called the Candidature phase, which is related to the identification of software components implementing abstractions in existing systems. The main problem of this phase is the definition of a candidature criterion and the model of the system to apply the criterion. Several program representations proposed in the literature and mainly used in software maintenance have been described. Such program representations are suitable to be used for applying different candidature criteria.

Candidature criteria can be distinguished in structural candidature criteria and specification driven candidature criteria. Structural candidature criteria aim to recover software components that satisfy particular structural properties based on a metric model or on the type of abstraction to be sought. A concept assignment process [18] is required in order to associate a human oriented concept with the recovered components. Several candidature criteria have been described based on simple software or searching for software components implementing functional and data abstractions. Specification driven methods presuppose the existence of information about the specification of the abstraction to be recovered and their application does not need to be followed by a concept assignment process. Specification driven candidature criteria using formal specifications, test cases and knowledge bases to describe the abstraction to be sought have been outlined.

Chapter 3

A New Specification Driven Candidature Criterion

In this chapter a new specification driven candidature criterion, called *specification driven program slicing* is presented. The method is based on program slicing as a program decomposition technique for isolating code fragments implementing functional abstractions. A different definition of slicing criterion is given in order to consider a program slice as a procedure. The specification of the functional abstraction to be isolated is used to find a suitable slicing criterion. To this aim symbolic execution [60, 64, 111] and theorem proving [6, 29, 143] techniques are used as a support for the maintainer. In section 3.1 the specification driven slicing process is defined and the main features of program slicing and symbolic execution are outlined. In section 3.2 we show how the process can be specialised for programs written in C language [110].

3.1 Specification Driven Program Slicing

In the previous chapter program slicing has been introduced and structural criteria based on program slicing have been showed. In the Weiser's definition a *slicing criterion* of a program P is a pair $\langle s_{out}, V_{out} \rangle$ where s_{out} is a statement in P and V_{out} is a subset of variables in P. A program slice is therefore an executable code fragment composed of all the statements and predicates of P (starting from the entry of the program) that might affect the values of the variables in V_{out} just before the statement s_{out} is executed. Actually, such a code fragment cannot be always considered as a reusable function. Indeed, the use of program slicing for isolating reusable functions also involves the knowledge of the statements where the computation of the slice must terminate. For example, Lanubile and Visaggio [124] provide information about the input data of the function to be isolated by adding a set of variables V_{in} to the slicing criterion. They also introduce the definition of the *transform slice* of a program P on a slicing criterion $\langle s_{out}, V_{in}, V_{out} \rangle$ as the set of statements and predicates of P that might affect the values of the variables in V_{out} just before the statement s_{out} is executed. The computation of a transform slice terminates as soon as the definition of each of the variables in V_{in} is found. Therefore, a transform slice fails to identify a function if the variables in V_{in} are defined more than once during its execution. A different approach can be followed if we consider that a function should have a unique entry point, i.e., a statement s_{in} which dominates all the other statements on the control flow graph [5, 100]. A new definition of slice can be obtained by adding such a statement s_{in} to the slicing criterion.

Definition 3.1 A slicing criterion of a program P is a triple $\langle s_{in}, s_{out}, V_{out} \rangle$ where s_{in} and s_{out} are statements in P, s_{in} dominates s_{out} and V_{out} is a subset of variables of P.

Definition 3.2 A slice of a program P on a slicing criterion $\langle s_{in}, s_{out}, V_{out} \rangle$ consists of all the statements and predicates of P that lie on a control flow path from s_{in} to s_{out} and that might affect the values of the variables in V_{out} just before the statement s_{out} is executed.

Cimitile and De Lucia [56] have defined algorithms based on the control flow graph [100] and the program dependence graph [81] to compute such a slice.

3.1.1 Finding a Slicing Criterion

The adequacy of a structural method can be measured as the percentage of elements produced that can be associated with a human oriented meaning [53]. While structural candidature criteria have showed good results in several experiments conducted within the RE^2 project [42] for the identification of data abstractions [39, 40, 54] and functional abstractions through the aggregation of procedures on the call graph of the program [53, 46], a pure structural method like program slicing is not completely adequate to isolate code fragments implementing functional abstractions [56]. Program slicing might very often fail to isolate slices which can be associated with a human oriented meaning. Indeed, the most important problem in using program slicing for isolating reuse-candidate code fragments is the identification of a suitable slicing criterion. This task requires the knowledge of the specification of the function we are looking for in the code and cannot be completely automated. In particular, human interaction is required to trace the data of the function into the program variables of the slicing criterion [1]. Moreover, a suitable program point has to be identified. A specification driven method is more suitable than a structural method in order to isolate code fragments implementing functional abstractions.

The need to know the specification of the function to be isolated has been outlined by Canfora *et al.* [41]. In this work a new slicing method, called *conditioned slicing*, has been defined in order to isolate slice fragments that can be executed whenever given conditions in the specification of the function hold true. This approach allows to isolate a particular behaviour of a function. A related work by Hall [92] introduces the *simultaneous dynamic program slicing* to compute a functional subset of an existing program. The approach extends and applies to a set of test cases the *dynamic slicing* [117, 118] which produces executable slices that are correct on only one input. However, in these works the specification of the function to be isolated is not used to identify the slicing criterion.

In this thesis we introduce a *specification driven program slicing* process for identifying a slice which implements a given specification of a functional abstraction. The specification of the functional abstraction is used together with symbolic execution and theorem proving techniques in order to identify a suitable slicing criterion. We assume that the specification of a function is given in terms of its sets of *input* and *output* data and two first order logic formulas called *precondition* and *postcondition* [101]. The precondition expresses the constraint which must hold on the input data to allow the execution of the function, while the postcondition expresses the condition which will hold on the output and input data after the execution of the function (i.e., it relates the output data to the input data). Figure 3.1 shows the specification driven program slicing process. From the source code of the program a *static analyser* produces a control flow based program representation. A *symbolic executor* processes the program representation and associates each statement with the symbolic state holding before its symbolic execution. Such a symbolic state contains the precondition of the statement. The precondition of a statement is also called invariant assertion [82], because it holds before the execution of the statement for each assignment to the symbolic constants. As the problem of finding invariant assertions is *undecidable*, human interaction is required to provide the assertions that cannot be automatically derived. Human interaction is also required to associate the data of the specification with the program variables and in particular to define the set of variables V_{out} corresponding to the output data of the function. These assertions and the specification of the function to be searched for are the input to the *slicing criterion finder.* Once a statement has been annotated with its entry symbolic state, the finder checks the equivalence of the statement precondition with the precondition and postcondition of the specification. A statement whose precondition is equivalent to the input precondition is candidate to be the statement s_{in} of the slicing criterion, while a statement whose precondition is equivalent to the input postcondition is candidate to be the statement s_{out} of the slicing criterion. If s_{in} also dominates s_{out} the slicing criterion $\langle s_{in}, s_{out}, V_{out} \rangle$

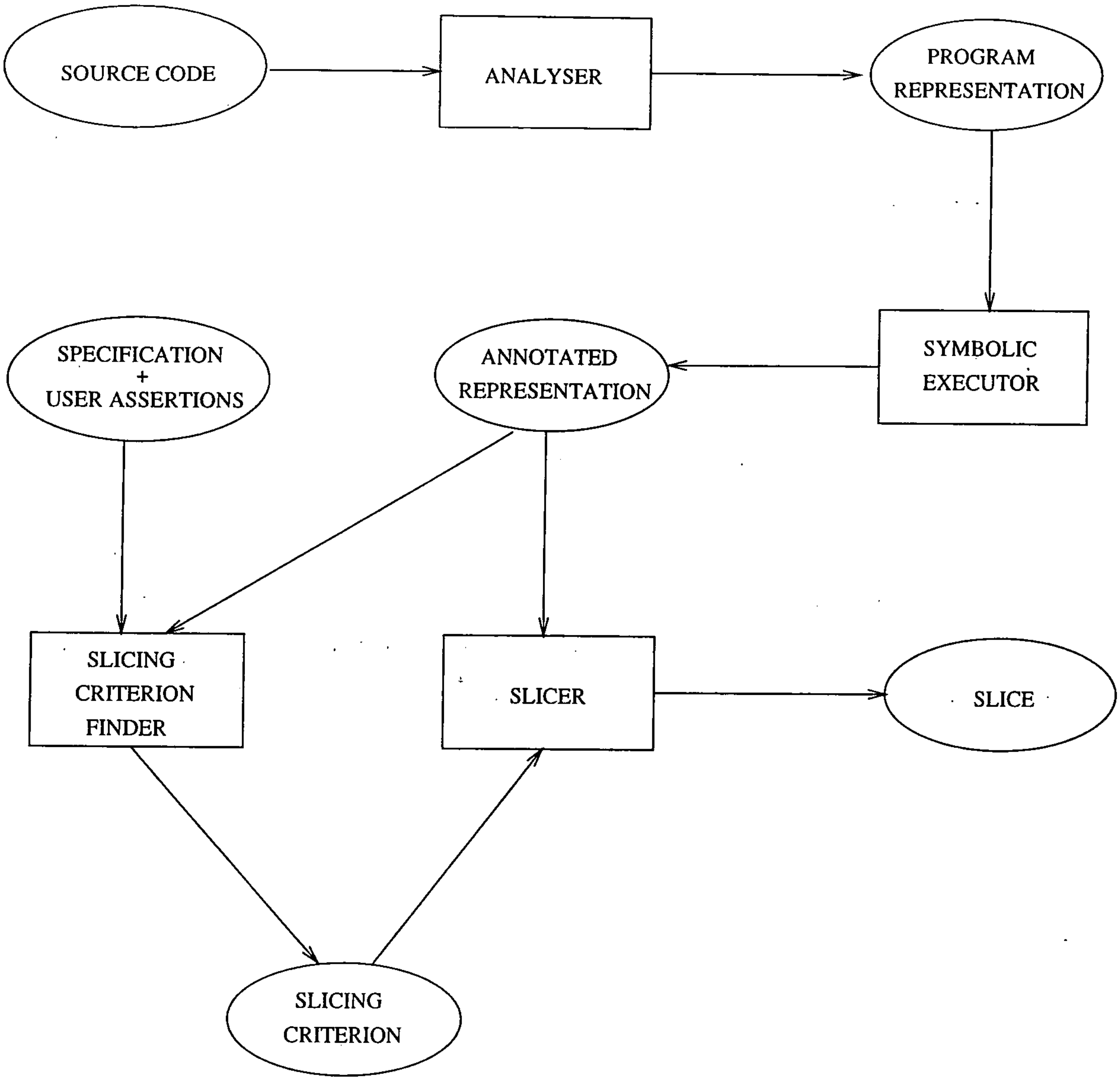


Figure 3.1: The specification driven program slicing process

is produced as output. Finally, a *program slicer* computes the slice and isolates the code implementing the functional abstraction.

3.1.2 Symbolic Execution

Symbolic execution was first introduced as a powerful tool for program testing [60, 111] and program verification [68, 94]. More recently, it has been used for software specialisation [61] and for recovering the formal specifications of reusable software components [2, 57].

The traditional execution model of a program is based on the control flow and on the concept of state. A program state is a set of pairs

$$\{\langle M_1, v_1\rangle, \langle M_2, v_2\rangle, \ldots, \langle M_n, v_n\rangle\}$$

where M_1, M_2, ..., M_n are the memory locations corresponding to the program's variables and for $1 \leq i \leq n$, v_i is the value stored in M_i at some point of the execution. An execution of a program can be represented by a sequence $S_0 p_1 S_1 \ldots p_f S_f$, where each S_i, $0 \leq i \leq f$, is a program state and each p_j, $1 \leq j \leq f$, is a program statement or predicate. S_0 is the initial state and S_i, $1 \leq i \leq f$, is the state resulting from the execution of p_i in the state S_{i-1}; S_f is the final state. If p_j, $1 \leq j \leq f$, is a predicate, the information contained in S_{j-1} unequivocally allows the selection of the branch to follow. Moreover, the resulting state S_j will not change (with respect to the state S_{j-1}) if p_j does not contain side-effects.

While in traditional execution the values of program's variables are constants, in symbolic execution they are represented by symbolic expressions, i.e., expressions containing symbolic constants. For example, the value v of a variable x might be represented by "$2 * \alpha + \beta$", where α and β are symbolic constants. Like a traditional execution, a symbolic execution of a program might be represented by a sequence alternating states and statements. However, if p_j is a predicate, the information contained in the state S_{j-1} may not suffice to select the branch to follow. Indeed, the symbolic boolean expression obtained by replacing the variables in p_j with the corresponding values in S_{j-1} could hold true for some assignments to the symbolic constants and false for other ones. For example, let us consider the following piece of code

```
if(a > b)
    c = a;
else
    c = b;
```

The result of the evaluation of the predicate a > b in the state

$$\{\langle \mathsf{a}, \alpha + \beta\rangle, \langle \mathsf{b}, \alpha - \beta\rangle, \langle \mathsf{c}, \mathit{undef}\rangle\}$$

(where *undef* stands for an undefined value and for the sake of simplicity each memory location is indicated with the name of the corresponding variable) will be $\alpha + \beta > \alpha - \beta$, which can be simplified in $\beta > 0$. As a consequence, it allows the selection of the *true* branch for all the positive values of β and the selection of the *false* branch otherwise.

In the following, the symbols $\wedge$, $\vee$ and $\neg$ will denote the logical *and*, *or* and *not*, respectively.

Multiple Execution Paths

Whatever the selected branch is we must keep track of the condition which caused the branch to be selected, i.e., if $p_j(S_{j-1})$ is the symbolic boolean expression resulting from the evaluation of the predicate p_j in the state S_{j-1}, we must associate with the resulting state S_j the expression $p_j(S_{j-1})$, if the *true* branch has been selected, the expression $\neg p_j(S_{j-1})$, otherwise. Therefore, a symbolic state is a pair $\langle State,\ PC\rangle$, where *State* is as usual a set of pairs of the form $\langle M,\ \alpha\rangle$, M and α being a memory location and a symbolic expression respectively, and *PC* is a first order logic formula called *path-condition* and representing the condition which must be satisfied in order for an execution to follow the particular associated path on the control flow. For example, let us consider the piece of code above. The symbolic execution of the predicate `a > b` in the symbolic state

$$\langle S_1,\ P_1\rangle \ = \ \langle\{\langle \mathsf{a},\ \alpha+\beta\rangle,\ \langle \mathsf{b},\ \alpha-\beta\rangle,\ \langle \mathsf{c},\ \mathit{undef}\rangle\},\ \alpha > 0\rangle$$

generates two possible independent executions depending on the selected branch. The resulting symbolic state will be

$$\langle S_2,\ P_2\rangle \ = \ \langle\{\langle \mathsf{a},\ \alpha+\beta\rangle,\ \langle \mathsf{b},\ \alpha-\beta\rangle,\ \langle \mathsf{c},\ \mathit{undef}\rangle\},\ \alpha > 0 \wedge \beta > 0\rangle$$

whenever the *true* branch is selected,

$$\langle S_3,\ P_3\rangle \ = \ \langle\{\langle \mathsf{a},\ \alpha+\beta\rangle,\ \langle \mathsf{b},\ \alpha-\beta\rangle,\ \langle \mathsf{c},\ \mathit{undef}\rangle\},\ \alpha > 0 \wedge \beta \leq 0\rangle$$

otherwise. On the contrary, if the predicate `a > b` is symbolically executed in the symbolic state

$$\langle\{\langle \mathsf{a},\ \alpha+\beta\rangle,\ \langle \mathsf{b},\ \alpha-\beta\rangle,\ \langle \mathsf{c},\ \mathit{undef}\rangle\},\ \beta > 1\rangle$$

only the *true* branch can be selected and the path-condition of the resulting symbolic state will remain unchanged (because $\beta > 1 \wedge \beta > 0$ is equivalent to $\beta > 1$). Indeed, the path-condition of the symbolic state resulting from the selection of the *false* branch would be $\beta > 1 \wedge \beta \leq 0$ which is identically false.

The previous example shows that when a predicate p is encountered in the symbolic state $\langle S,\ PC\rangle$, at most one of the following two implications can hold true (if *PC* is not identically false):

$$\begin{array}{llcl} \text{(a)} & PC & \Rightarrow & p(S) \\ \text{(b)} & PC & \Rightarrow & \neg\ p(S) \end{array}$$

If exactly one implication holds true, the execution continues on the *true* branch, when the implication (a) holds true, or on the *false* branch, when the implication (b) holds true, and the path-condition *PC* does not change. When neither the implication (a) nor the implication (b) holds true there exists at least one set of inputs to the program satisfying the implication (a) and another set satisfying the implication (b). In this case, both the *true* and the *false* branches must be selected and two symbolic executions will proceed independently. Whenever the *true* branch is selected $p(S)$ must hold true and this information has to be recorded in the path-condition which changes in $PC \wedge p(S)$. Analogously, whenever the *false* branch is selected the path-condition becomes $PC \wedge \neg p(S)$. The evaluation of the implications (a) and (b) can be made by a theorem prover [6, 29, 143]. However, as in general this problem is *undecidable*, human interaction is required to make some decisions whenever the theorem prover is not able to reach a result.

Joining Execution Paths

The presence of predicates in a program and their dependence on the program's input generates more than one symbolic execution of a program [111]. However, two independent symbolic executions could join at a program statement. For example, let us consider again the piece of code above. As already seen, the evaluation of the predicate of the *if* statement in the symbolic state $\langle S_1, P_1 \rangle$ generates two independent executions. Whenever the *true* branch is selected, the statement `c = a` is executed in the symbolic state $\langle S_2, P_2 \rangle$ producing the symbolic state

$$\langle S_4, P_4 \rangle = \langle \{\langle \mathsf{a}, \alpha+\beta \rangle, \langle \mathsf{b}, \alpha-\beta \rangle, \langle \mathsf{c}, \alpha+\beta \rangle\}, \alpha > 0 \wedge \beta > 0 \rangle$$

The selection of the *false* branch will generate the execution of the statement `c = b` in the symbolic state $\langle S_3, P_3 \rangle$ producing the symbolic state

$$\langle S_5, P_5 \rangle = \langle \{\langle \mathsf{a}, \alpha+\beta \rangle, \langle \mathsf{b}, \alpha-\beta \rangle, \langle \mathsf{c}, \alpha-\beta \rangle\}, \alpha > 0 \wedge \beta \leq 0 \rangle$$

At this point, the two symbolic executions could be joined in a single execution by collapsing the two symbolic states. The *state folding* operator $^\circ$ [61] takes as arguments two symbolic states and produces a new symbolic state resulting from the composition of its arguments. Given two symbolic states $\langle State_1, PC_1 \rangle$, and $\langle State_2, PC_2 \rangle$, where

$$\begin{aligned} State_1 &= \{\langle M_1, \alpha_1 \rangle, \ldots, \langle M_n, \alpha_n \rangle\} \\ State_2 &= \{\langle M_1, \beta_1 \rangle, \ldots, \langle M_n, \beta_n \rangle\} \end{aligned}$$

the result of the application of the operator $^\circ$ is the following:

$$\langle State_1, PC_1 \rangle \,^\circ\, \langle State_2, PC_2 \rangle = \langle State_3, PC_3 \rangle$$

where $\langle State_3, PC_3 \rangle$ is defined such that

$$State_3 = \{\langle M_1, \gamma_1 \rangle, \ldots, \langle M_i, \gamma_i \rangle, \ldots, \langle M_n, \gamma_n \rangle\}$$

where $\forall\, i,\ 1 \leq i \leq n$,

$$\begin{aligned}\alpha_i = \beta_i &\Rightarrow \gamma_i = \alpha_i = \beta_i \quad \text{and} \\ \alpha_i \neq \beta_i &\Rightarrow \gamma_i \text{ is a new symbol}\end{aligned}$$

and

$$PC_3 = (PC_1 \wedge \Omega_1) \vee (PC_2 \wedge \Omega_2)$$

where

$$\begin{aligned}\Omega_1 &= \bigwedge_{i \,|\, 1 \leq i \leq n \,\wedge\, \alpha_i \neq \beta_i} (\gamma_i = \alpha_i) \\ \Omega_2 &= \bigwedge_{i \,|\, 1 \leq i \leq n \,\wedge\, \alpha_i \neq \beta_i} (\gamma_i = \beta_i)\end{aligned}$$

If a variable has the same value in both the input symbolic states its value remains unchanged in the resulting state, otherwise it receives a new symbolic value. The path-condition of the resulting state is the disjunction of two parts corresponding to the two input symbolic states, respectively. Each part is the conjunction of the path-condition of the input symbolic state and a predicate binding the new symbolic constants assigned to the memory locations in the new state to the symbolic expressions the memory locations had in the input symbolic state. For example, the result of the composition of the states $\langle S_4,\, P_4\rangle$ and $\langle S_5,\, P_5\rangle$ by the state folding operator will be the state

$$\begin{aligned}\langle S_6,\, P_6\rangle = {} & \{\langle \mathsf{a},\, \alpha+\beta\rangle,\, \langle \mathsf{b},\, \alpha-\beta\rangle,\, \langle \mathsf{c},\, \gamma\rangle\}, \\ & (\alpha > 0 \wedge \beta > 0 \wedge \gamma = \alpha+\beta) \vee (\alpha > 0 \wedge \beta \leq 0 \wedge \gamma = \alpha - \beta)\end{aligned}$$

The memory location c has different values in S_4 and S_5 ($\alpha+\beta$ and $\alpha-\beta$, respectively) and receives a new symbolic value γ in S_6. The path-condition P_6 is the disjunction of two parts. The first part is the conjunction of the path-condition P_4 (i.e., $\alpha > 0 \wedge \beta > 0$) and a predicate binding the value the memory location c has in the states P_6 (i.e., γ) and P_4 (i.e., $\alpha+\beta$). Analogously, the second part is the conjunction of the path-condition P_5 (i.e., $\alpha > 0 \wedge \beta \leq 0$) and a predicate binding the values the memory location c has in the states P_6 (i.e., γ) and P_5 (i.e., $\alpha - \beta$).

Dealing with Loops

Whenever the number of times a loop can be executed is not known symbolic execution fails to keep track of all the paths the loop can generate. This happens when the current path-condition does not imply neither the loop condition nor its negation. To correctly handle this problem we need to find a logical formula that is true at the beginning of the loop (before the evaluation of the loop condition) after an arbitrary number $n \geq 0$ of iterations. Such a formula is called *loop invariant.* The symbolic execution of the invariant in the state holding before the first execution of the loop produces the symbolic state holding before any execution of the loop. To exit the loop and continue the symbolic execution, it is sufficient to assume the loop condition false by the conjunction of its negation with the path-condition.

Although the problem of finding inductive assertions for a given program is *undecidable*, many methods and tools [12, 78, 88, 109, 165, 170, 173] based both on heuristic and deterministic approaches have been proposed in the past to automatically derive invariants (see [165], for a survey). This attempts were related to the problem of proving partial program correctness. Heuristic or top-down methods, e.g. [12, 173], presuppose the existence of a specification (hypothesis) and use the postcondition and the loop exit test to form candidates for invariants. On the other hand, deterministic or bottom-up methods, e.g. [109, 170], do not need input hypotheses and algorithmically extract information from the body and the exit test of the loop. A different approach by Waters [172] tries to analyse loops by decomposing them into smaller code fragments and using a library of plans. This approach allows to deal with more complex loops but has to face with space and time problems to store and search for plans. A variant of this knowledge-based method by Abd-El-Afiz and Basili [3, 4] allows to mechanically annotate a loop with a formal specification and try to cope with search problems by using a hierarchical organisation for the plan library. However, in general the user interaction is required to find and provide an appropriate invariant for a loop in order to continue the symbolic execution of the function.

Module Calls

Another problem in symbolic execution concerns module calls. Two approaches have been proposed in the literature. The *macro-expansion* approach [26, 61] consists in expanding a call statement by symbolically executing the called module. The execution of the module is then repeated each time it is invoked. On the other side, the *lemma* approach [49, 94] symbolically executes a module once, and then uses the results each time the module is invoked.

Considerations similar to the ones made for loops can be made for finding invariants for recursive functions [151]. Like for loop invariants, the problem of finding invariant assertions for recursive functions is *undecidable* and the user interaction is in general necessary to find a solution.

3.2 Specialising Specification Driven Slicing for C Programs

In this section we show how the specification driven program slicing process can be specialised to programs written in the C language [110]. First some problems involving the symbolic execution of C programs are outlined and then we show how both symbolic execution and program slicing can be performed by exploiting an intermediate representation for C programs called Combined C Graph [112, 113, 114].

3.2.1 Problems in Symbolic Execution of C Programs

Two main problems have to be considered during the symbolic execution of C programs. First aliasing can arise in the presence of pointer variables. Second the C language allows expressions containing embedded side-effects and control flows.

Pointer Variables and Aliasing

To deal with alias variables each memory location is associated with its symbolic address and with the type of the stored information. Hence, a memory location M in a symbolic state $\langle State, PC\rangle$ is a pair $\langle \mu, T\rangle$ where μ and T are the symbolic address and the type, respectively, of M[1]. Moreover, given a symbolic address μ, the expression $p(\mu)$ is meant to refer to the memory location having μ as symbolic address. For example, the declaration

```
int x[3], *p;
```

will generate four memory locations[2]:

$$\begin{aligned}
\texttt{x[0]} &= \langle \chi, \text{int}\rangle \\
\texttt{x[1]} &= \langle \chi + 2, \text{int}\rangle \\
\texttt{x[2]} &= \langle \chi + 4, \text{int}\rangle \\
\texttt{p} &= \langle \pi, *\text{int}\rangle
\end{aligned}$$

In general, the symbolic address of an array element $X[i]$ of type T is $\chi + i * sizeof(T)$ where χ is the symbolic address of $X[0]$ and the operator $sizeof$ returns the number of bytes required to store a value of type T. The extension to the case of multidimensional arrays can be easily obtained.

[1] We consider the memory as being an array of bytes. As a consequence, an address is any of the array positions.

[2] We assume that two bytes are required to store an integer value.

Let us suppose to symbolically execute the following program fragment

```
p = x;
*p = *p + 1;
p++;
```

in the symbolic state

$$\langle\{\langle \texttt{x[0]}, \alpha_1\rangle, \langle \texttt{x[1]}, \alpha_2\rangle, \langle \texttt{x[2]}, \alpha_3\rangle, \langle \texttt{p}, \mathit{undef}\rangle\}, \mathit{PC}\rangle$$

The assignment `p = x` gives rise to the aliases `*p` and `x[0]`. Its symbolic execution will result in the assignment of the symbolic address of `x[0]` to the memory location p producing the state

$$\langle\{\langle \texttt{x[0]}, \alpha_1\rangle, \langle \texttt{x[1]}, \alpha_2\rangle, \langle \texttt{x[2]}, \alpha_3\rangle, \langle \texttt{p}, \chi\rangle\}, \mathit{PC}\rangle$$

As the symbolic value of p is χ, the expression `*p` in the statement `*p = *p + 1` will refer to the memory location $p(\chi)$, i.e., the location `x[0]`. The state resulting from the symbolic execution of this statement is

$$\langle\{\langle \texttt{x[0]}, \alpha_1 + 1\rangle, \langle \texttt{x[1]}, \alpha_2\rangle, \langle \texttt{x[2]}, \alpha_3\rangle, \langle \texttt{p}, \chi\rangle\}, \mathit{PC}\rangle$$

Finally, the symbolic execution of the statement (3) will produce the state

$$\langle\{\langle \texttt{x[0]}, \alpha_1 + 1\rangle, \langle \texttt{x[1]}, \alpha_2\rangle, \langle \texttt{x[2]}, \alpha_3\rangle, \langle \texttt{p}, \chi + 2\rangle\}, \mathit{PC}\rangle$$

where the symbolic value of p is obtained as

$$\chi + 1{*}\mathit{sizeof(int)} \quad = \quad \chi + 2$$

In general, if the type and the symbolic value of a memory location M are $*T$ and α[3], respectively, the symbolic value of the expression $M{+}i$ will be $\alpha{+}i{*}\mathit{sizeof}(T)$. If such a value is generated during a symbolic execution it should coincide with the symbolic address of a memory location M' in the current symbolic state. However, symbolic execution in the presence of pointer arithmetics suffers of the same problems as symbolic execution in the presence of arrays. Indeed, the variable i above has the same role of an array subscript and whenever it is input dependent, the address of the involved memory location cannot be decided. A solution to deal with this problem is to consider the disjunction of different symbolic executions for all the feasible values of i in the path-condition. Although the value of i is not input dependent, the value of the expression above could not coincide with any symbolic address, because in C pointers can be used to simulate arrays without fixed length or dynamic structures. In this case a new memory location $\langle\alpha{+}i{*}\mathit{sizeof}(T), T\rangle$ is created and added to the symbolic state.

[3] M is the memory location of a pointer variable and α is the symbolic address of a memory location.

Embedded Side-Effects and Control Flows

Embedded side-effects and control flows also play a critical role in symbolic execution of C programs. Side-effects occur when a variable is defined during the evaluation of an expression. In C side-effects can arise as a result of assignment operators, increment and decrement operators, comma operator and function calls. For example, the variables a, b and c are defined as side-effects of the evaluation of the assignment expression

```
a += b = c++ + f(d)
```

and other variables could be defined during the execution of the function f. Embedded side-effects involve a state change for each variable definition. Hence, while the evaluation order of the operands in a commutative binary operation should not affect the final result, it plays a critical role if the operands may contain embedded side-effects. This is in particular true whenever a C expression also contains embedded control flows that occur with the conditional operator ?: and the boolean operators && and ||. For example, the conditional expression

```
a ? b : c
```

evaluates either b or c depending on the value of a. Moreover, due to the *short-circuiting* evaluation of boolean expressions in C [110], if a is false in the expression

```
if(a && b)
```

the value of the entire expression is false and b will not be evaluated. This leads to changes in the control flow of the program execution and, whenever the expressions contain embedded side-effects, also in the state. For example, let us consider the symbolic execution of the statement

```
if(a > b || a > c++) { ... }
```

in the symbolic state

$$\langle\{\langle \mathtt{a}, \alpha\rangle, \langle \mathtt{b}, \beta\rangle, \langle \mathtt{c}, \gamma\rangle\}, PC\rangle$$

Whenever $PC \Rightarrow \alpha > \beta$ the right-hand side operand of the || operator is not evaluated and the variable c is not incremented, while if $PC \Rightarrow \alpha \leq \beta$ a change in the symbolic state will result because of the increment operator.

3.2.2 The Combined C Graph

Symbolic execution of C programs requires a program representation able to represent embedded side-effects and control flows, in order to take into account embedded state changes. We use the Combined C Graph (CCG), a fine-grained representation for the maintenance of C programs designed by Kinloch and Munro [112, 113, 114] to provide a solution for two problems:

1. combining the features of different program representations into a unique unified intermediate representation for a maintenance environment, on the same line as Harrold and Malloy [98];
2. understanding problems induced by pointers and expressions containing embedded side-effects (resulting from assignment operators, increment and decrement operators, comma operator and function calls) and control flows (due to the *short-circuit* evaluation of the boolean expressions [110]).

The first problem has been solved by: (i) designing a representation for C functions which consists of superimposing several types of intraprocedural edges (enclosing control and data dependences) on a control flow graph and (ii) interconnecting the representations for the different functions by various interprocedural edges. The second problem has been solved by: (i) considering *pointer-induced* aliasing during data flow analysis; (ii) providing explicit representation on the FCCGs for embedded side-effects and control-flows. The use of CCG is also motivated by the fact that its features allow to integrate different tools in a reuse reengineering environment [75].

The CCG is composed of a collection of Function CCGs (FCCGs), each representing an individual function of the C program, connected by various interprocedural edges, like call interface edges and interprocedural data dependencies [103]. Each FCCG is a control flow graph [100] with a variety of superimposed edge types, enclosing control and data dependencies [81]. Rather than a one to one correspondence between the function's statements and its FCCG vertices, a finer-grained representation of these statements is required to deal with embedded side-effects and control flows. Whenever a statement contains embedded side-effects or control flows an additional vertex is created for each sub-expression containing a definition or a possible change in control flow. For example, for an assignment expression

```
a = b;
```

the following vertices are created

- `(a = b)` : no side-effects

- (a), (= b) : side-effect in a
- (b), (a =) : side-effect in b
- (a), (b), (=) : side-effect in a and b

The control flow between these vertices is from left to right. In this way the vertex representing the complete statement is always the last of the sequence. Moreover, two edge types are used to relate these extra vertices, *expression-use edges* and *lvalue-definition edges.* An expression-use edge from vertex p to vertex q, $p \rightarrow_{eu} q$ indicates the evaluation of an expression at vertex p followed by a use of the resulting value at q. This situation occurs when the expression in the right-hand side of an assignment contains a side-effect or an embedded control flow. For example, a side effect in b in the expression above generates the expression-use edge

(b) $\rightarrow_{eu}$ (a =)

An *lvalue* is an expression referring to a named region of storage in the left-hand side of an assignment. An lvalue-definition edge from vertex p to vertex q, $p \rightarrow_{ld} q$ indicates the evaluation of an lvalue at vertex p followed by a writing to the corresponding storage location at vertex q. This situation occurs when the operand in the left-hand side of an assignment contains a side-effect. For example, a side-effect in a in the expression above generates the lvalue-definition edge

(a) $\rightarrow_{ld}$ (= b)

Figure 3.2 shows the CCG subgraph corresponding to the statement

```
*p++ = ++b + a;
```

Each node corresponds to an operator which produces a variable change.

Analogously, a boolean expression with short-circuiting evaluation

- a && b || c

generates the three vertices

- (a), (&& b), (|| c)

and the internal control flow edges

- (a) $\rightarrow_{cf_{true}}$ (&& b)
- (a) $\rightarrow_{cf_{false}}$ (|| c)

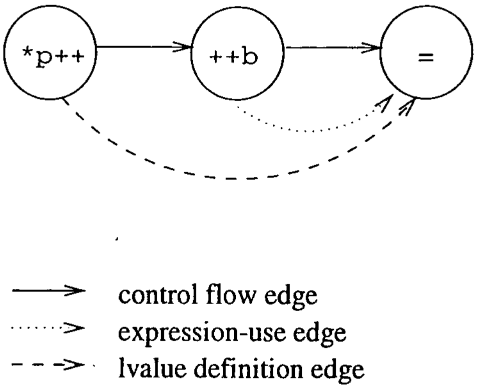


Figure 3.2: Example CCG subgraph

- (&& b) $\rightarrow_{cf_{false}}$ (|| c)

while a conditional expression

- a ? b : c

gives the three vertices

- (a), (b), (c)

and the control flow edges

- (a) $\rightarrow_{cf_{true}}$ (b)
- (a) $\rightarrow_{cf_{false}}$ (c)

An FCCG is completed by control dependencies and intraprocedural data dependencies. The CCG is constructed interconnecting the FCCGs by interprocedural edges such as:

- *call* edges from a call vertex in the calling function to the entry vertex of the called function;
- *parameter binding* edges from each actual parameter vertex to its corresponding formal parameter vertex;
- *return expression-use* edges from a return statement vertex in the called function to the call site in the calling function;
- interprocedural data dependence edges.

A more complete discussion is contained in [114].

The CCG has been extended to allow the representation of the abstract syntax tree [5] of each function in a C program. The abstract syntax tree of a function is important to

```
main() {
  int a, b;
  a = 1 + (b = 0);
  while(a <= 10)
    b += double(&a);
}

int double(int *p) {
      return (*p)++ * 2;
}
```

Figure 3.3: Example C program

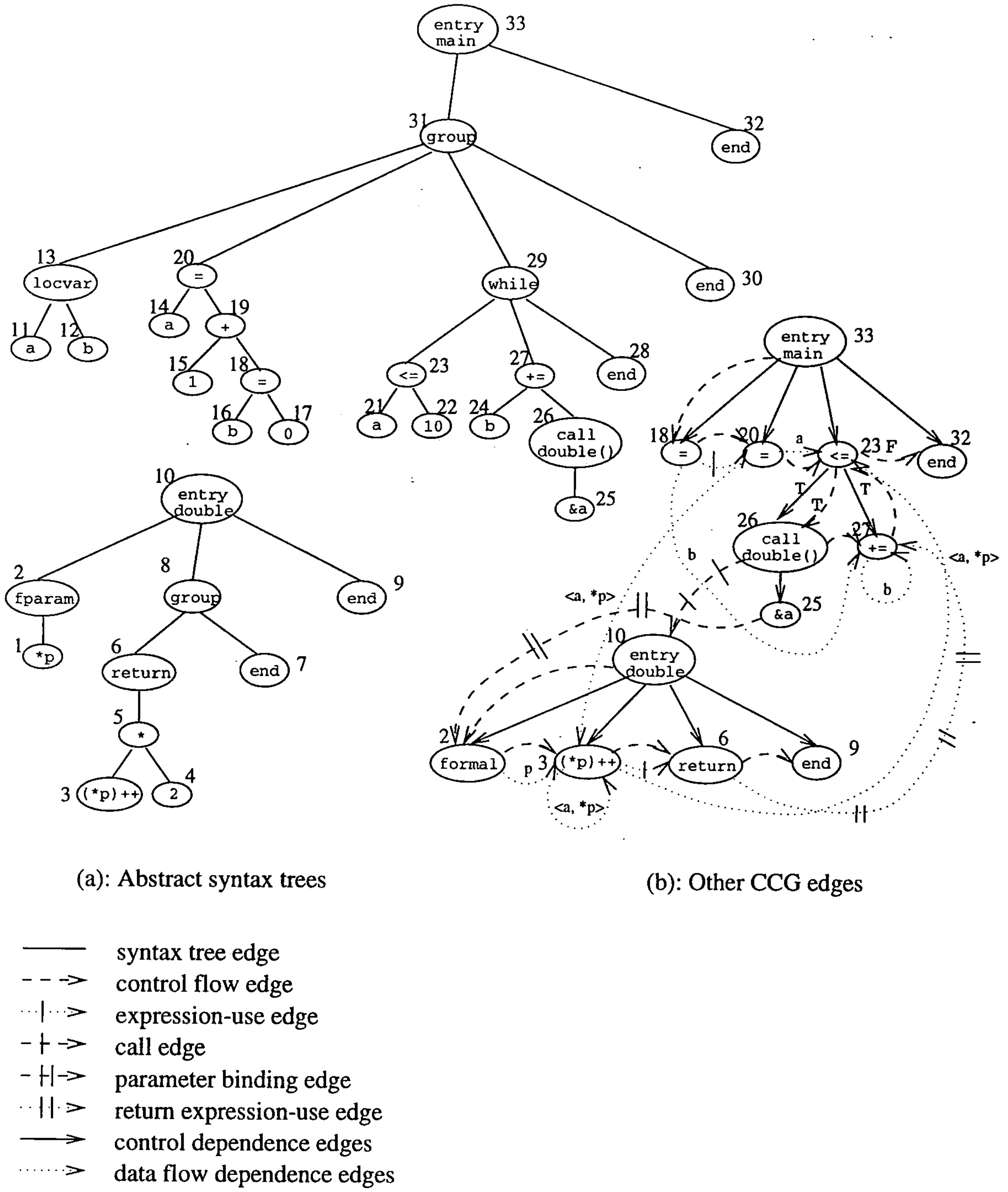
represent the control structures and the expressions contained in a program. The control flow graph and other edges are superimposed to the abstract syntax tree. As an example, let us consider the sample C program in figure 3.3, where the main function contains a call to the function `double`. Figure 3.4 shows the CCG representation of the program above. For sake of simplicity, variable and formal parameter declarations and variable references in the syntax tree are represented by the corresponding identifiers, without semantic information. Figure 3.4.a shows the abstract syntax tree and some of the superimposed edge types. Control and data flow dependence edges are depicted in figure 3.4.b.

3.2.3 Symbolic Execution Using CCG

The symbolic execution of a function is performed on its FCCG representation. A token containing the initial state is placed on the entry node of the FCCG and moves across the control flow edges. In the current implementation of the symbolic executor ambiguous expressions like

```
*p++ = ++b + a-- + b;
```

are not allowed. Indeed, with the exception of the boolean operators `&&` and `||`, the conditional operator `?:` and the comma operator, the order of evaluation for operands within C expressions is undefined [110]. If α and β are the values of the variable `a` and `b` before the execution of the expression above, a left to right evaluation of the right-hand side of the assignment expression gives $\alpha + 2 * \beta + 2$ as result, while a right to left evaluation gives



(a): Abstract syntax trees

(b): Other CCG edges

Figure 3.4: Example CCG

$\alpha + 2 * \beta + 1$. However, compilers impose an evaluation order for operands within expressions and usually they allow ambiguous expressions.

Symbolic Execution in Absence of Ambiguous Expression

If ambiguous expressions are not allowed, the symbolic executor can evaluate an expression containing embedded side-effects or control flows, incrementally, by evaluating the subexpressions containing a variable definition or a change in the control flow whenever the corresponding vertices are encountered on the control flow subgraph. For example, let us consider again the expression `*p++ = ++b + a` whose CCG subgraph is depicted in figure 3.2. Let "int" be the type of a, b and *p and

$$\langle\{\langle \mathsf{a}, \alpha\rangle, \langle \mathsf{b}, \beta\rangle, \langle \mathsf{p}, \chi\rangle, \langle \mathtt{x[0]}, \gamma\rangle, \langle \mathtt{x[1]}, \delta\rangle, \ldots\}, \mathit{PC}\rangle$$

be the symbolic state before the execution of the expression, where χ and $\chi + 2$ are the symbolic addresses of `x[0]` and `x[1]`, respectively. The execution of the expression is obtained by the following steps, each of which corresponds to a CCG node in figure 3.2:

- symbolic execution of the expression *p++: evaluate the value stored in p and increment it; the evaluation result is $\mathit{val}(\text{*p++}) = p(\chi)$[4] which corresponds to the memory location `x[0]`, while the resulting state is:

$$\langle\{\langle \mathsf{a}, \alpha\rangle, \langle \mathsf{b}, \beta\rangle, \langle \mathsf{p}, \chi + 2\rangle, \langle \mathtt{x[0]}, \gamma\rangle, \langle \mathtt{x[1]}, \delta\rangle, \ldots\}, \mathit{PC}\rangle$$

- symbolic execution of the expression b++: increment the value stored in b and evaluate the result; the evaluation result is $\mathit{val}(\text{++b}) = \beta + 1$ and the new state is:

$$\langle\{\langle \mathsf{a}, \alpha\rangle, \langle \mathsf{b}, \beta + 1\rangle, \langle \mathsf{p}, \chi + 2\rangle, \langle \mathtt{x[0]}, \gamma\rangle, \langle \mathtt{x[1]}, \delta\rangle, \ldots\}, \mathit{PC}\rangle$$

- symbolic execution of the expression *val*(*p++) = *val*(++b) + a: evaluate the expression *val*(++b) + a $= \beta + 1 + \alpha$ and assign it to `x[0]`; the resulting state will be

$$\langle\{\langle \mathsf{a}, \alpha\rangle, \langle \mathsf{b}, \beta + 1\rangle, \langle \mathsf{p}, \chi + 2\rangle, \langle \mathtt{x[0]}, \alpha + \beta + 1\rangle, \langle \mathtt{x[1]}, \delta\rangle, \ldots\}, \mathit{PC}\rangle$$

Symbolic Execution of Ambiguous Expression

If ambiguous expressions are allowed we must decide an evaluation order for the expressions. However, in this case we cannot take advantage of the CCG extra-vertices representing embedded side-effects. Indeed, nodes representing embedded side-effects of an expression

[4] The operator *val* returns the value of an expression.

must be skipped until the node representing the whole expression statement is reached (a node representing an embedded side-effect can be recognised because of its outgoing expression-use or lvalue definition edge). Whenever the token reaches a statement node the symbolic evaluation of the corresponding expression starts with respect to the symbolic state associated with the token. The symbolic evaluation is made by traversing the abstract syntax tree of the expression according to the chosen evaluation order. Whenever a subexpression contains a side-effect (it is linked to a vertex in the control flow graph) the symbolic state associated with the token changes. For example, let us assume that the evaluation order of the operands is left to right (i.e., in the binary expression a $\oplus$ b, the operand a is first evaluated, then the operand b is evaluated and finally, the result of the operation is computed). The symbolic execution of the expression

```
*p++ = ++b + a-- + b
```

where the type of a, b and *p is "int", in the symbolic state

$$\langle\{\langle \texttt{a}, \alpha\rangle, \langle \texttt{b}, \beta\rangle, \langle \texttt{p}, \chi\rangle, \langle \texttt{x[0]}, \gamma\rangle, \langle \texttt{x[1]}, \delta\rangle, \ldots\}, PC\rangle$$

where χ and $\chi + 2$ are the symbolic addresses of x[0] and x[1], respectively, is obtained by the following steps:

- evaluate the expression *p referring to the memory location pointed by p and increment the pointer p; the evaluation result is the memory location $val(\texttt{*p++}) = p(\chi) = \texttt{x[0]}$, while the resulting state is

$$\langle\{\langle \texttt{a}, \alpha\rangle, \langle \texttt{b}, \beta\rangle, \langle \texttt{p}, \chi + 2\rangle, \langle \texttt{x[0]}, \gamma\rangle, \langle \texttt{x[1]}, \delta\rangle, \ldots\}, PC\rangle$$

- increment the value stored in b and evaluate the result; the evaluation result is $val(\texttt{++b}) = \beta + 1$ and the new state is

$$\langle\{\langle \texttt{a}, \alpha\rangle, \langle \texttt{b}, \beta + 1\rangle, \langle \texttt{p}, \chi + 2\rangle, \langle \texttt{x[0]}, \gamma\rangle, \langle \texttt{x[1]}, \delta\rangle, \ldots\}, PC\rangle$$

- evaluate the value stored in a and decrement it; the evaluation result will be $val(\texttt{a--}) = \alpha$, while the new state is

$$\langle\{\langle \texttt{a}, \alpha - 1\rangle, \langle \texttt{b}, \beta + 1\rangle, \langle \texttt{p}, \chi + 2\rangle, \langle \texttt{x[0]}, \gamma\rangle, \langle \texttt{x[1]}, \delta\rangle, \ldots\}, PC\rangle$$

- evaluate the symbolic expression

$$val(\texttt{++b}) + val(\texttt{a--}) + \texttt{b} = \beta + 1 + \alpha + \beta + 1 = \alpha + 2 * \beta + 2$$

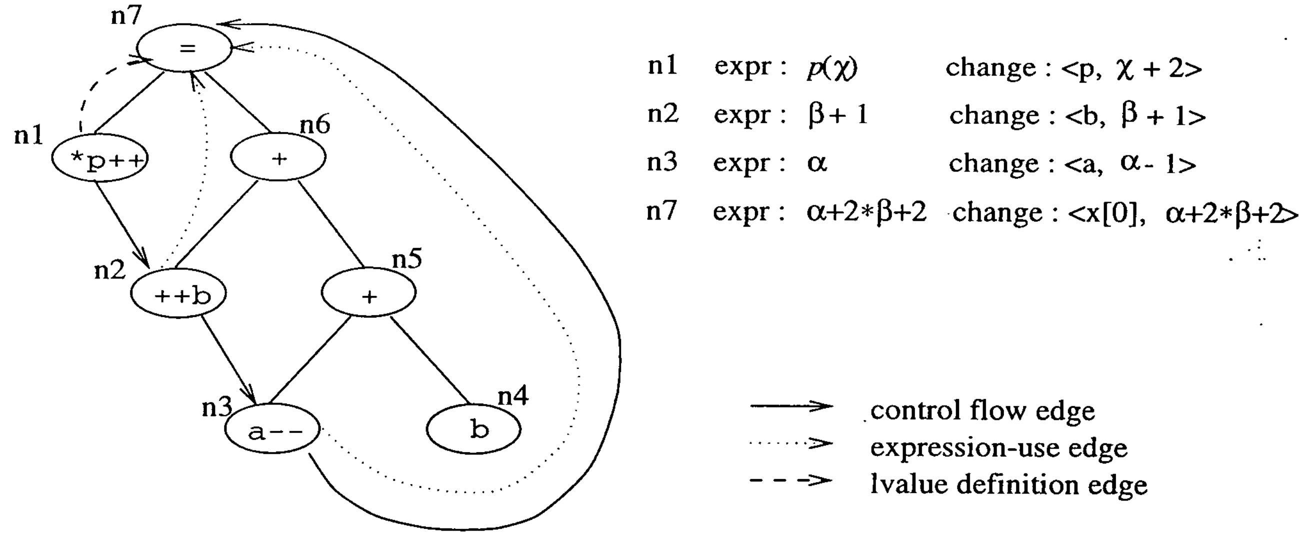

n7
=
n1
*p++
n6
+
n2
++b
n5
+
n3
a--
n4
b
n1 expr : p(χ) change : <p, χ + 2>
n2 expr : β+ 1 change : <b, β + 1>
n3 expr : α change : <a, α- 1>
n7 expr : α+2*β+2 change : <x[0], α+2*β+2>
control flow edge
expression-use edge
lvalue definition edge

Figure 3.5: Example CCG subgraph of ambiguous expression and execution sequence

and assign it to `x[0]`; the resulting state will be

$$\langle\{\langle \mathtt{a}, \alpha - 1\rangle, \langle \mathtt{b}, \beta + 1\rangle, \langle \mathtt{p}, \chi + 2\rangle, \langle \mathtt{x[0]}, \alpha + 2 * \beta + 2\rangle, \langle \mathtt{x[1]}, \delta\rangle, \ldots\}, PC\rangle$$

Figure 3.5 shows the CCG representation and the sequence of evaluations and state changing at each node of the syntax tree. The evaluation is made by a depth-first left traversal of the abstract syntax tree of the expression.

Splitting and Folding Tokens

As seen in the previous section, a symbolic execution can generate two symbolic executions whenever the current path-condition does not imply the truth value of a predicate (a vertex of the CCG with two outgoing control flow edges). In this case the corresponding token is replaced by two new tokens corresponding to the symbolic states produced by the "forking" operation. As a consequence, several tokens can execute concurrently. Moreover, two or more tokens can be folded in a unique token whenever they reach the same vertex. This leads to synchronisation problems among the different tokens which actually advance on the CCG representation like concurrent processes. Figure 3.6 shows the state transition diagram of a token. In particular, one token at a time is *executing*, while tokens which lie on a vertex joining different execution paths are *waiting* and the other tokens are *ready*. The symbolic executor is interactive. The user interaction may be required to schedule the tokens for the advancement. Indeed, as the symbolic executor is just an automatic support for program understanding, the user should choose the execution path to follow. Hence, the user can decide to suspend a token (by making it ready) and execute another one which is in the

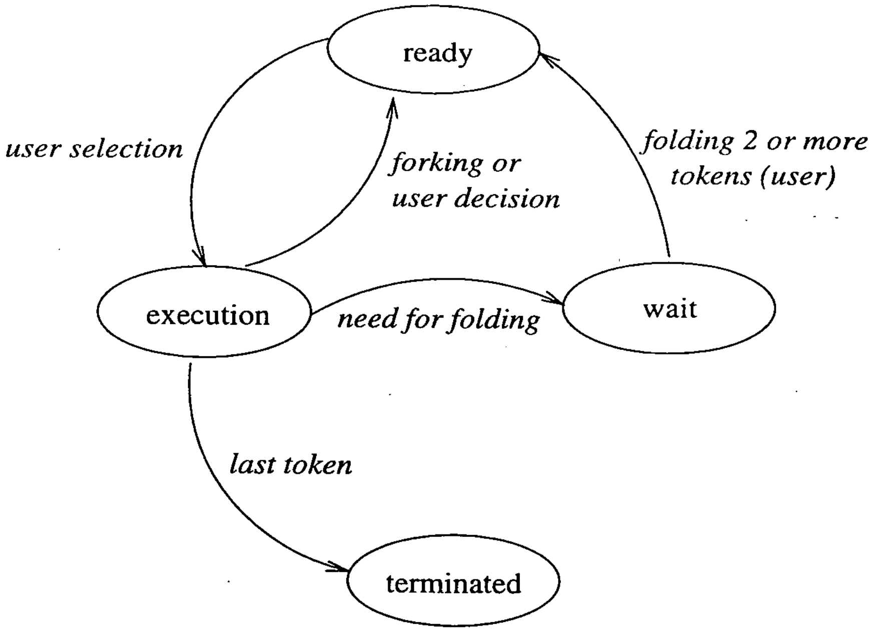


Figure 3.6: State transition diagram for tokens

ready state. Two or more waiting tokens lying on a same vertex, are folded in one token and eventually the user can decide to move them in the ready state. The user interaction is also required to assert an implication and to force an execution to follow only one branch of a predicate, whenever the theorem prover is not able to do it.

Loop Invariants

Loop invariants are recovered using a deterministic approach based on symbolic execution. If a loop is encountered on the execution path, a symbolic execution starts in a fictitious state containing the values of the variables after $n-1$ iterations. The symbolic execution of the loop body produces the state holding after n iterations. From the two states the recurrence equations can be extracted, where the initial conditions are given by the values in the symbolic state holding before the first iteration of the loop. The solution of the recurrence equation gives the loop invariant.

For more complex loops a library of (domain independent) plans can be used to recover invariants [3]. However, the problem of finding the loop invariant is *undecidable* and the user interaction is in general required. If a loop invariant cannot be automatically recovered, and the user is not able to provide it, a solution is to symbolically execute the loop a fixed number n of times, by keeping track of this constraint in the path-condition. The number n is chosen by the user; however, a loop should be symbolically executed a number of times sufficient to understand its behaviour. Where possible, the user can try to generalise the loop's behaviour from the sample iterations. The generalised behaviour can be inductively verified by symbolically executing the loop body for the $n+1^{th}$ time and proving that the behaviour of the n^{th} iteration implies the behaviour of the $n+1^{th}$ iteration [62].

Function Calls

The macro-expansion approach is used for symbolic execution of function calls. Whenever a token reaches a call site on the CCG, the called function is symbolically executed. To keep track of the call site and of the local variables of the calling function a *Stack* is used. The stack is used like the activation stack of traditional execution. Every time a function *f* is called, a quadruple

$$\langle S,\ \textit{local-stack},\ \langle f,\ \delta\rangle,\ rp\rangle$$

is pushed on the stack. S is a sequence of pairs $\langle M,\ \alpha\rangle$, where M is the memory location of a formal parameter of the function and α is its symbolic value; *local-stack* is a stack where the local variables declared at the beginning of a compound statement are allocated; δ is the return value of the function and *rp* indicates the return point to the calling function. A symbolic state for symbolic execution of C programs also contains a set of pairs for global variables and a set for dynamic variables. However, for sake of simplicity only the stack is shown in the examples of the next section.

3.2.4 Slicing C Programs

In this section an algorithm to perform program slicing using the Combined C Graph representation of a program is defined. The algorithm considers slicing criteria of the type $\langle s_{in},\ s_{out},\ V_{out}\rangle$, where s_{in} and s_{out} are vertices of the same FCCG, s_{in} dominates s_{out} on the control flow, and V_{out} is the set of variables referenced at s_{out}. The algorithm specialise to C programs the intraprocedural slicing algorithm presented in [56] and extend it to the interprocedural case.

Previous slicing algorithms computes slices on a slicing criterion of the type $\langle s_{out},\ V_{out}\rangle$. A first iterative algorithm proposed by Weiser [176] computes a slice by backward traversing the control flow of the program starting from the statement of the slicing criterion. The slice is computed as the transitive closure of the direct influence relation. The complexity of the intraprocedural algorithm is $O(ne\log e)$, where n and e are respectively the number of vertices and edges in the control flow graph. Ottenstein and Ottenstein [138] propose an intraprocedural slicing algorithm which exploits the program dependence graph [81]. The slice is computed by backward traversing the control and data dependencies from the statement of the slicing criterion and coincides with the transitive closure of these dependencies.

Therefore, the time complexity is linear in the number of vertices of the program dependence graph. The restriction of this algorithm is that the set of variables V_{out} of the slicing criterion must be referenced at s_{out}.

The interprocedural slicing algorithm proposed by Weiser [176] is imprecise. Horwitz *et al.* [103] show that it fails to account for the calling context of a called procedure. Indeed, a called procedure returns to all call sites and not just the one which generated the call on the specific interprocedural path. The interprocedural slicing algorithm presented in [103] solves this problem by using the system dependence graph. The algorithm extends the intraprocedural algorithm presented in [138] and consists of two phases. A new type of interprocedural transitive dependencies between the actual input and output parameters is exploited to account the calling context: in the first phase the algorithm does not descend into the called procedures, while in the second phase it does not ascend into callee procedures.

Such algorithms do not consider problems arising in C programs in the presence of arrays and pointer variables and do not deal with statements such as `break`, `continue` and `goto` which have effects on the slice. Jiang *et al.* [106] describe these problems and present enhancements to both the intraprocedural and interprocedural Weiser's slicing algorithms [176]. A major problem in static data flow analysis is that the elements of an array or specified by a pointer cannot be distinguished. All the elements are treated as one object. Modification and reference to different elements are considered as references to the whole object [106]. Other problems arise because of *pointer-induced aliasing* [123]. Although this problem is in general *undecidable*, several approximating solutions have been proposed in the literature (see [114] for a survey). Moreover, the occurrence of `break`, `continue` and `goto` statements in a program are not considered properly during slicing because they do not reference any variable [106]. This leads to incorrect slices. Collecting rules are proposed [106] to allow a correct inclusion of such statements in a slice.

While control and data dependencies [81] are sufficient to compute a program slice on a slicing criterion $\langle s_{out}, V_{out} \rangle$, where V_{out} are variables referenced at s_{out}, they do not suffice for computing a slice on the slicing criterion defined above for isolating functional abstractions. Indeed, the program dependence graphs based algorithms [138, 103] calculate a program slices, by backward traversing control and data dependencies, until the entry node of the program is reached. To calculate a slice based on the new definition of slicing criterion, the control flow must be considered in order to restrict the computation to only the CCG vertices lying on a path between s_{in} and s_{out}. To this aim the proposed slicing algorithm consists of three phases, each phase implemented by a specific algorithm. Table 3.1 show the convention used for the CCG edges in the algorithms.

In the first phase (algorithm *ControlPaths* in figure 3.7), the list *Pathlist* of all the nodes

$\rightarrow_{cf_x}$	$x \in \{true, false, uncond\}$	control flow edge
$\rightarrow_{eu}$		expression-use edge
$\rightarrow_{ld}$		lvalue-definition edge
$\rightarrow_{cd_x}$	$x \in \{true, false, uncond\}$	control dependence edge
$\rightarrow_{df}$		data flow dependence edge
$\rightarrow_{pb}$		parameter binding edge
$\rightarrow_{reu}$		return-expression-use edge
$\rightarrow_{cl}$		call edge

Table 3.1: CCG edges reference table

lying on an intraprocedural control flow path between the two vertices of the slicing criterion is detected. The control flow edges are backward traversed from the vertex s_{out}, until the vertex s_{in} is reached. The algorithm constructs the list *Pathlist* incrementally and uses the list *Worklist* storing all vertices already reached and to be included in the list *Pathlist.* At each step a vertex n is selected from the list *Worklist* and added to the list *Pathlist.* All vertices m such that $m \rightarrow_{cf_x} n$ that have not yet been considered (i.e., they have not yet been included in *Pathlist*) are added to the list *Worklist.*

The second phase is interprocedural (algorithm *CallPaths* in figure 3.8) and computes the list `Funlist` of functions that can be reached through a call chain from a call site contained in *Pathlist.* For each call site in *Pathlist* the algorithm computes the initial set of the called functions; then it forward traverses the call edges and add to the list *Funlist* all the functions that can be reached. The algorithm uses a list *Worklist* to store all the functions that have already been reached but not yet considered for inclusion in the list *Funlist.* The algorithm behaves in a similar way to the algorithm *ControlPaths.*

Finally, the slice is computed (algorithm *Slice* in figure 3.9) by backward traversing the CCG dependence edges (*control dependence*, *data flow dependence*, *expression-use*, *lvalue-definition*, *parameter binding*, *return-expression-use*, and *call* edges). The list *Slicelist* is constructed incrementally. At each step a vertex n is selected from the list *Worklist* (containing the vertices to be inserted in the slice) and added to the list *Slicelist.* Moreover, each vertex m linked to n by a CCG dependence edges (and not already in the slice) is added to the list *Worklist*, if either $m \in$ *Pathlist* or m belongs to a function mbox$g \in$ *FunList.* In chapter 4 a Prolog implementation of the algorithm is shown.

Note that at intraprocedural level this algorithm produce more precise slices than other algorithms [138, 176], because of the extra vertices in the CCG representing embedded side-

```
procedure ControlPaths($G$, $s_{in}$, $s_{out}$, Pathlist)
    declare
        $G$: a FCCG;
        $s_{in}$, $s_{out}$: FCCG vertices such that $s_{in}$ dominates $s_{out}$
                on the control flow;
        $n$, $m$: FCCG vertices;
        Pathlist, Worklist: sets of FCCG vertices;
    begin
        Worklist $\leftarrow \{s_{out}\}$;
        Pathlist $\leftarrow \emptyset$;
        while Worklist $\neq \emptyset$ do
            Select and remove a node $n$ from Worklist;
            Pathlist $\leftarrow$ Pathlist $\cup \{n\}$;
            if $n \neq s_{in}$ then
                $\forall m$ such that $m \notin$ Pathlist and $m \rightarrow_{cf_x} n$ with
                  $x \in \{true, false, uncond\}$
                    Worklist $\leftarrow$ Worklist $\cup \{m\}$;
            endif
        endwhile
    end
```

Figure 3.7: Algorithm for computing all nodes lying on control flow paths between the vertices of the slicing criterion

```
procedure CallPaths($G$, Pathlist, FunList)
    declare
        $G$: a CCG;
        $n$, $m$: CCG vertices;
        $f$, $g$: functions;
        Pathlist: set of FCCG vertices such as computed
                  by the procedure ControlPaths;
        Worklist, FunList: sets of functions;
    begin
        Worklist $\leftarrow \emptyset$;
        $\forall\ n \in$ Pathlist such that $n \rightarrow_{cl} m$ where
          $m$ is the entry FCCG node of the function $f$
            Worklist $\leftarrow$ Worklist $\cup \{f\}$;
        Funlist $\leftarrow \emptyset$;
        while Worklist $\neq \emptyset$ do
            Select and remove a function $f$ from Worklist;
            Funlist $\leftarrow$ Funlist $\cup \{f\}$;
            $\forall g$ such that $g \notin$ Funlist and $n \rightarrow_{cl} m$ where $n$ is a FCCG
              call vertex of $f$ and $m$ is the FCCG entry vertex of $g$
                Worklist $\leftarrow$ Worklist $\cup \{g\}$;
        endwhile
    end
```

Figure 3.8: Algorithm for computing all functions reachable from call sites lying on control flow paths between the vertices of the slicing criterion

```
procedure $Slice(G,\ f,\ s_{in},\ s_{out},\ Slicelist)$
    declare
        $G$: a CCG;
        $f$, $g$: functions;
        $s_{in}$, $s_{out}$: FCCG vertices of the function $f$ such that $s_{in}$ dominates $s_{out}$
                on the control flow;
        $n$, $m$: FCCG vertices;
        $Pathlist$, $Worklist$, $Slicelist$: sets of CCG vertices;
        $FunList$: set of functions;
    begin
        $ControlPaths(G,\ s_{in},\ s_{out},\ Pathlist)$;
        $CallPaths(G,\ Pathlist,\ FunList)$;
        $Worklist \leftarrow \{s_{out}\}$;
        $Slicelist \leftarrow \emptyset$;
        while $Worklist \neq \emptyset$ do
            Select and remove a node $n$ from $Worklist$;
            $Slicelist \leftarrow Slicelist \cup \{n\}$
            $\forall m$ such that $m \notin Slicelist$ and
             (($m$ belongs to the FCCG of the function $f$ and $m \in Pathlist$) or
              ($m$ belongs to the FCCG of the function $g$
               with $g \neq f$ and $g \in Funlist$)) and
                ($m \rightarrow_x n$ with $x \in \{eu, ld, df, pb, reu, cl\}$ or $m \rightarrow_{cd_x} n$
                 with $x \in \{true, false, uncond\}$)
                $Worklist \leftarrow Worklist \cup \{m\}$;
        endwhile
    end
```

Figure 3.9: Algorithm for computing a program slice

effects and control flow [113]. Indeed, each vertex can define at most one variable. The control and data dependencies are used as usual [138]. Moreover, expression-use and lvalue-definition edges due to embedded side-effects and control flows, ensure that a slice reaching a refined vertex also includes vertices affecting the value of the expression at the refined vertex. At interprocedural level the data flow dependencies, the parameter binding edges, the return expression-use edges and the call edges are considered. Due to the use of the the list *Funlist* the algorithm only considers functions that can be reached from a call site in the list *Pathlist*. In this way the slice does not include functions that do not belong to an interprocedural path between the two nodes of the slicing criterion. However, inaccurate overly-conservative slices can result due to the lack of calling context information. Unlike Horwitz *et al.* algorithm [103], the algorithm proposed in this thesis uses direct instead of transitive interprocedural data flow dependencies. This prevents the use of a two phase algorithm. However this choice is more appropriate in the presence of aliasing due to pointer variables [114]. In chapter 5 the candidature criterion is evaluated and a discussion about advantages and limitations of the slicing algorithm is given, together with suggestions for further improvements.

3.3 Summary

In this chapter a new specification driven candidature criterion has been presented. The candidature criterion is based on the theoretical framework of program slicing [176], a program decomposition technique. We use a new definition of program slice which contains two statements in the slicing criterion. The two statements, delimit the region of code in which the slice must be computed. In this way we can obtain more precise slices with respect to the functional abstraction to be recovered. The slice extracted can be easily reengineered and clustered into a module.

The specification of the function to be isolated is used together with symbolic execution and theorem proving techniques to correctly identify the initial and final statements of the slicing criterion. Symbolic execution allows the association of a program statement or predicate with its precondition, i.e. the condition which must hold on the program variables before its execution. The specification of the functional abstraction, given in terms of a precondition and a postcondition, is then compared with the conditions associated with program statements, also called invariant assertions. The statement whose precondition is equivalent to the precondition of the functional abstraction is candidate to be the initial statement of the slicing criterion, while the statement whose precondition is equivalent to the postcondition of the functional abstraction is candidate to be the final statement of the slicing criterion. The slicing criterion is produced whenever the initial statement dominates the final statement on the control flow. Human interaction is required during this task. First, the software engineer must associate the output data of the specification with the program's variables. Moreover, as the problem of finding invariant assertions is in general *undecidable*, symbolic execution can require human interaction in order to prove some implications and assert some invariants.

The specification driven program slicing process is language independent. However, the candidature criterion has been specialised for programs written in C language. A fine-grained representation for C programs, the Combined C Graph (CCG) [114], is used to perform both symbolic execution and program slicing. The CCG contains the features of several different program representation and can be used for most of the software maintenance tasks allowing a better integration of different software tools. Some problems arising in symbolic execution of C programs, such us pointer variables and embedded side-effects and control flows have also been outlined.

Chapter 4

Implementation

This chapter describes a prototype implementation of a tool for isolating reusable functions written in C language [110] using the specification driven program slicing technique. The prototype is intended to be used for the evaluation of the case studies and not for commercial reasons. Therefore, issues like time/space performance and user-friendliness have not been taken into account.

The language chosen for implementing the prototype tool is Prolog [164]. The choice of the Prolog environment has been made for several reasons. First, the Prolog language allows the easy implementation of logic theories needed for proving implications between predicates. Second, tool integration in a reuse reengineering environment can be easily achieved through sharing the common data base of Prolog facts [75]. Moreover, a prototype tool can be extended and enhanced by simply adding new rules. Finally, the Prolog runtime system provides a query language that can be exploited for user interaction.

The architecture of the prototype system is shown in figure 4.1. The system is composed of four main subsystems: the *CCG Analyser* (CCGA), the *Slicing Criterion Finder* (SCF), the *Slicer* and the *Graphical Display Tool* (GDT). The source C code is translated into Prolog facts forming the *CCG fact base* by the CCGA. The CCG fact base and the *function specification* are the input for the module SCF embedding a symbolic executor and a theorem prover for the production of the *slicing criterion*. The Slicer takes as input the CCG fact base and the slicing criterion and produces the *slice* implementing the function specification. Finally, the graphical display tool allows the visualisation of the slice. The following subsections show the architecture of the system modules.

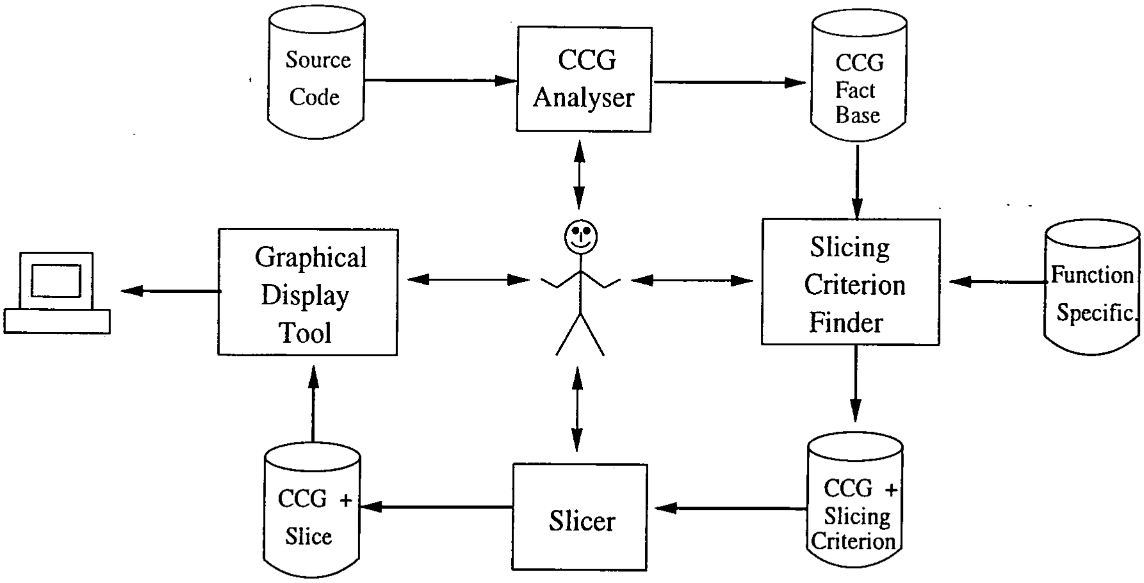


Figure 4.1: The prototype system architecture

4.1 The CCG Analyser

The first module of the prototype system is the analyser CCGA whose architecture is shown in figure 4.2. The construction of the CCG fact base consists of two stages. In the first stage, *syntax directed translation* techniques [5] are used to produce a *partial CCG fact base* from the C source files. This task is accomplished by a static analyser written using the YACC [107] compiler-compiler. The YACC analyser is an enhanced version of the CCG translator *ccg_trans* described by Kinloch [114]. It is based on the C analysis tool PERPLEX [30] and uses a grammar corresponding to that in [110]. PERPLEX produces a generic control-flow based program representation in the form of Prolog facts to allow the easy development of software engineering tools. Problems like embedded side-effects and control flows are not considered. The CCG translator *ccg_trans* modifies PERPLEX by taking in account these problems and providing an explicit representation for them. It produces Prolog facts for representing the vertices of the CCG, the *expuse* and *lvaldef* dependencies, variable, function and type declarations of the source program and the control flow within each C function.

In order to allow symbolic execution using the CCG fact base, the CCG YACC tool has been enhanced with the production of facts for representing the abstract syntax tree (vertices and edges) of each C function, the scope of each compound statement (useful for identifying the scope of local variables) and semantic edges between the syntax tree vertices and the control flow graph vertices.

The second stage of the CCG construction is to augment the partial CCG fact base

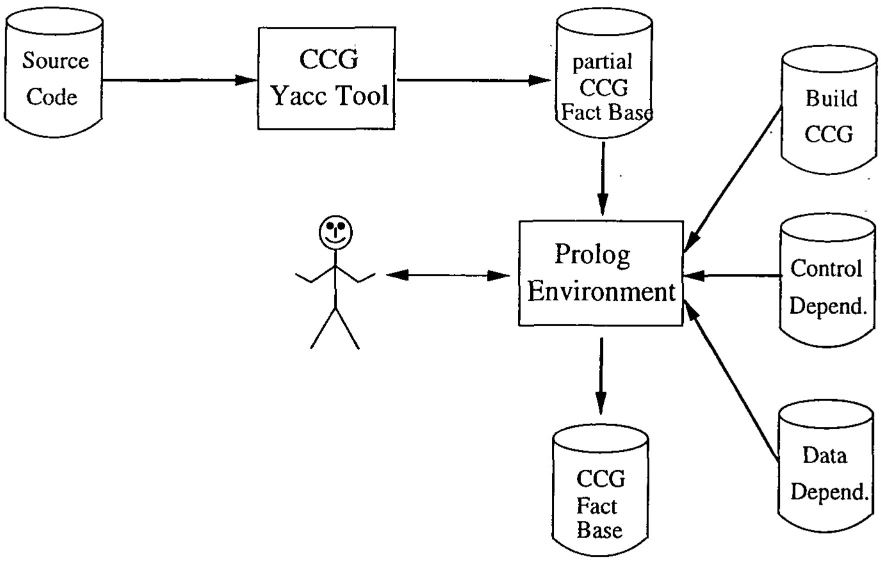


Figure 4.2: The CCG analyser architecture

to give the complete CCG representation, the *CCG fact base* [114]. This task is achieved by loading in the Prolog environment three Prolog metaprograms together with the partial CCG fact base. A brief description of the three metaprograms follows.

build_ccc This program implements algorithms for interconnecting the different FCCGs by various interprocedural edges, like call interface edges (enclosing binding edges between actual and formal parameters and between return statements and calling sites). Moreover, the original version [114] has been modified by adding rules for producing facts linking each identifier in the syntax tree with its declaration.

control_dep This program implements algorithms for computing control dependencies [81]: post-dominator sets and relations are easily computed by making extensive use of the control flow graph.

data_dep This program implements algorithms for computing data dependencies in presence of pointer variables, structures and dynamic analysis. The method employed [114] is based on that described by Horwitz *et al.* [102] which addresses the *reaching definitions* problem in terms of memory locations, rather than variable names and aliases.

The metaprograms above are executed in the order in the Prolog environment and modify the CCG fact base.

A full description of the architecture of the CCG fact base is given in appendix A. A complete CCG fact base for the sample C program of figure 3.3, whose graphical representation is depicted in figure 3.4 is contained in appendix B. Further implementation details of the CCG analyser can be found in [114].

The CCG analyser has been implemented following the logic based paradigm for reverse engineering tool production described by Canfora *et al.* in [35]. Actually, the partial CCG fact base contains a set of *direct relations* that are obtained from the source code by static analysis, while the Prolog metaprograms allow the abstraction of a set of *summary relations* that complete the CCG fact base.

4.2 The Symbolic Executor and Slicing Criterion Finder

The symbolic executor and slicing criterion finder is the core of the system and is used to find a slicing criterion from a CCG representation of a program and the specification of a functional abstraction. This subsystem is composed of four Prolog modules, as depicted in figure 4.3 which are loaded into the Prolog environment together with the CCG fact base and the specification of the functional abstraction given in term of a precondition and a postcondition and binding between the the data of the function and the program's variables. The logic formulas of the function specification are universally quantified as well as the path-condition in a symbolic state. A logic formula is expressed as a Prolog term corresponding to the syntax tree of the expression. As an example, the formula:

$$(a - b > 0 \land c = a - b) \lor (b - a \geq 0 \land c = b - a)$$

is expressed by the Prolog term:

```
expr(or,
  expr(and,
    expr(gt, expr(minus, a, b), 0), expr(eq, c, expr(minus, a, b))),
  expr(and,
    expr(geq, expr(minus, b, a), 0), expr(eq, c, expr(minus, b, a))))
```

Figure 4.4 shows the initial phase of the execution of the prototype tool. The four modules of the symbolic executor and slicing criterion finder are loaded together with some CCG Prolog libraries by consulting the program `symb_exec.pl`. The symbolic execution and the search for the slicing criterion start with the query `start_exec`. Once the CCG file, the specification file, the starting function and the initial path-condition have been entered, the first token is created and the control passes to the scheduler for the selection of an operation. The format of a token is:

```
token(Tok_id, CCG_Vertex, Tok_state, State, PC, Call_stack)
```

where:

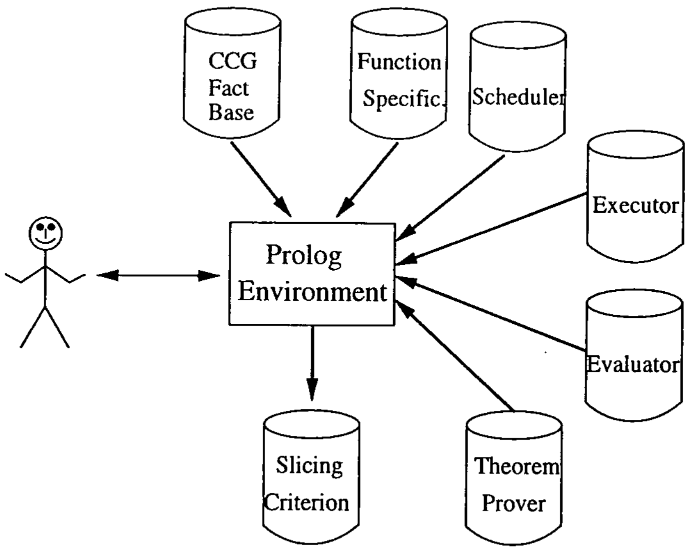


Figure 4.3: The symbolic executor and slicing criterion finder architecture

`Tok_id` is the token identifier;

`CCG_Vertex` is the CCG control flow graph vertex;

`Tok_state` is the state of the token (execution, ready, wait, terminated);

`State` is the program state, defined as:

`state(Local, Heap, Global)`

where `Local`, `Heap` and `Global` are `Heap` and `Global` are lists corresponding to dynamic and global variables, respectively, and `Local` is the stack for the local variables. Each element contained in `Local` has the form:

`local(Form_par, Local_stack, Ret_point, Ret_value)`

where `Form_par` is the list of formal parameters, `Local_stack` is the stack for the local variables of a function, `Ret_point` is the return point of the function on the CCG and `Ret_value` is the return value;

`PC` is the path-condition;

`Call_stack` is a stack containing all the call sites still to which the execution has not returned yet on the interprocedural control flow path. This is information is necessary to avoid folding two tokens executing a function from different call-site chains.

In the following a description of the four modules of the subsystem follows.

```
altair(sun4):dcs3adl[38]: prolog
Welcome to SWI-Prolog (Version 1.8.6 December 1993)
Copyright (c) 1993, University of Amsterdam.  All rights reserved.

1 ?- consult('symb_exec.pl').
ccg_lib.pl compiled, 0.07 sec, 12,052 bytes.
ccgquery.pl compiled, 0.10 sec, 16,652 bytes.
scheduler.pl compiled, 0.08 sec, 13,228 bytes.
executor.pl compiled, 0.12 sec, 19,580 bytes.
evaluator.pl compiled, 0.10 sec, 16,968 bytes.
theorem_prover.pl compiled, 0.03 sec, 4,324 bytes.
symb_exec.pl compiled, 0.51 sec, 63,564 bytes.

Yes
2 ?- start_exec.

CCG File Name |: fact_and_sum.pl.
fact_and_sum.pl compiled, 0.12 sec, 18,312 bytes.

Specification File Name |: factorial.pl
factorial.pl compiled, 0.03 sec, 1,084 bytes.

Starting Function |: main.

Initial path-condition |: true.


 ... Token state lists ...

Execution
token(0, cf_node(1, main, 0), [])

Ready

Wait


 ... Operations ...

1. change token state
2. execute token
3. fold tokens
4. terminate

Select Operation |: 2.
```

Figure 4.4: Unix script for symbolic executor

scheduler This module is invoked whenever an operation has to be performed. The user is asked for the operation (see figure 4.4) and its feasibility is checked. The user is also asked about the token (or the tokens) on which the operation must be performed. The operation to change the state of a token and to fold two tokens are also contained in this module.

executor This module is invoked whenever a token has been selected to advance its execution. Different cases are considered for the execution of loops, predicates, function calls and function returns. The token moves on the CCG accordingly.

evaluator This module is invoked whenever an expression must be evaluated. The expression is evaluated by traversing the CCG abstract syntax tree and producing an expression in the same format described above for a logic formula. The expression is simplified when possible and linked to the root of the corresponding CCG abstract syntax subtree. This is useful when evaluating side-effected sub-expressions of an expression.

theorem_prover This module is invoked whenever an implication must be proved. In particular, a proof is required each time a predicate is encountered during symbolic execution. Moreover, after each token execution, the user can invoke the module to prove the implication between the current path condition and the specification of the function, in order to find a slicing criterion.

Whenever a slicing criterion has been found (or the symbolic execution terminates unsuccessfully) the execution of the subsystem terminates. The slicing criterion is produced in the form:

```
slicing_criterion(Fid, FName, NodeFrom, NodeTo)
```

where `Fid` is the file identifier, `FName` is the function name, `NodeFrom` and `NodeTo` are the CCG vertices corresponding to the statements s_{in} and s_{out}, respectively.

4.3 The Program Slicer

The program slicer is a Prolog program which takes the slicing criterion produced by the module SCF as input and produce the program slice (see figure 4.5). The appendix C contains the listing of the program. The slicing algorithm implemented considers slicing criteria of the type $\langle s_{in}, s_{out}, V_{out}\rangle$ as defined in chapter 3, where V_{out} is the set of variables referenced at s_{out}. The slicing criterion is given as input to the procedure[1]:

[1] A Prolog procedure is a set of clauses with the same predicate in the head [164].

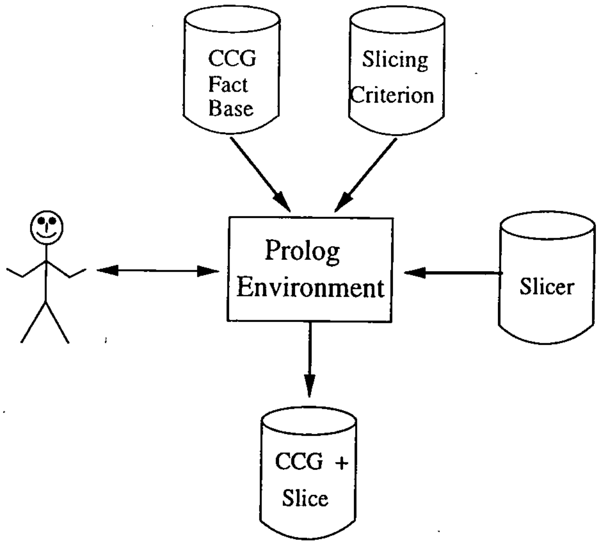


Figure 4.5: The slicer architecture

```
slice(Fid, FName, NodeFrom, NodeTo)
```

where Fid and Fname are the *file identifier* and the *function name* containing the statements s_{in} and s_{out}, while NodeFrom and NodeTo are the CCG control flow nodes corresponding to the two statements, respectively.

The algorithm implemented works both at intraprocedural and interprocedural level. At intraprocedural level, it only considers in the slice the set Pathlist of CCG nodes lying on a control flow path between the statements s_{in} and s_{out} of the slicing criterion [56]. At interprocedural level, the algorithm only considers the set Funlist of functions that can be reached through a call chain from a call site contained in Pathlist. In this way only interprocedural control flow paths between s_{in} and s_{out} can be considered. The sets Pathlist and Funlist are computed by the procedures find_path and find_call_chain, respectively. The slice is computed by backwards traversing CCG dependence edges (procedure traverse). At each step a CCG control flow node n is considered and added to the current slice. Moreover, all nodes m linked to n by one of the following edges:

- *control dependence* edge;
- *data flow dependence* edge;
- *expression-use* edge;
- *lvalue-definition* edge;
- *parameter binding* edge;

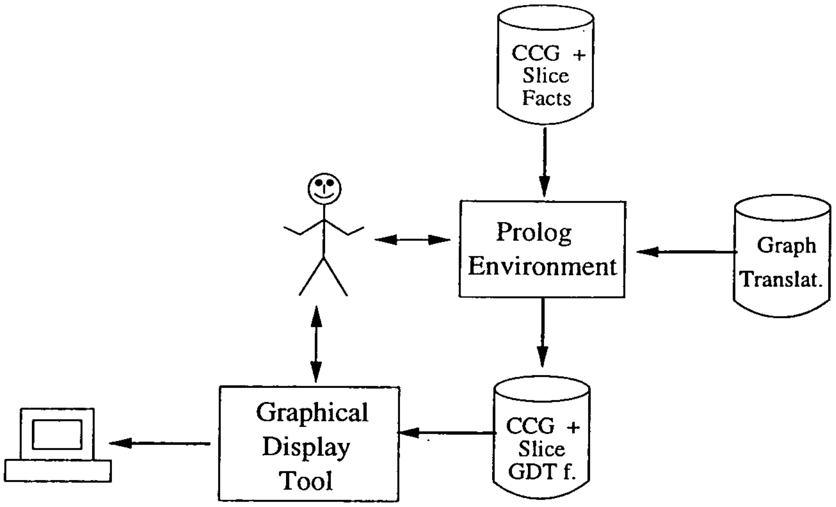


Figure 4.6: The graphical display tool architecture

- *return-expression-use* edge;
- *call* edge

are added to the list of the nodes to be still considered for inclusion in the slice (procedure backwards_trav). The control dependencies, the data flow dependencies, the parameter binding edges and the call edges are considered as usual [103] to affect the value of the variable defined at a CCG vertex. Moreover, as an expression containing embedded side-effects and control flows is represented in the CCG as a set of refined vertices, expression-use, lvalue-definition and return-expression edges ensure that a slice reaching a refined vertex also includes vertices contributing values to the expression at the refined vertex.

4.4 The Graphical Display Tool

The CCG Graphical Display system consists of two modules [114], as shown in figure 4.6. The first module is a Prolog metaprogram for *tool bridge technology.* This metaprogram is loaded in the Prolog environment together with the CCG and slice fact base and its execution produces an alternative representation format (*GDT f.*) of the data base. The new database is the input for the Graphical Display Tool developed by Bodhuin [20] which allow the visualisation of program slices and other program views.

4.5 Summary

This chapter has described the prototype implementation of a system for identifying specification driven program slices. The system uses the CCG analyser for the production of the CCG fact base and a Graphical Display Tool for the visualisation of a slice. The core of the system consists of a symbolic executor and a theorem prover for the selection of a slicing criterion, according to an input specification. Finally, a slicer produces the slice implementing the functional abstraction. The system has been integrated in Prolog environment. A static analyser written using the YACC compiler-compiler and embedded in the CCG analyser produces a partial CCG fact base from the C source code. The Graphical Display Tool is written in C and takes as input an alternative representation format of the CCG fact base produced by a Prolog metaprogram.

Chapter 5

Evaluation

In this chapter the specification driven program slicing method is evaluated. In the first section of this chapter we will demonstrate the specification driven slicing process through an example conducted on a simple C program. In the second section we present a summary of a case study in identifying, extracting and reengineering code fragments implementing functional abstractions in an existing C software system, after it has undergone two meaningful maintenance interventions [58]. Finally, the method and the results obtained are evaluated and some conclusions are outlined in the last section.

5.1 A Simple Example

The precondition of a statement is a first order logic formula which must hold true in order for the statement to be executed. The precondition of a statement can be derived from the symbolic state holding before its execution. Each vertex in the control flow subgraph of a program's CCG is therefore annotated with the symbolic state holding before its execution. During symbolic execution the finder first looks for a statement s_{in} whose precondition is equivalent to the precondition of the functional abstraction. Once such a statement has been found, it looks for a statement s_{out} whose precondition is equivalent to the postcondition of the functional abstraction. If also the statement s_{out} is found and s_{in} dominates s_{out}, the slicing criterion $\langle s_{in}, s_{out}, V_{out} \rangle$ is produced, where V_{out} is the set of the program variables corresponding to the output data of the function. Finally, the slice is computed.

Let us consider the sample program in figure 5.1 which computes the factorial `fact` and the sum `sum` of the absolute difference of two integers `a` and `b`. Let us suppose to extract the slice implementing the factorial function, whose specification is:

factorial: $n \in \mathrm{N}_0 \rightarrow m \in \mathrm{N}$
precondition: *true*

```
main()
{
   int a, b, diff, fact, sum, i;
   scanf("%d", &a);
   scanf("%d", &b);
   if (a > b)
      diff = a - b;
   else
      diff = b - a;
   fact = i = 1;
   sum = 0;
   while (i <= diff) {

      fact *= i;
      sum += i++;
   }
   printf("%d \n", fact);
   printf("%d \n", sum);
}
```

Figure 5.1: A sample C program

postcondition: $m = n!$

where $\mathbf{N}$ is the set of the natural numbers and $\mathbf{N}_0$ is the set of natural numbers enclosing 0. The software engineer must provide some assertions to associate the data of the specification with the variables of the program. In this case he will associate the variable `diff` with the input data n and the variable `fact` with the output data m. Moreover, as `diff` and `fact` are integers variables while n and m are natural numbers, the specification of the function must be changed according to the program variables in:

factorial: $n \in \mathbf{Z} \rightarrow m \in \mathbf{Z}$
precondition: $n \geq 0$
postcondition: $m = n!$

where $\mathbf{Z}$ is the set of relative numbers. Let us symbolically execute the program above. The symbolic execution will start in the symbolic state[1]:

$$\langle S_1,\, P_1\rangle \;=\; \langle\{\ \},\, \mathit{true}\rangle$$

After the allocation of the local variables and the execution of the two input statements, the symbolic state becomes:

$$\langle S_2,\, P_2\rangle \;=\; \langle\{\langle \texttt{a},\, \alpha\rangle,\, \langle \texttt{b},\, \beta\rangle,\, \langle \texttt{diff},\, \mathit{undef}\rangle,\, \langle \texttt{fact},\, \mathit{undef}\rangle,\, \langle \texttt{sum},\, \mathit{undef}\rangle,\, \langle \texttt{i},\, \mathit{undef}\rangle\},\, \mathit{true}\rangle$$

At this point the `if` statement is encountered. As the path-condition does not implies the `if` predicate nor its negation, the two symbolic states are generated on the *true* and *false* branches, respectively:

$$\begin{aligned}
\langle S_3,\, P_3\rangle &= \langle\{\langle \texttt{a},\, \alpha\rangle,\, \langle \texttt{b},\, \beta\rangle,\, \langle \texttt{diff},\, \mathit{undef}\rangle,\, \langle \texttt{fact},\, \mathit{undef}\rangle,\, \langle \texttt{sum},\, \mathit{undef}\rangle,\, \langle \texttt{i},\, \mathit{undef}\rangle\},\\
&\quad \alpha-\beta>0\rangle\\
\langle S_4,\, P_4\rangle &= \langle\{\langle \texttt{a},\, \alpha\rangle,\, \langle \texttt{b},\, \beta\rangle,\, \langle \texttt{diff},\, \mathit{undef}\rangle,\, \langle \texttt{fact},\, \mathit{undef}\rangle,\, \langle \texttt{sum},\, \mathit{undef}\rangle,\, \langle \texttt{i},\, \mathit{undef}\rangle\},\\
&\quad \beta-\alpha\geq 0\rangle
\end{aligned}$$

The symbolic execution of the statement `diff = a - b` in the state $\langle S_3,\, P_3\rangle$ produces the symbolic state:

$$\begin{aligned}
\langle S_5,\, P_5\rangle &= \langle\{\langle \texttt{a},\, \alpha\rangle,\, \langle \texttt{b},\, \beta\rangle,\, \langle \texttt{diff},\, \alpha-\beta\rangle,\, \langle \texttt{fact},\, \mathit{undef}\rangle,\, \langle \texttt{sum},\, \mathit{undef}\rangle,\, \langle \texttt{i},\, \mathit{undef}\rangle\},\\
&\quad \alpha-\beta>0\rangle
\end{aligned}$$

while the symbolic execution of the statement `diff = b - a` in the state $\langle S_4,\, P_4\rangle$ produces the symbolic state:

$$\begin{aligned}
\langle S_6,\, P_6\rangle &= \langle\{\langle \texttt{a},\, \alpha\rangle,\, \langle \texttt{b},\, \beta\rangle,\, \langle \texttt{diff},\, \beta-\alpha\rangle,\, \langle \texttt{fact},\, \mathit{undef}\rangle,\, \langle \texttt{sum},\, \mathit{undef}\rangle,\, \langle \texttt{i},\, \mathit{undef}\rangle\},\\
&\quad \beta-\alpha\geq 0\rangle
\end{aligned}$$

At this point, the two symbolic executions can be joined in a single execution by folding the two symbolic states. The resulting state is:

$$\begin{aligned}
\langle S_7,\, P_7\rangle &= \langle\{\langle \texttt{a},\, \alpha\rangle,\, \langle \texttt{b},\, \beta\rangle,\, \langle \texttt{diff},\, \gamma\rangle,\, \langle \texttt{fact},\, \mathit{undef}\rangle,\, \langle \texttt{sum},\, \mathit{undef}\rangle,\, \langle \texttt{i},\, \mathit{undef}\rangle\},\\
&\quad (\alpha-\beta>0\wedge\gamma=\alpha-\beta)\vee(\beta-\alpha\geq 0\wedge\gamma=\beta-\alpha)\rangle
\end{aligned}$$

This symbolic state holds before the execution of the expression `i = 1`[2] whose precondition is therefore:

[1] For sake of simplicity, in this example the empty lists of dynamic and global variables are not shown. Moreover, as the `main` function does not contain any function call and does not return any value, the local stack is simplified into a list of local variables, to make easier the reading.

[2] The statement `fact = i = 1` gives rise the two CCG vertices `(i = 1)` and `(fact =)` in that order.

$$(\alpha - \beta > 0 \wedge \gamma = \alpha - \beta) \vee (\beta - \alpha \geq 0 \wedge \gamma = \beta - \alpha)$$

From this formula the precondition $n \geq 0$ of the function *factorial* can be deduced, under the assumption $n = \gamma$ (n has been associated with `diff` and γ is the symbolic value of `diff`), i.e., $\forall$ α and β

$$((\alpha - \beta > 0 \wedge \gamma = \alpha - \beta) \vee (\beta - \alpha \geq 0 \wedge \gamma = \beta - \alpha)) \wedge n = \gamma \Rightarrow n \geq 0$$

This also means that once the values of α and β have been assigned (i.e., they are constants) and n and `diff` have been bound, the path-condition above and the precondition of the function *factorial* can be considered equivalent. Therefore, `i = 1` is candidate to be the initial statement of the slicing criterion. The symbolic state holding before the execution of the `while` statement is:

$$\begin{aligned}\langle S_8, P_8 \rangle \;=\; & \langle \{ \langle \texttt{a}, \alpha \rangle, \langle \texttt{b}, \beta \rangle, \langle \texttt{diff}, \gamma \rangle, \langle \texttt{fact}, 1 \rangle, \langle \texttt{sum}, 0 \rangle, \langle \texttt{i}, 1 \rangle \}, \\ & (\alpha - \beta > 0 \wedge \gamma = \alpha - \beta) \vee (\beta - \alpha \geq 0 \wedge \gamma = \beta - \alpha) \rangle\end{aligned}$$

At this point, the `while` loop is encountered. Symbolic execution can be used to obtain the recurrence equations from which the loop invariant is generated [57]. Let

$$\langle LS_1, LP_1 \rangle \;=\; \langle \{ \langle \texttt{a}, \lambda \rangle, \langle \texttt{b}, \mu \rangle, \langle \texttt{diff}, \nu \rangle, \langle \texttt{fact}, \rho \rangle, \langle \texttt{sum}, \psi \rangle, \langle \texttt{i}, \xi \rangle \}, PC \rangle$$

be the symbolic state resulting from $n - 1$ iterations of the loop and let us execute the n^{th} iteration (supposing that the $PC \Rightarrow \xi \leq \nu$). The execution of the loop body produces the state:

$$\langle LS_2, LP_2 \rangle \;=\; \langle \{ \langle \texttt{a}, \lambda \rangle, \langle \texttt{b}, \mu \rangle, \langle \texttt{diff}, \nu \rangle, \langle \texttt{fact}, \rho * \xi \rangle, \langle \texttt{sum}, \psi + \xi \rangle, \langle \texttt{i}, \xi + 1 \rangle \}, PC \rangle$$

From the analysis of the symbolic values in the states $\langle LS_1, LP_1 \rangle$ and $\langle LS_2, LP_2 \rangle$ it is easy to see that the variables that change their value during the execution of the loop are `fact`, `sum` and `i`. The recurrence equations for these variables can be written as:

$$\begin{aligned}\texttt{fact}_n &= \texttt{fact}_{n-1} * \texttt{i}_{n-1} \\ \texttt{sum}_n &= \texttt{sum}_{n-1} + \texttt{i}_{n-1} \\ \texttt{i}_n &= \texttt{i}_{n-1} + 1 \\ \texttt{i}_n &\leq \texttt{diff}_0 \\ \texttt{fact}_0 &= 1 \\ \texttt{sum}_0 &= 0 \\ \texttt{i}_0 &= 0 \\ \texttt{diff}_0 &= \gamma\end{aligned}$$

where the value for $\texttt{fact}_0$, $\texttt{sum}_0$, $\texttt{i}_0$ and $\texttt{diff}_0$ are obtained from the initial state $\langle S_8, P_8\rangle$. The solution for the system above is:

$$\begin{aligned}\texttt{fact}_n &= n!\\ \texttt{sum}_n &= n*(n+1)/2\\ \texttt{i}_n &= n+1\\ \texttt{i}_n &\leq \gamma\end{aligned}$$

from which, substituting the value of n obtained from the third equation in the first and second equation and eliminating the subscript n, we obtain the loop-invariant:

$$\begin{aligned}\texttt{fact} &= \texttt{(i-1)!}\\ \texttt{sum} &= \texttt{i*(i-1)/2}\\ \texttt{i} &\leq \gamma\end{aligned}$$

By associating `fact`, `sum` and `i` with the new symbolic constants δ, σ and ι, respectively, and executing the loop invariant in the symbolic state $\langle S_8, P_8\rangle$ we obtain the state holding before any iteration:

$$\begin{aligned}\langle S_9, P_9\rangle = &\ \langle\{\langle \text{a}, \alpha\rangle, \langle \text{b}, \beta\rangle, \langle \texttt{diff}, \gamma\rangle, \langle \texttt{fact}, \delta\rangle, \langle \texttt{sum}, \sigma\rangle, \langle \texttt{i}, \iota\rangle\},\\ &((\alpha-\beta>0\wedge\gamma=\alpha-\beta)\vee(\beta-\alpha\geq 0\wedge\gamma=\beta-\alpha))\wedge\\ &\delta=(\iota-1)!\wedge\sigma=\iota*(\iota-1)/2\wedge\iota\leq\gamma\rangle\end{aligned}$$

To obtain the symbolic state holding at the exit of the loop, let us suppose that the condition of the loop is false (i.e. $\iota=\gamma+1$). The resulting state will be:

$$\begin{aligned}\langle S_{10}, P_{10}\rangle = &\ \langle\{\langle \text{a}, \alpha\rangle, \langle \text{b}, \beta\rangle, \langle \texttt{diff}, \gamma\rangle, \langle \texttt{fact}, \delta\rangle, \langle \texttt{sum}, \sigma\rangle, \langle \texttt{i}, \iota\rangle\},\\ &((\alpha-\beta>0\wedge\gamma=\alpha-\beta)\vee(\beta-\alpha\geq 0\wedge\gamma=\beta-\alpha))\wedge\\ &\delta=(\iota-1)!\wedge\sigma=\iota*(\iota-1)/2\wedge\iota=\gamma+1\rangle\end{aligned}$$

This symbolic state holds before the execution of the statement `printf("%d \n", fact)` whose precondition is then:

$$((\alpha-\beta>0\wedge\gamma=\alpha-\beta)\vee(\beta-\alpha\geq 0\wedge\gamma=\beta-\alpha))\wedge\iota=\gamma+1\wedge\delta=(\iota-1)!\wedge\sigma=\iota*(\iota-1)/2$$

which can also be written as:

$$((\alpha-\beta>0\wedge\gamma=\alpha-\beta)\vee(\beta-\alpha\geq 0\wedge\gamma=\beta-\alpha))\wedge\iota=\gamma+1\wedge\delta=\gamma!\wedge\sigma=\gamma*(\gamma+1)/2$$

It is easy to prove, in a similar way as before, that whenever the precondition of the specification of the function *factorial* holds true, the condition above is equivalent to the postcondition of the specification. Moreover, as the statement `i = 1` dominates the statement `printf("%d \n", fact)` the slicing criterion ⟨`i = 1, printf("%d \n", fact), fact`⟩ is produced. The slice on this slicing criterion can be easily extracted using the algorithm described in chapter implementation. The slice produced is:

```
fact = i = 1;
while (i <= diff) {
   fact *= i;
   i++;
}
```

5.2 A Case Study

The program chosen to demonstrate the specification driven program slicing process is the PERPLEX tool [30], a subsystem of the CCG analyser [113, 114]. The CCG system is concerned with the production of a data base of Prolog [164] facts containing the CCG representation of a C program and consists of two subsystems (see section 4.1).

The first subsystem is the PERPLEX tool which is written using the YACC [107] compiler-compiler and uses a grammar corresponding to that in [110]. PERPLEX produces a Prolog data base containing several types of facts that support intraprocedural information about the individual FCCGs. In particular, the data base contains facts for *syntax tree nodes*, *syntax tree edges*, *control flow nodes*, *control flow edges*, *type declarations*, *function declarations*, *variable declarations*, *formal parameter declarations.* Other fact types represent *expression-use* and *lvalue-definition* edges, which carry information about the internal data flow in expressions containing embedded side-effects or control flows, and the scope of the identifiers in the syntax tree.

The second subsystem consists of three analysis meta programs [113, 114] written in Prolog that enrich the data base produced by PERPLEX with interprocedural information (*call edges*, *parameter binding edges* and *return expression-use edges*), to connect the individual FCCGs and make up the CCG, and control and data dependencies.

The PERPLEX subsystem is over 5000 LOC long (not including comments) and consists of four header files, seven C files and one YACC file. The original version of the system [30] did not produce the abstract syntax tree in the FCCG representation. Moreover, it did not produce extra vertices for sub-expressions containing embedded side-effects or control flows, nor *expression-use* and *lvalue-definition* edges. The latter functionality has been added to

File	LOC 1	LOC 2	LOC 3	diff. 1/2	diff. 2/3	diff. 1/3
bunter.h	9	9	9	0	0	0
data.h	141	153	164	12	11	23
global.h	86	86	86	0	0	0
lex.h	7	7	7	0	0	0
alloc.c	367	393	393	26	0	26
ccg_ld.c	78	78	78	0	0	0
interface.c	73	123	123	50	0	50
lex.c	353	353	386	0	33	33
main.c	119	143	144	24	1	25
print_fun.c	766	1550	2542	784	992	1776
supply_fun.c	399	489	498	90	9	99
perplex.y	1014	1020	1041	6	21	27
Total	3412	4404	5481	992	1067	2059

Table 5.1: PERPLEX files: LOC of original (1), second (2) and third (3) versions

PERPLEX during a first perfective maintenance intervention which allowed the computation of more accurate data dependencies and program slices [114]. A second maintenance intervention allowed PERPLEX to produce the FCCG syntax trees, in order to use CCG for symbolic execution [57]. Table 5.1 shows the number of LOC of each file of the system in the original version (1) and after the two perfective maintenance operations (2 and 3, respectively). The final version is over 2000 LOC longer than the original one.

5.2.1 The sample functions

In both the maintenance operations most functionalities have been added to the `print_fun.c` file, as the new version has over 1700 LOC more than the original one. This file contains the functions for the production of the file containing the Prolog data base, while the other C files concern lexical analysis (`lex.c`), dynamic memory allocation (`alloc.c`), data structure traversal (`supply_fun.c`), interface management (`interface.c`) and multiple input files management (`ccg_ld.c`), besides the main program file (`main.c`). After the first maintenance operation the number of functions in the `print_fun.c` file increased from 20 to 30 and the new functions were composed of over 600 LOC. On the contrary, after the second maintenence operation, although the number of new functions has increased from 30 to 54,

Function	LOC 1	LOC 2	LOC 3	diff. 1/2	diff. 2/3	diff. 1/3
`stmt_print`	179	266	392	87	126	126
`leaf_expr`	0	130	230	130	100	230
`access_print`	0	123	202	123	79	202
`anon_access_print`	0	96	168	96	72	168
Total	179	615	992	436	377	813

Table 5.2: File `print_fun.c`: LOC of original (1), second (2) and third (3) versions of the sample functions

the number of LOC for the new functions is about 500 only. Therefore, the other LOC have been added to earlier functions and in particular, about 400 LOC have been added to four functions only, each of them is over 150 LOC in the final version, as shown in table 5.2. All the other functions in the `print_fun.c` file have less than 150 LOC. The four functions are `stmt_print`, for the production of information about statements (*expression* statements, *compound* statements, *selection* statements, *loop* statements, *jump* statements and *label* statements), `leaf_expr`, for the production of information about atomic expressions (like identifiers with increment or decrement operators), `access_print` and `anon_access_print` for the access to function parameters, arrays, structures and pointers. In particular, the function `access_print` is called by the function `leaf_expr` to access atomic side-effected structures, like for example `a[2][1]++`, while the function `anon_access_print` deals with expressions without side-effects, like for example `(*(a+2))[1]` or `*(a+2)[1]`. Note that the last three functions were not present in the original version of PERPLEX and have been introduced during the first maintenance intervention.

Meaningful subfunctionalities have been isolated in these functions using the specification driven slicing process. After the isolation phase, the functions have been reengineered. The interface of each new function has been defined and its code in the existing functions has been substituted with a function call. The case study also revealed the existence of duplicated code in the functions `leaf_expr`, `access_print` and `anon_access_print`, as the code of some of the isolated subfunctions was found more than once. The reengineering phase allowed to eliminate the duplicated code and to cluster similar functions into a more general one.

5.2.2 The function stmt_print

The function `stmt_print` has been decomposed in 6 functions as shown in table 5.3. The main function `stmt_print` calls the other functions depending on the type of the statement

Function	LOC
stmt_print	58
comp_stmt_print	39
sele_stmt_print	87
iter_stmt_print	150
label_stmt_print	62
jump_stmt_print	90
Total	486

Table 5.3: Decomposition of stmt_print

to be printed. In particular:

- comp_stmt_print deals with *compound* statements
- sele_stmt_print deals with *selection* statements
- iter_stmt_print deals with *loop* statements
- label_stmt_print deals with *label* statements
- jump_stmt_print deals with *jump* statements

while in the earlier version a function expr_print dealing with *expression* statements was already called by the function stmt_print. For each of the identified functions it was very easy to find the statement s_{in} of the corresponding slicing criterion. Indeed, the function stmt_print is characterised by a switch statement on the expression statement->type[3] carrying the type of the statement. The result of the symbolic execution of the switch statement was the generation of several execution paths corresponding to the different case labels. Each of the resulting path-conditions was equivalent to the precondition in the specification of one of the functions above. For example, the precondition of the subfunction comp_stmt_print was found after the execution of the statement:

```
switch (statement->type) {
  . . . . .
  case COMP:
    block_c_up (block_counter);
```

[3] The variable statement is a pointer to a structure modelling a statement node in the syntax tree. The structure is defined in the file data.h.

Function	LOC
leaf_expr	68
fun_call_expr	59
create_leaf_expr	75
no_create_leaf_expr	45
Total	247

Table 5.4: Decomposition of leaf_expr

As statement is a formal parameter of the function stmt_print, let α be the value of the field statement->type. At this point the symbolic state contains the path-condition:

$$\alpha = \texttt{COMP}$$

which is equivalent to the precondition of the function. Therefore, the statement:

```
block_c_up (block_counter);
```

was chosen as initial statement of the slicing criterion. The function block_c_up increments a counter carrying information about the control nesting. It was more difficult to find the postcondition of the functions, due to the recursive nature of stmt_print. User interaction was required to understand the code and to provide some assertions. Symbolic execution was used to understand the behaviour of the recursive calls. For example, for the function comp_stmt_print the following break statement was chosen as final statement of the slicing criterion:

```
block_c_down (block_counter);
break;
```

Indeed, the precondition of the break statement above was equivalent to the postcondition of the function. Note that the function block_c_down decrements the counter carrying information about the control nesting and therefore, it is actually the last statement of the function comp_stmt_print.

5.2.3 The function leaf_expr

The function leaf_expr has been decomposed in 4 functions as shown in table 5.4. The main function leaf_expr calls the other functions. In particular, fun_call_expr is called whenever the sub-expression is a function call, create_leaf_expr is called whenever the sub-expression contains an embedded side-effect or control flow (in this case a node in the control flow graph must be created), while no_create_leaf_expr is called to deal with sub-expressions without embedded side-effects or control flows (no node in the control flow graph has to be created). For these functions also the identification of the preconditions was very easy. For example, the precondition of the function fun_call_expr was found after the execution of the statements[4]:

```
switch(expr->operator) {
   . . . . .
   case ID:
   . . . . . }
if((expr->in != NULL) &&
   (expr->in->type == FUN)) {
   call_node_no = *node_countp;
```

As expr is a formal parameter let α, χ and β be the values of the fields expr->operator, expr->in and expr->in->type, respectively[5]. Before the execution of the statement:

```
call_node_no = *node_countp;
```

the symbolic state contains the path-condition:

$$\alpha = \text{ID} \wedge \chi \neq \text{NULL} \wedge \beta = \text{FUN}$$

which is equivalent to the precondition of the function. Therefore, this statement was chosen as the initial statement of the slicing criterion for the isolation of the function. User interaction was required to find the postcondition of the function and the final statement for the slicing criterion. The statement chosen was the return statement in the following code fragment:

```
if (val == LVALNODE)
   int_list_add (lvalp, call_node_no);
else
   int_list_add (rvalp, call_node_no);
return;
```

Similarly, the two different symbolic states generated by the execution of the if statement:

[4] The variable expr is a pointer to a structure modelling an expression node in the syntax tree. The structure is defined in the file data.h.

[5] The variable expr->in is a pointer of the same type as expr. Therefore, the value χ is the symbolic address of the structure pointed by expr->in.

Function	LOC
access_print	48
anon_access_print	42
acc_fun_print	52
acc_arr_print	35
acc_ptr_print	24
acc_ptr_ref_print	23
acc_str_ref_print	30
set_def_ref	38
Total	292

Table 5.5: Decomposition of access_print and anon_access_print

```
if((makenode == NOTCREATE) &&
   (expr_in_post_pre(expr) == FALSE))
```

carry the preconditions for the functions no_create_leaf_expr and create_leaf_expr, respectively. Indeed, the variable makenode is a parameter of the function which asserts that a node in the control flow must be created for the current expression (pointed by expr), while the function expr_in_post_pre checks for embedded side-effects due to increment or decrement operators. User interaction was required to find the postconditions of the two functions. The initial and the final statements of the slicing criteria were a switch statement and a return statement, respectively, for both the functions, as shown in the following code fragment:

```
switch (expr->def_ref) {. . .}
. . . . .
return;
```

5.2.4 The functions access_print and anon_access_print

Some interesting cases of duplicated code and generalisation have been found in the functions access_print and anon_access_print. Indeed, the only difference between the two functions is that the former also deals with the access to the actual parameters of a *function call* expression. The code for this function has been identified, isolated and reengineered in the function acc_fun_print. The other subfunctionalities of the two functions are the same and concern the access to array subscripts, pointers, structure fields and pointer structure fields (e.g., a->b, where a is a pointer variable). The code of each of these functionalities was the same in both the original functions. This duplication was justified because in the original design the two functions were intended to deal with two different cases. Moreover, as the common subfunctionalities were too simple (and the resulting code too small), the designer decided to not create functions for them and to have small pieces of duplicated code. However, during the perfective maintenance phase, the same code has been added to these subfunctionalities, making the dimension of the duplicated code meaningful. This problem has been discovered during the isolation phase and eliminated in the reengineering phase, by clustering the code of the common subfunctionalities in the functions acc_arr_print, acc_ptr_print, acc_ptr_ref_print and acc_str_ref_print, respectively, as shown in table 5.5.

The precondition of each function was found after the execution of a case label of a switch statement, while the postcondition was equivalent to the precondition of the break statement terminating the *switch case*. For example, for the function acc_arr_print the initial and final statements of the slicing criterion were switch (expr->def_ref) and break, respectively, in the following code fragment:

```
switch (expr->type) {
  case ARR:
    switch (expr->def_ref) {. . .}
    . . . . .
    break;
```

Table 5.5 also contains the function set_def_ref. This function updates the list of the definitions and uses of the identifiers in the current *sub-expression node* of the control flow graph. This information is used for the data flow analysis. Moreover, the function returns the label of the node in the syntax tree whenever the current operator is an increment or decrement operator or the & unary operator (which returns the address of a variable). The code for this function was found in different versions in the functions acc_arr_print, acc_ptr_print, acc_str_ref_print, create_leaf_expr and no_create_leaf_expr. Again, in the original design this functionality was too simple to be implemented as a function, but after the perfective maintenance the number of duplicated LOC was meaningful.

Figure 5.2 and 5.3 show part of the code fragments isolated in the functions acc_arr_print and acc_ptr_print, respectively. Actually, the only difference among the different versions of isolated code fragments was in one of the actual parameters appearing

```
char *ptr = "*";
char *no_op = "@";
char *labnode;
labnode = NULL;
switch (expr->def_ref) {
  case DEF_AND_REF:
    var_list_add (var_listp, ptr, REF);
    . . . . .
    /* compute labnode and var_listp for
       increment/decrement operators and use no_op */
    break;
  case DEF_ONLY:
    var_list_add (var_listp, ptr, DEF);
    break;
  case ADDRESS:
    var_list_add (var_listp, ptr, ADDR);
    labnode = make_string("address_of");
    break;
  default:
    var_list_add (var_listp, ptr, REF); }
```

Figure 5.2: Code fragment isolated in acc_arr_print

```
char *dot = ".";
char ident[200];
char *no_op = "@";
char *labnode;
labnode = NULL;
strcpy (ident, dot);
strcat (ident, expr->ident);
switch (expr->def_ref) {
  case DEF_AND_REF:
    var_list_add (var_listp, ident, REF);
    . . . . .
    /* compute labnode and var_listp for
       increment/decrement operators and use no_op */
    break;
  case DEF_ONLY:
    var_list_add (var_listp, ident, DEF);
    break;
  case ADDRESS:
    var_list_add (var_listp, ident, ADDR);
    labnode = make_string("address_of");
    break;
  default:
    var_list_add (var_listp, ident, REF); }
```

Figure 5.3: Code fragment isolated in acc_ptr_print

in the calls to the function var_list_add. This function updates the list var_listp containing the variables defined and used in the current expression. For example, in figure 5.2 the actual parameter is ptr, while in figure 5.3 it is ident. The different versions of isolated code fragments have been generalised and clustered in a function at a higher abstraction level, as shown in figure 5.4.

5.3 Evaluation and Conclusion

In this chapter an example and a case study have been shown to demonstrate the specification driven program slicing process. The example showed some typical problems of symbolic execution, like loop invariants and theorem proving. Due to the possible presence of loops and recursion in a function the problem of abstracting the precondition of a statement is *undecidable* and the user interaction is necessary. However, where possible a deterministic approach based on symbolic execution can be used to automatically recover the recurrence equations for the variables involved in a loop. The solution of these equations provides invariant assertions which allow symbolic execution to continue. In particular loops involving pointer and array variables are difficult to deal with. A solution to the problem of finding loop invariants for programs with arrays has been proposed by Ellozy [78]. On the same line and using symbolic addresses for memory locations as proposed in this paper, a solution can be provided whenever a loop involves arithmetics on pointer variables. For more complex loops a library of (domain independent) plans can be used to recover invariants [3].

However, symbolic execution allows the user to follow the different execution paths of a program and then to understand how the program variables change on the different paths. In this case symbolic execution can be used as a tool for program comprehension [19]. Finally, whenever the user is not able to provide a suitable loop invariant, a "partial" specification can be obtained by executing the loop a fixed number of times and adding this constraint to the path-condition.

Theorem proving is a fundamental aspect of the specification driven program slicing. Implications have to be proved between the path-condition and the pre and postcondition of the function specification. Moreover, implication must also be proved between the path condition and a predicate encountered, in order to avoid *infeasible paths.* Theorem proving is a very expensive activity, in particular if a proof involving the path condition and the function specification is required at each step. The interactivity of the tool makes more flexible this task, because the user can decide when a proof is necessary. Also the user can decide to discard some execution paths, so reducing the number of proofs involving path-conditions and program predicates.

```
char *set_def_ref(expr, var_listp, id_ptr)
   EXPR_PTR expr;
   VAR_LIST_PTR *var_listp;
   char *id_ptr;
{
   char *no_op = "@";
   char *labnode;
   labnode = NULL;
   switch (expr->def_ref) {
     case DEF_AND_REF:
       var_list_add (var_listp, id_ptr, REF);
       . . . . .
       /* compute labnode and var_listp for
          increment/decrement operators and use no_op */
       break;
     case DEF_ONLY:
       var_list_add (var_listp, id_ptr, DEF);
       break;
     case ADDRESS:
       var_list_add (var_listp, id_ptr, ADDR);
       labnode = make_string("address_of");
       break;
     default:
       var_list_add (var_listp, id_ptr, REF); }
   return (make_string(labnode));
}
```

Figure 5.4: Reengineered function `set_def_ref`

Another problem in symbolic execution is due to module calls. We adopt the macro-expansion approach, rather than the less expensive lemma approach, because it considers the symbolic state at the call site, which actually contains the precondition for the called function. Besides an improved precision of the process, this approach also allows the user to understand the function behaviour in the context of the calling function.

The case study demonstrated the feasibility of the method and its applicability to any programming language. Although until now program slicing has been used to isolate reusable functions in large monolithic programs written in COBOL language [45, 41, 65, 66, 124, 136], we are confident that this approach can give good results also for programs written in C language. Indeed, even though the C language provides a primitive for implementing functional abstractions, very often a single C *function* implements more than one functional abstractions. The case study presented in this chapter validated this assertion. The specification driven program slicing process has been applied to PERPLEX [30], a subsystem of the CCG analyser [114] for C programs. Four large functions implementing more than one functionality have been identified and reengineered by decomposing them into smaller functions each of which implements one functionality. Figure 5.5 shows the call graph of the reengineered functions. Solid lines represent call edges, while dashed lines denote the existence of a path between two functions on the call graph. The effort required for the comprehension process during the maintenance of these functions has been reduced and their reusability has been improved as a result of the reengineering process. The case study demonstrated that even though the original design has been made following software engineering principles, maintenance operations (in particular perfective maintenance) can add new functionalities (in terms of code) to existing functions and sometime the same code in different functions or even in different places in the same function.

Other considerations can be made about the use of CCG for program slicing. The refined vertices of the CCG ensure that only a single program object is defined at any single vertex and that any flow dependency incident on that vertex also involves a single variable contributing to the defined object [113, 114]. As a result, more accurate slices can be obtained. For example, note that in the statement `sum += i++` of the sample program of section 5.1 only the expression `i++` is considered in the resulting slice. This accuracy is possible thanks to the extra vertices representing embedded side-effects in the CCG. Figure 5.6.a shows a partial view of the CCG corresponding to the code fragment:

```
fact = i = 1;
sum = 0;
while (i <= diff) {
   fact *= i;
```

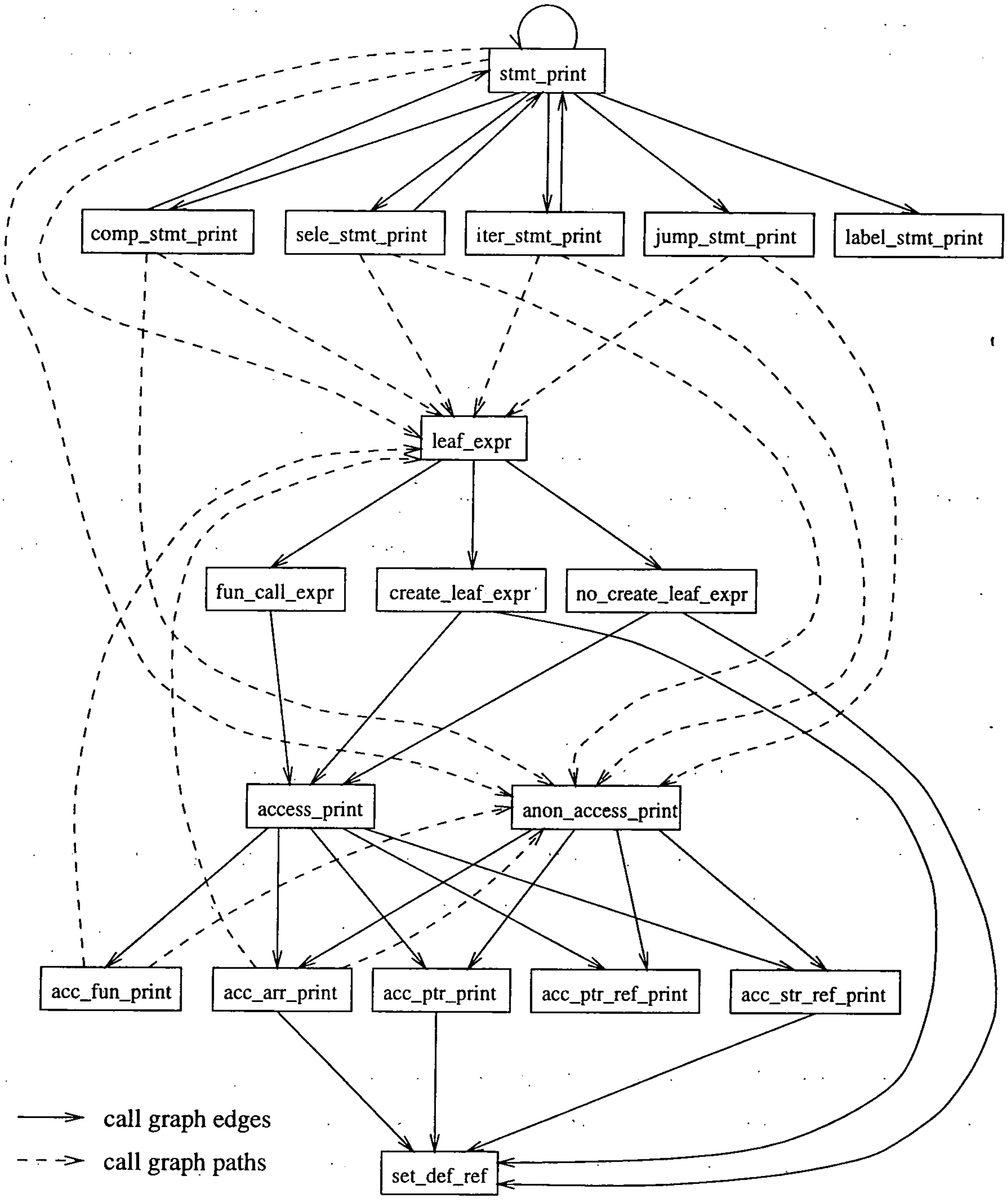


Figure 5.5: Call graph of the reengineered functions

```
        sum += i++;
    }
    printf("%d \n", fact);
```

Control and data dependence edges and expression-use edges are represented in the figure. If no extra nodes are created for the embedded side-effects `i = 1` and `i++` the graph in figure 5.6.b results and the program slice becomes

```
    fact = i = 1;
    sum = 0;
    while (i <= diff) {
        fact *= i;
        sum += i++;
    }
```

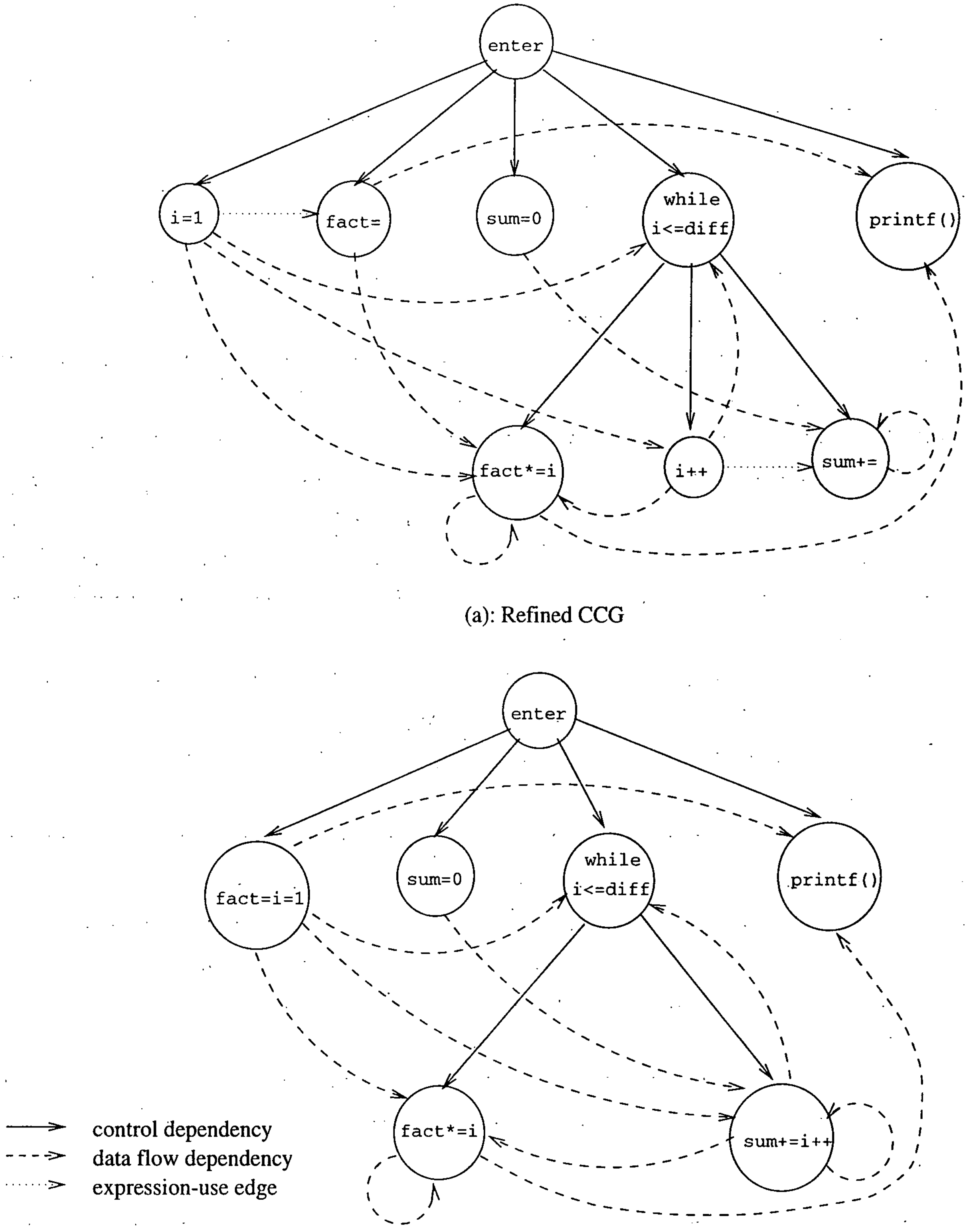
However, the algorithm accuracy is affected by two limitations of the CCG implementation. First, the absence of information about the memory location involved by a data flow dependency allows accurate slices only on a variable defined at a given vertex, or on each variable defined or used at the vertex. Slices on a single variable used at the vertex would not be precise. For example, let us consider the statement:

```
a = x + y;
```

A slice on variable a will include any vertices which contribute to the values of x and y. As both x and y contribute to the value of a, the slice is correct. On the contrary, a slice on the variable x would not be precise, because the result would also include vertices affecting the value of y which does not contribute to the value of x. This problem could be overcome by annotating each flow dependency in the CCG with information about the memory location involved. The program slicer can be consequently modified to take into account this information.

Another limitation of the CCG is the lack of calling context information in the graph during interprocedural slicing. Calling context information allows a slice descending into a function to correctly return to only the call site from which the function was entered. On the contrary, the lack of such information causes the slice to return to any other call site, resulting in an inaccurate overly-conservative slice [176].

This problem has been solved by Horwitz *et al.* [103] by developing a two-phase traversal algorithm based on the SDG representation modelling multiple-procedure programs with parameters passed by value-result. Explicit vertices represent actual and formal input/output parameters while binding edges link actual input to formal input parameters and formal



(a): Refined CCG

(b): Embedded side-effects

Figure 5.6: Enhanced slicing accuracy with refined CCG

output to actual output parameters. In this way, *transitive interprocedural data flow dependencies* due to the effects of procedure calls are computed by an attribute grammar and encapsulated between the actual input and actual output parameters of the call interfaces, so avoiding the use of explicit data flow dependencies between vertices of the SDG. The two-phase graph traversal algorithm makes use of these new transitive edges to solve the problem of calling context. In the first phase the algorithm traverses the graph moving 'across' a call (through the transitive dependencies) without descending in the called procedure. In the second step the graph is traversed without ascending to call sites.

Horwitz *et al.* [103] do not consider pointer variables and pointer-induced aliasing. These problems that are typical of C programs are considered in the data-flow analysis algorithm performed on the CCG representation [112, 114]. In order define a two phase traversal algorithm based on that described in [103] a new representation for pointer parameters would be required to explicitely model the referenced objects by formal and actual parameter vertices. However, the presence of explicit data dependence edges in the CCG prevents the application of the SDG traversal algorithm. A program slice proceeds throughout the CCG representation via the CCG dependences and the absence of calling context information causes the backward slicing algorithm to consider all the incoming interprocedural edges of a CCG vertex, so resulting in an inaccurate slice.

A simple solution to this problem, in absence of recursive functions, has been suggested by Kinloch [114] and consists in introducing separate copies of each FCCG for each individual call site. Although in this way calling context problems are eliminated completely, the space requirements of multiple FCCG representations are prohibitive.

An alternative, but not expensive solution to this problem can be provided by adding calling context information to the interprocedural flow dependencies. In particular, any interprocedural flow dependency should contain the list of the call sites encountered in the interprocedural control flow path between the vertex which defines the value of a memory location and a vertex which uses this value. This information can be carried during the data flow analysis performed on the CCG. The order of the call sites in the list must be inverted for ascending flow dependencies[6] (enclosing return-expression-use edges) with respect to descending flow dependencies (enclosing parameter binding edges and call edges). The algorithm can then be modified in the following way:

1. control dependence, expression-use and lvalue-definition edges are considered as usual

[6] An interprocedural flow dependency between a vertex n in the FCCG of a function p and a vertex m in the FCCG of a function q is descending if and only if it has been generated through an interprocedural control flow path containing a call chain from p to q. The interprocedural flow dependency is ascending if and only if the function p returns the control to the function q through the interprocedural control flow path.

during the backward traversal of the graph;

2. an ascending flow dependency is considered as usual during the backward traversal of the graph and the information about the corresponding list of the call sites is saved in the slice

3. a descending flow dependency is not considered until the corresponding list of call sites is not a sublist of an already considered ascending dependency's list.

This prevents a program slice propagating along any interprocedural edges from a given function and returning out of the context call sites along these edges. Moreover, the algorithm terminates when a fixed point is reached (no flow dependency can be added anymore to the slice).

Chapter 6

Conclusion

The work described in this thesis addresses the field of software reuse. Software reuse is a most promising approach to solve the problem of the software crisis. The reuse of software components, either design and specification documents or pieces of code, can increase the productivity and improve the quality and reliability of new software systems, leading software engineering to the state of an assessed engineering discipline.

Although reusable software components can be designed and implemented during the development of new software projects, existing software is widely considered to be the main source for the extraction of reusable assets. Moreover, existing software systems play an important role in the economy of a company. Very often the evolution of a legacy system is a critical problem and its modularisation according to the software engineering principles can drastically reduce the maintenance effort. Modularising an existing system involves searching it for software components to be reengineered and reused in the modularised system. However, the reuse of these software components should not be limited to the system to be modularised. On the contrary, reusable software components should be qualified, classified and stored in a repository in order to reuse them during the development of new software systems.

The term reuse reengineering refers to processes aiming to exploit reverse engineering and reengineering techniques for identifying and extracting reuse-candidate software components from existing systems, reengineering them according to predefined module templates and interconnection standards, producing their specifications, in order to populate a repository of reusable assets. Different paradigms for setting up reuse reengineering processes have been proposed in the literature. In particular, a reference paradigm has been defined within the RE^2 project. The key role of the paradigm is to define a framework where relevant methodologies and tools can be systematically used and experiments can be repeated. The thesis has focused on the first phase of the RE^2 project reference paradigm, called Candidature phase, which is concerned with the analysis of source code for the identification of software components implementing abstractions. Such components are candidate to be reengineered and reused. Several program representations and code scavenging techniques have been described. Candidature criteria have been classified in structural and specification driven criteria. A new specification driven candidature criterion has been presented which uses a formal specification of the functional abstraction to be recovered and is based on the theoretical frameworks of program slicing and symbolic execution. The implementation and the evaluation of the method have also been presented. In the next section a discussion of the criteria for success stated in the introduction is outlined.

6.1 Evaluation of the Criteria for Success

In chapter 1 the criteria for success are stated as follows:

- review of existing program representations;
- description and evaluation of existing methods for the identification of components driven by structural properties of the software;
- description and evaluation of existing methods for the identification of components driven by the specification of the abstraction sought;
- development of a new specification driven method for the identification of code fragments implementing functional abstractions;
- formalisation of the new method;
- prototype implementation of the new method to show that it is automatable;
- evaluation of the new method by the use of a case study.

Each of the criteria is now addressed for the evaluation of the thesis.

Review of existing program representations

Program representations play a key role in a reuse reengineering process. Several program representations have been proposed in the literature that can be used in reuse reengineering and in general, in software maintenance and software engineering environments. Each representation try to model a particular aspect of the program. For example, the control flow graph statically models the internal execution flow of a procedure, while the call graph considers the possible flow of control at the interprocedural level. Other representations outline the control and dependencies between the statements in a program.

However, in a reuse reengineering process different program representations can be necessary in order to solve different problems. Several attempts have been made to combine the features of different representations. Merging different representations has to deal with efficiency and space problems. Two representations have been identified that better achieve these results, the web representation [130] and the unified interprocedural graph [98]. However, for the purpose of the work of this thesis the two representation deal with complementary aspects of a program: the web representation is more suited for combining syntactic and semantic aspects of the program, while the unified interprocedural graph better summarises interprocedural level information.

Description and evaluation of existing methods for the identification of components driven by structural properties of the software

Most existing methods search for reuse-candidate software components based on structural properties of the software. Structural candidature criteria are applied to one or more systems and produce a large set of candidate modules. A concept assignment process is required to be applied to the candidate modules in order to associate them with human oriented meanings. Modules that cannot be associated with any human oriented meaning are discarded.

Structural candidature criteria can be classified in methods driven by a metric model (METMOD) and methods driven by the type of the abstraction to be sought (METTYP). METMOD methods entail the selection of a set of metrics and the definition, for each metric, of a value range that may be considered characteristic of code implementing a reusable abstraction. METTYP methods are specialised to search for only one type of abstraction and are generally defined in terms of summary relations [35] obtained by combining relations directly produced through static analysis of code. METTYP methods searching for both functional and data abstractions have been described.

Structural candidature criteria have been experimented in several successful case studies. They are generally easy to implement and efficient, because only require static analysis of the code and of the model of the program chosen to apply the criterion. However, some of these criteria (in particular methods searching for data abstractions) have only been experimented on programs written in languages such as C and Pascal and are not yet mature to be scaled up to large software systems written in COBOL.

On the other hand, slicing methods, that have been extensively used on monolithic COBOL systems, suffers of the difficulties in identifying a suitable slicing criterion for isolating the functions of interest. Very often, they are only successful in identifying external functionalities, by using slicing criteria involving output statements.

Description and evaluation of existing methods for the identification of components driven by the specification of the abstraction sought

A different approach in the identification of reusable software components in code is to use the specification of the abstraction to be sought. The system is then searched for code fragments which implement the specification. The code component extracted does not require to be validated by a concept assignment process.

Specification driven candidature criteria have been classified, depending on the form of specification, in methods driven by a formal specification of the abstraction, methods based on a set of test cases, and knowledge based methods.

Knowledge based methods encode the knowledge about the functions to be identified in the form of programming plans and make use of an internal representation of the program for mapping program actions to these plans. The main limitation of knowledge-based methods is that they can require a large library of plans. Moreover, bottom-up approaches can also suffer of a combinatorial explosion. While this approach can be effective for recognising stereotypical domain independent plans, it can be too expensive for dealing with domain dependent functions and then not convenient for a reuse reengineering process.

A way to provide the specification of a functionality is by carefully designing a set of test cases for a program. The set of test cases expresses a behaviour of the program corresponding to an external functionality. The functionality can be isolated by simply instrumenting the program and extracting the components exercised. Methods based on test cases are very efficient and and easy to implement, but can suffer of lacking of precision when different functionalities share same components. In this case the reuse-candidate module identified could be too large and include more functionalities than the one sought.

A more precise method consists in using a set of test cases together with slicing techniques. This method, called simultaneous slicing [92], is a generalisation of dynamic slicing [118] and produces more refined modules than the method above. Indeed, it also takes into account the data flow of the program by allowing the reduction of the set of selected statements. However, the method does not consider the problem of finding a slicing criterion and then it can only be used to identify external functionality. Lack of precision can result in identifying internal functions.

Methods driven by a formal specification of the abstraction to be sought can be suitably used to isolate both external and internal functionalities. However, methods proposed in the literature for the isolation of different kind of software components, do not use formal method tools for achieving a greater automation. User interaction is intensively required to map the specification of the abstraction onto the code.

Development of a new specification driven method for the identification of code fragments implementing functional abstractions

A new candidature criterion driven by a formal specification of the abstraction to be recovered has been developed. The effort has been devoted to the identification of formal method tools useful to map the specification of the function onto code. The method proposed looks for code fragments implementing functional abstractions and is based on program slicing as an isolation primitive.

The main result has been the use of symbolic execution and theorem proving techniques to automate the identification of the slicing criterion from which the slice implementing the required functional abstraction can be extracted. To this aim the formal specification of a function, given in terms of a pre and postcondition is compared, by theorem proving techniques, with the invariant assertions for the statements abstracted by using symbolic execution.

Although the method allows more automation compared with other methods based on program slicing, it is not completely automatable, due to some undecidable problems. Human interaction is required in this cases. Human interaction is also required to associate the data of the functional abstractions with the program statement. This is mainly a concept assignment problem [18] where human factors are involved [1].

Formalisation of the new method

The method proposed has been specialised and formalised for programs written in C language. First, a combined interprocedural representation for C programs [112, 113, 114] suitable to perform program slicing has been identified and enhanced with syntactical information useful for symbolic execution. Problems involved in symbolic execution of C programs have been identified and solutions have been formally outlined. Also, an execution model based on the CCG has been proposed. Finally, an algorithm for slicing C programs using CCG and based on a new definition of slicing criterion and program slice, useful for isolating functional abstractions in code, has been described.

Prototype implementation of the new method to show that it is automatable

The method proposed has been implemented in Prolog. The Prolog environment has been chosen because it allows to easily implement logic theories, integrate tool, enhance a tool, and interact with the user.

The CCG representation of the program is obtained in terms of a data base of Prolog facts by a static analyser and three Prolog metaprograms. The core of the system consists of th symbolic executor and the theorem prover that allow the selection of a slicing criterion (in terms of vertices on the CCG), driven by to the input specification. Finally, the slicing algorithm is implemented in Prolog for the extraction of the slice implementing the functional abstraction.

Evaluation of the new method by the use of a case study

The method has been experimented in a case study conducted on a system written in C language. Four large functions implementing more than one functionality have been identified and reengineered by decomposing them into smaller functions each of which implements one functionality. The effort required for the comprehension process during the maintenance of these functions has been reduced and their reusability has been improved as a result of the reengineering process.

The case study demonstrated the feasibility of the method and the applicability of slicing techniques not only to monolithic programs written in a language like COBOL, but also to programs written in languages like C and Pascal that provides a primitive for implementing functional abstractions. Indeed, very often a single procedure implements more than one function. This happens in particular as a consequence of maintenance (in particular perfective) operations.

6.2 Further Work

A number of further researches issues can be addressed from the results of the work presented in this thesis. A first issue can be to scale up the method to large sized software systems. For example, the method could experimented to isolate functionalities in large and monolithic COBOL programs, where program slicing has already been used as structural candidature criterion.

Moreover, other approaches could be used based on program slicing and symbolic execution for the isolation of functional abstractions in code. For example, conditioned slicing [41], used for isolating function behaviours in code, could exploit symbolic execution techniques to instrument the program and identify the portion of the program that can be executed under the given condition. Program slicing could then be applied to the statements of the program that have been excited by the symbolic executor.

Finally, symbolic execution has already been used in the Qualification phase of the RE^2 reference paradigm for abstracting the specification from code of a module implementing a functional abstraction. The interactive symbolic executor can also be used as a tool for program understanding in a reuse reengineering environment [75] to support the concept assignment process [18] and select the subset of meaningful reuse-candidate modules (identified and extracted by structural methods) to be de-coupled and reengineered in the Election phase.

Appendix A

CCG Fact Base

In the following, the architecture of the data base of Prolog facts for the representation of declarations, abstract syntax tree, control flow graph, interprocedural edges and program dependences of the CCG is illustrated.

Declarations

The data base contains the following facts for representing source code files and the symbol table:

- `file(File-Name, File-ID)` associates a source code file name with a unique identifier;
- `type(File-ID, Function, sc(Stmt-Block, Storage-Specifier), Type-Specifier, Name, Access-List)` for user-defined type declarations;
- `tag(File-ID, Function, sc(Stmt-Block, Storage-Specifier), Tag-Type, Name, Member-List)` for `struct`, `union` or `enum` definitions;
- `object(File-ID, Function, sc(Stmt-Block, Storage-Specifier), Type-Specifier, Name, Access-List)` for variable and function declarations.

In the facts `type`, `tag` and `object`, the terms `File-ID`, `Function`, `Stmt-Block` and `Storage-Specifier` are sufficient to determine the scope of the declaration. In the facts `type` and `object`, the terms `Type-Specifier` and `Access-List` express the type of the item identified by the term `Name`. In the fact `tag`, the term `Tag-Type` can assume the values `struct`, `union` or `enum`, `Name` is the name of the structured item and `Member-List` is a list of the item's components (enumerated constants or terms of type `mem(Type-Specifier, Name, Access-List)` for `struct` and `union`.

Abstract Syntax Tree

Two types of facts are used for representing the nodes and the edges of the Abstract Syntax Tree (AST):

- `st_node(File-ID, Function, Node-Type, Node-ID)`
- `st_link(File-ID, Function, Edge-Type, Node-From, Node-To)`

where `File-ID`, `Function` and `Node-ID` identify a node of the AST, `Node-Type` and `Edge-Type` denote the types of a node and of an edge, respectively. Moreover, each node representing a compound statement (`Node-Type` is `group`) is associated with its scope by the fact

- `scope(File-ID, Function, Stmt-Block, Node-ID)`

while each node corresponding to an identifier (`Node-Type` is `id`) is associated with its declaration by the fact

- `id_decl(File-ID, Function, Node-ID, obj_loc(File-ID, Function, sc(Stmt-Block, Storage-Specifier), Name))`

Control Flow Graph

Two types of facts are used for representing the nodes and the edges of the Control Flow Graph (CFG):

- `node(File-ID, Function, Node-ID, Node-Type, Node-Qual, Expr-List)`
- `edge(File-ID, Function, Node-From, Node-To, Edge-Label)`

where `File-ID`, `Function` and `Node-ID` identify a node of the CFG, `Node-Type` is the type of the node and `Node-Qual` is an identifier that qualifies the node or `@` if undefined. `Expr-List` is a list of terms `ex(Access, Elem)` that expresses the accesses (definitions and/or uses) to the variables or constants of an expression, while `Edge-Label` denotes the label of a control flow edge (`true`, `false`, `uncond`). Expression-use and lvalue definition edges are respectively represented by the facts

- `expuse(File-ID, Function, Node-From, Node-To)`
- `lvaldef(File-ID, Function, Node-From, Node-To)`

while links between syntax tree and control flow graph nodes are expressed by facts of type

- `st_cf_link(File-ID, Function, Edge-Type, Node-AST, Node-CFG)`

Interprocedural Edges

Interprocedural edges consist of *call* edges, between the call site and the entry of the called procedure, *parameter-binding* edges between actual and formal parameters and *return expression-use* edges between a return node and the node using the expression evaluated by the called function:

- `call(File-ID1, Calling-Fun, Call-Node, File-ID2, Called-Fun, 0)`
- `bind(File-ID1, Calling-Fun, Actual, File-ID2, Called-Fun, Formal)`
- `return_expuse(File-ID2, Called-Fun, Return-Node, File-ID1, Calling-Fun, Call-Site)`

Program Dependencies

The control dependencies are represented by facts of the type:

- `control(File-ID, Function, Node-From, Node-To, Edge-Label)`

where `Edge-Label` can be `true` or `false`. The data flow dependencies are represented by facts of the type:

- `flow(File-ID1, Function1, Node1, File-ID2, Function2, Node2)`

that model both intraprocedural and interprocedural dependencies.

Appendix B

Example CCG Fact Base

In the following the listing of the Prolog fact base produced by the CCG analyser for the sample C program of figure 3.3 is shown.

```
bind(1, main, 5, 1, double, 1).

call(1, main, 4, 1, double, 0).

control(1, double, 0, 3, true).
control(1, double, 0, 2, true).
control(1, double, 0, 1, true).
control(1, main, 3, 7, true).
control(1, main, 3, 4, true).
control(1, main, 0, 3, true).
control(1, main, 0, 2, true).
control(1, main, 0, 1, true).
control(1, main, 4, 5, true).


edge(1, double, 0, 1, true).
edge(1, double, 0, 4, false).
edge(1, double, 1, 2, uncond).
edge(1, double, 2, 3, uncond).
edge(1, double, 3, 4, uncond).
edge(1, main, 0, 1, true).
edge(1, main, 0, 8, false).
edge(1, main, 1, 2, uncond).
edge(1, main, 2, 3, uncond).
edge(1, main, 3, 4, true).
edge(1, main, 3, 8, false).
edge(1, main, 4, 5, true).
edge(1, main, 4, 6, false).
edge(1, main, 4, 7, uncond).
edge(1, main, 5, 6, uncond).
edge(1, main, 7, 3, uncond).

expuse(1, double, 2, 3).
expuse(1, main, 1, 2).

file('exccg.c', 1).

flow(1, double, 1, 1, double, 2).
flow(1, main, 2, 1, double, 2).
flow(1, double, 2, 1, double, 2).
flow(1, main, 2, 1, main, 3).
flow(1, double, 2, 1, main, 3).
flow(1, main, 1, 1, main, 7).
flow(1, main, 7, 1, main, 7).

id_decl(1, main, 14, obj_loc(1, main, sc([1], @), a)).
id_decl(1, main, 18, obj_loc(1, main, sc([1], @), a)).
id_decl(1, main, 21, obj_loc(1, '@external', sc([], @), double)).
id_decl(1, main, 23, obj_loc(1, main, sc([1], @), b)).
id_decl(1, main, 3, obj_loc(1, main, sc([1], @), b)).
id_decl(1, main, 9, obj_loc(1, main, sc([1], @), a)).
id_decl(1, double, 5, obj_loc(1, double, sc([], @), p)).


node(1, double, 0, entry, @, []).
node(1, double, 1, formal, @, [ex(def, p)]).
node(1, double, 2, expr, @, [ex(ref, p),ex(ref, *),ex(postdef, @)]).
node(1, double, 3, expr, @, [ex(ref, ’@constant’)]).
node(1, double, 4, end, @, []).
node(1, main, 0, entry, @, []).
node(1, main, 1, expr, @, [ex(ref, ’@constant’),ex(rval, @),ex(def, b)]).
node(1, main, 2, expr, @, [ex(ref, ’@constant’),ex(rval, @),ex(def, a)]).
node(1, main, 3, expr, @, [ex(ref, ’@constant’),ex(ref, a)]).
node(1, main, 4, call, double, []).
node(1, main, 5, expr, @, [ex(address, a)]).
node(1, main, 6, end_params, @, []).
node(1, main, 7, expr, @,
                    [ex(ref, double),ex(rval, @),ex(ref, b),ex(def, @)]).
node(1, main, 8, end, @, []).

object(1, ’@external’, sc([], @), @, main, [’@fun’]).
object(1, ’@external’, sc([], @), int, double, [’@fun’]).
object(1, double, sc([], @), int, p, [’@pointer’]).
object(1, main, sc([1], @), int, a, []).
object(1, main, sc([1], @), int, b, []).

return_expuse(1, double, 3, 1, main, 7).

scope(1, double, [1], 8).
scope(1, main, [1], 26).

st_cf_link(1, double, expr_stat, 5, 2).
st_cf_link(1, double, expr_stat, 6, 3).
st_cf_link(1, double, group_begin, 8, 2).
st_cf_link(1, main, expr_stat, 10, 2).
st_cf_link(1, main, expr_stat, 15, 3).
st_cf_link(1, main, expr_stat, 18, 5).
st_cf_link(1, main, expr_stat, 20, 4).
st_cf_link(1, main, expr_stat, 24, 7).
st_cf_link(1, main, expr_stat, 4, 1).
st_cf_link(1, main, group_begin, 26, 1).
st_cf_link(1, main, group_exit, 26, 3).
st_cf_link(1, main, while_body, 25, 4).
st_cf_link(1, main, while_continue, 25, 7).
st_cf_link(1, main, while_exit, 25, 3).
st_cf_link(1, main, while_test, 25, 3).

st_link(1, double, compound, 8, 7).
st_link(1, double, in, 5, 3).
st_link(1, double, left, 6, 5).
st_link(1, double, op, 3, 4).
st_link(1, double, ret_expr, 7, 6).
st_link(1, double, right, 6, 1).
st_link(1, double, value, 1, 0).
st_link(1, main, act_par, 20, 18).
st_link(1, main, body, 25, 24).
st_link(1, main, compound, 26, 10).
st_link(1, main, compound, 26, 25).
st_link(1, main, condition, 25, 15).
st_link(1, main, in, 20, 21).
st_link(1, main, left, 10, 9).
st_link(1, main, left, 15, 14).
st_link(1, main, left, 24, 23).
st_link(1, main, left, 4, 3).
st_link(1, main, left, 7, 6).
st_link(1, main, op, 18, 19).
st_link(1, main, right, 10, 7).
st_link(1, main, right, 15, 12).
st_link(1, main, right, 24, 20).
st_link(1, main, right, 4, 1).
st_link(1, main, right, 7, 4).
st_link(1, main, value, 1, 0).
st_link(1, main, value, 12, 11).
st_link(1, main, value, 6, 5).

st_node(1, double, 2, 0).
st_node(1, double, const_int, 1).
st_node(1, double, group, 8).
st_node(1, double, id, 5).
st_node(1, double, mul, 6).
st_node(1, double, pointer, 3).
st_node(1, double, postadd, 4).
st_node(1, double, return, 7).
st_node(1, main, 0, 0).
st_node(1, main, 1, 5).
st_node(1, main, 10, 11).
st_node(1, main, add, 7).
st_node(1, main, addass, 24).
st_node(1, main, address_of, 19).
st_node(1, main, assign, 10).
st_node(1, main, assign, 4).
st_node(1, main, const_int, 1).
st_node(1, main, const_int, 12).
st_node(1, main, const_int, 6).
st_node(1, main, fun_call, 20).
st_node(1, main, group, 26).
st_node(1, main, id, 14).
st_node(1, main, id, 18).
st_node(1, main, id, 21).
st_node(1, main, id, 23).
st_node(1, main, id, 3).
st_node(1, main, id, 9).
st_node(1, main, leq, 15).
st_node(1, main, while, 25).
```

Appendix C

The Prolog Slicing Program

In the following the Prolog implementation of the program slicing algorithm is given.

```
slice(Fid, FName, NodeFrom, NodeTo) :-
    node(Fid, FName, NodeTo, _, _, _),
    find_path(Fid, FName, NodeFrom, NodeTo, Pathlist),
    find_call_chains(Pathlist, Funlist),
    traverse(Fid, FName, NodeFrom, NodeTo, Pathlist, Funlist,
             [node(Fid, FName, NodeTo)], [], Result),
    length(Result, Int),
    write_list_with_header(Result, 'Slice :- '),
    nl, writef('%w nodes in the slice', [Int]).



find_path(File, Fun, NodeFrom, NodeTo, Pathlist) :-
    node(Fid, FName, NodeTo, _, _, _),
    back_path(Fid, FName, NodeFrom, NodeTo, [node(Fid, FName, NodeTo)],
             [], Pathlist).



back_path(_, _, _, _, [], Pathlist, Pathlist).

back_path(Fid, FName, NodeFrom, NodeTo, [node(Fid, FName, NodeFrom)|Rest],
          PathIn, PathOut) :-
    not(member(node(Fid, FName, NodeFrom), PathIn)), !,
    back_path(Fid, FName, NodeFrom, NodeTo, Rest,
             [node(Fid, FName, NodeFrom)|PathIn], PathOut).
```

```
back_path(Fid, FName, NodeFrom, NodeTo, [node(Fid, FName, No)|Rest],
          PathIn, PathOut) :-
    diff(No, NodeFrom),
    not(member(node(Fid, FName, No), PathIn)), !,
    findall(node(Fid, FName, No1),
            (edge(Fid, FName, No1, No, _)),
            PredList),
    append(PredList, Rest, NextSet),
    list_to_set(NextSet, Open),
    back_path(Fid, FName, NodeFrom, NodeTo, Open,
              [node(Fid, FName, No)|PathIn], PathOut).

back_path(Fid, FName, NodeFrom, NodeTo, [_| Rest],
          PathIn, PathOut) :-
    back_path(Fid, FName, NodeFrom, NodeTo, Rest, PathIn, PathOut).


traverse(_, _, _, _, _, _, [], Slice, Slice).

traverse(Fid, FName, NodeFrom, NodeTo, Pathlist, Funlist,
         [node(Fid, FName, NodeFrom)|Rest], SliceIn, SliceOut) :-
    not(member(node(Fid, FName, NodeFrom), SliceIn)), !,
    traverse(Fid, FName, NodeFrom, NodeTo, Pathlist, FunList,
             Rest, [node(Fid, FName, NodeFrom)|SliceIn], SliceOut).

traverse(Fid, FName, NodeFrom, NodeTo, Pathlist, Funlist,
         [node(Fid, FName, No)|Rest], SliceIn, SliceOut) :-
    not(member(node(Fid, FName, No), SliceIn)),
    diff(No, NodeFrom),
    member(node(Fid, FName, No), Pathlist), !,
    backwards_trav(node(Fid, FName, No), NextSet),
    append(NextSet, Rest, ListOpen),
    list_to_set(ListOpen, Open),
    traverse(Fid, FName, NodeFrom, NodeTo, Pathlist, Funlist,
             Open, [node(Fid, FName, No)|SliceIn], SliceOut).
```

```
traverse(Fid, FName, NodeFrom, NodeTo, Pathlist, Funlist,
          [node(Fid1, FName1, No)|Rest], SliceIn, SliceOut) :-
    diff(FName, FName1),
    not(member(node(Fid1, FName1, No), SliceIn)),
    member(fun(Fid1, FName1), Funlist), !,
    backwards_trav(node(Fid1, FName1, No), NextSet),
    append(NextSet, Rest, ListOpen),
    list_to_set(ListOpen, Open),
    traverse(Fid, FName, NodeFrom, NodeTo, Pathlist, Funlist,
             Open, [node(Fid1, FName1, No)|SliceIn], SliceOut).

traverse(Fid, FName, NodeFrom, NodeTo, Pathlist, Funlist,
          [_| Rest], SliceIn, SliceOut) :-
    traverse(Fid, FName, NodeFrom, NodeTo, Pathlist, Funlist,
             Rest, SliceIn, SliceOut).


backwards_trav(node(Fid, FName, No), NextNodes) :-
    findall(node(Fid2, FName2, No2),
            (flow(Fid2, FName2, No2, Fid, FName, No);
             bind(Fid2, FName2, No2, Fid, FName, No);
             return_expuse(Fid2, FName2, No2, Fid, FName, No);
             call(Fid2, FName2, No2, Fid, FName, No)),
            InterNodes),
    findall(node(Fid, FName, No3),
            (control(Fid, FName, No3, No, _);
             expuse(Fid, FName, No3, No);
             lvaldef(Fid, FName, No3, No)),
            IntraNodes),
    append(InterNodes, IntraNodes, NodeList),
    list_to_set(NodeList, NextNodes).


find_call_chains([], []).
find_call_chains([node(Fid, FName, No)|Pathlist], Funlist) :-
   findall(fun(Fid2, FName2),
          (call(Fid, FName, CallNode, Fid2, FName2, 0),
           member(node(Fid, FName, CallNode), Pathlist)),
           Startset),
   call_chain(Startset, [], Funlist).


call_chain([], Funlist, Funlist).

call_chain([fun(Fid, FName)|Rest], FunIn, FunOut) :-
   not(member(fun(Fid, FName), FunIn)), !,
   findall(fun(Fid1, FName1),
          (call(Fid, FName, CallNode, Fid1, FName1, 0)),
          Calledlist),
   append(Calledlist, Rest, Nextset),
   list_to_set(Nextset, Open),
   call_chain(Open, [fun(Fid1, FName1)|FunIn], FunOut).

call_chain([_|Rest], FunIn, FunOut) :-
   call_chain(Rest, FunIn, FunOut).
```

Bibliography

[1] F. Abbattista, F. Lanubile, and G. Visaggio, "Recovering conceptual data models is human-intensive", in *Proceedings of the 5th International Conference on Software Engineering and Knowledge Engineering*, S. Francisco, California, U.S.A., 1993.

[2] S.K. Abd-El-Hafiz, V.R. Basili, and G. Caldiera, "Towards automated support for extraction of reusable components", in *Proceedings of the International Conference on Software Maintenance*, Sorrento, Italy, IEEE Comp. Soc. Press, 1991, pp. 212-219.

[3] S.K. Abd-El-Hafiz and V.R. Basili, "Documenting programs using a library of tree structured plans", in *Proceedings of the International Conference on Software Maintenance*, Montreal, Quebec, Canada, IEEE Comp. Soc. Press, 1993, pp. 152-161.

[4] S.K. Abd-El-Hafiz and V.R. Basili, "A tool for assisting the understanding and the formal development of software", in *Proceedings of the 6th International Conference on Software Engineering and Knowledge Engineering*, Jurmala, Latvia, 1994, pp. 36-45.

[5] A.V. Aho, R. Sethi, and J.D. Ullmann, *Compilers: Principles, Techniques and Tools*, Reading, MA: Addison-Wesley, 1986.

[6] P.B. Andrews, "Theorem proving via general matings", *Journal of the ACM*, vol. 28, no. 2, 1981, pp. 193-214.

[7] R.S. Arnold and W.B. Frakes, "Software reuse and reengineering", *CASE Trends*, vol. 4, no. 2, 1992, pp. 44-48.

[8] B.B. Baker, "An algorithm for structuring flowgraphs", *Journal of the ACM*, vol. 24, 1977, pp. 98-120.

[9] B.S. Baker, "On finding duplication and near-duplication in large software systems", in *Proceedings of the 2nd Working Conference on Reverse Engineering*, Toronto, Ontario, Canada, IEEE Comp. Soc. Press, 1995, pp. 86-95.

[10] V.R. Basili and H.D. Mills, "Understanding and documenting programs", *IEEE Transactions on Software Engineering*, vol. SE-8, no. 3, 1982, pp. 270-283.

[11] V.R. Basili, "Viewing maintenance as reuse-oriented software developmemnt", *IEEE Software*, vol. 7, no. 1, Jn. 1990, pp. 19-26.

[12] S. Basu and J. Misra, "Proving loop programs", *IEEE Transactions on Software Engineering*, vol. SE-1, no. 1, Mar. 1975, pp. 76-86.

[13] P. Benedusi, A. Cimitile, and U. De Carlini, "Reverse engineering, design document production, and structure charts", *The Journal of Systems and Software*, vol. 19, 1992, pp. 225-245.

[14] K.H. Bennett, B.J. Cornelius, M. Munro, and D.J. Robson, "Software Maintenance", in *Software Engineering Reference Book*, J. McDermid (ed), Butterworth-Heinemann, 1991.

[15] K.H. Bennett, "An introduction to software maintenance", in *Proceedings of the Summer School on Engineering of Existing Software*, Monopoli, Bari, Italy, Laterza publisher, 1995, pp. 31-67.

[16] K.H. Bennett, "Legacy systems: coping with success", *IEEE Software*, vol. 12, no. 1, Jan. 1995, pp. 19-23. Introduction to the special issue on "Legacy Systems".

[17] T.J. Biggerstaff and A.J. Perlis (eds), *Software Reusability*, voll. I and II, ACM Press and Addison Wesley, 1989.

[18] T.J. Biggerstaff, B.G. Mitbander, and D. Webster, "Program understanding and the concept assignment problem", *Communications of the ACM*, vol. 37, no. 5, May 1994, pp. 72-83.

[19] S. Blazy and P. Facon, "Partial Evaluation as an Aid to the Comprehension of Fortran Programs", in *Proceedings of the 2nd Workshop on Program Comprehension*, Capri, Italy, IEEE Comp. Soc. Press, 1993, pp. 46-54.

[20] T. Bodhuin, "An interaction paradigm for impact analysis", M.Sc. Thesis, 1995, Dep. of Computer Science, University of Durham, U.K..

[21] B.W. Boehm, "Software Engineering", *IEEE Transactions on Computer*, vol. 25, no. 2, 1976, pp. 1226-1241.

[22] B.W. Boehm, "Improving software productivity", *IEEE Software*, vol. 4, 1987, pp. 43-57.

[23] C. Bohm and G. Jacopini, "Flow diagrams, Turing machines and languages with only two formation rules", *Communications of the ACM*, vol. 9, 1966, pp. 366-371.

[24] G. Booch and D. Bryan, *Software Engineering with Ada*, The Benjamin/Cummings Publishing Company, Inc., third ed., Redwood City, California, U.S.A., 1994.

[25] F. Bott and M. Ratcliffe, "Reuse and design", in *Software Reuse and Reverse Engineering*, P.A.V. Hall (ed.), Chapman & Hall, London, pp. 35-51.

[26] R.S. Boyer, B. Elspas, and K.N. Levitt, "SELECT – a formal system for testing and debugging programs by symbolic execution", in *Proceedings of the International Conference on Reliable Software*, 1975, pp. 234-244.

[27] J.P. Bowen and M.G. Hinchey, "Ten commandments of formal methods", *IEEE Computer*, vol. 28, no. 4, Apr. 1995, pp. 56-63.

[28] J.P. Bowen and M.G. Hinchey, "Seven more myths of formal methods", *IEEE Software*, vol. 12, no. 4, July 1995, pp. 34-40.

[29] R.S. Boyer and J.S. Moore, *A Computational Logic*, Academic Press, New York, 1979.

[30] T. Bünter, "PERPLEX: an extensible tool architecture for C source code", Tech. rep. Computer Science 6/93, 1993, School of Engineering and Computer Science, University of Durham, U.K..

[31] J.M. Buxton, P. Naur, and B. Randell, Report on the NATO Software Engineering Conference, "Garmish, 1968", in *Software Engineering Concepts and Techniques*, J.M. Buxton, P. Naur, and B. Randell (eds), Petrocelli/Charter, New York, 1976. *1968 NATO Conference on Software Engineering.*

[32] G. Caldiera and V.R. Basili, "Identifying and qualifying reusable software components", *IEEE Computer*, vol. 24, no. 2, Feb. 1991, pp. 61-70.

[33] D. Callahan, "The program summary graph and flow-sensitive interprocedural data flow analysis", in *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation*, Atlanta, Georgia, U.S.A., ACM Press, 1988, pp. 47-56.

[34] F.W. Calliss, "Inter-module code analysis for software maintenance", Ph.D. Thesis, 1989, School of Engineering and Applied Science - Computer Science, University of Durham, U.K..

[35] G. Canfora, A. Cimitile, and U. De Carlini, "A logic-based approach to reverse engineering tools production", *IEEE Transactions on Software Engineering*, vol. 18, no. 12, Dec. 1992, pp. 1053-1064.

[36] G. Canfora and A. Cimitile, "An approach to reuse reengineering of existing software", in *Proceedings of Workshop on Software Evolution*, Bari, Italy, Fratelli Laterza (Pub.), 1992, pp. 73-85.

[37] G. Canfora, A. Cimitile, and M. Munro, "A reverse engineering method for identifying reusable abstract data types", in *Proceedings of Working Conference on Reverse Engineering*, Baltimore, Maryland, IEEE Comp. Soc. Press, 1993, pp. 73-82.

[38] G. Canfora, A. Cimitile, and M. Munro, "An improved algorithm for identifying reusable objects in code", Tech. Rep. 1993, School of Engineering and Computer Science, University of Durham, U.K.. To appear in *Software – Practice and Experience.*

[39] G. Canfora, A. Cimitile, M. Munro, and C.J. Taylor, "Extracting abstract data types from C programs: a case study", in *Proceedings of the International Conference on Software Maintenance*, Montreal, Quebec, Canada, IEEE Comp. Soc. Press, 1993, pp. 200-209.

[40] G. Canfora, A. Cimitile, M. Munro, and M. Tortorella, "Experiments in identifying reusable abstract data types in Program Code", in *Proceedings of the 2nd Workshop on Program Comprehension*, Capri, Italy, IEEE Comp. Soc. Press, 1993, pp. 36-45.

[41] G. Canfora, A. Cimitile, A. De Lucia, and G.A. Di Lucca, "Software salvaging based on conditions", in *Proceedings of the International Conference on Software Maintenance*, Victoria, British Columbia, Canada, IEEE Comp. Soc. Press, 1994, pp. 424-433.

[42] G. Canfora, A. Cimitile, and M. Munro, "RE^2: reverse engineering and reuse reengineering", *Journal of Software Maintenance: Research and Practice*, vol. 6, no. 2, 1994, pp. 53-72.

[43] G. Canfora, A. Cimitile, M. Munro, and M. Tortorella, "A precise method for identifying reusable data types in code", in *Proceedings of the International Conference on Software Maintenance*, Victoria, British Columbia, Canada, IEEE Comp. Soc. Press, 1994, pp. 216-225.

[44] G. Canfora, A. Cimitile, and G. Visaggio, "Assessing modularization and code scavenging techniques", *Journal of Software Maintenance: Research and Practice*, vol. 7, no. 5, 1995, pp. 317-331.

[45] G. Canfora, A. De Lucia, G.A. Di Lucca, and A.R. Fasolino, "Slicing large programs to isolate reusable functions", in *Proceedings of the EUROMICRO Conference*, Liverpool, U.K., IEEE Comp. Soc. Press, 1994, pp. 140-147.

[46] G. Canfora, A. De Lucia, G.A. Di Lucca, and A.R. Fasolino, "Recovering the architectural design for software comprehension", in *Proceedings of the 3rd Workshop on Program Comprehension*, Washington, D.C., U.S.A., IEEE Comp. Soc. Press, 1994, pp. 30-38.

[47] G. Canfora, G.A. Di Lucca, and M. Tortorella, "Recovering object classes and inheritance relationships fom existing code", in *Proceedings of the 6th International Conference on Software Engineering and Knowledge Engineering*, Jurmala, Latvia, 1994.

[48] G. Cantone, A. Cimitile, and U. De Carlini, "Well-formed conversion of unstructured one-in one-out schemes for complexity measurement and program maintenance", *The Computer Journal*, vol. 29, no. 4, 1986, pp. 322-329.

[49] T.E. Cheathaman, G.H. Holloway, and J.A. Townley, "Symbolic execution and the analysis of programs", *IEEE Transactions on Software Engineering*, vol. SE-5, no. 4, 1979, pp. 402-417.

[50] E.J. Chikofsky and J.H. Cross II, "Reverse engineering and design recovery: a taxonomy", *IEEE Software*, vol. 7, no. 1, Jan. 1990, pp. 13-17.

[51] A. Cimitile and U. De Carlini, "Reverse engineering: algorithms for program graph production", *Software – Practice and Experience*, vol. 21, no. 5, May 1991, pp. 519-537.

[52] A. Cimitile, "Towards reuse re-engineering of old software", in *Proceedings of the 4th International Conference on Software Engineering and Knowledge Engineering*, Capri, Italy, IEEE Comp. Soc. Press, 1992, pp. 140-149.

[53] A. Cimitile, A.R. Fasolino, and P. Maresca, "Reuse-reengineering and validation via concept assignment", in *Proceedings of the International Conference on Software Maintenance*, Montreal, Quebec, Canada, IEEE Comp. Soc. Press, 1993, pp. 216-225.

[54] A. Cimitile, M. Munro, and M. Tortorella, "Program comprehension trough the identification of abstract data types", in *Proceedings of the 2nd Workshop on Program Comprehension*, Washington, D.C., U.S.A., IEEE Comp. Soc. Press, 1994, pp. 12-19.

[55] A. Cimitile and G. Visaggio, "Software salvaging and the call dominance tree", *The Journal of Systems and Software*, vol. 28, no. 2, Feb. 1995, pp. 117-127.

[56] A. Cimitile and A. De Lucia, "Existing software reuse through the isolation of functional abstractions in code", Tech. Rep. PF-CNR "SICP" Sp6R119, 1995, Dep. of "Informatica e Sistemistica", University of Naples, Italy. Submitted for publication.

[57] A. Cimitile, A. De Lucia, and M. Munro, "Qualifying reusable functions using symbolic execution", in *Proceedings of the 2nd Working Conference on Reverse Engineering*, Toronto, Ontario, Canada, IEEE Comp. Soc. Press, 1995, pp. 178-187.

[58] A. Cimitile, A. De Lucia, and M. Munro, "Identifying reusable functions using specification driven program slicing: a case study", in *Proceedings of the International Conference on Software Maintenance*, Nice, France, IEEE Comp. Soc. Press, 1995, pp. 124-133.

[59] A. Cimitile, A. De Lucia, and M. Munro, "A specification driven slicing process for identifying reusable functions", Tech. Rep. 3/95, 1995, Dep. of Computer Science, University of Durham, U.K.. To appear in *Journal of Software Maintenance: Research and Practice.*

[60] L.A Clarke and D.J. Richardson, "Symbolic evaluation methods - implementations and applications", in *Computer Program Testing*, B. Chandrasekaran, and S. Radicchi (eds.), Amsterdam: North-Holland, 1981, pp. 65-102.

[61] A. Coen-Porisini, F. De Paoli, C. Ghezzi, and D. Mandrioli, "Software specialization via symbolic execution", *IEEE Transactions on Software Engineering*, vol. 17, no. 9, Sept. 1991, pp. 884-899.

[62] G. Colman, P. M. Andreae, and L. Groves, "Program analysis by symbolic execution and generalization", in *The Unified Computation Laboratory*, C.M.I. Rattray and R,G. Clark (eds.), Oxford University Press, 1991.

[63] K. Cooper and K. Kennedy, "Fast interprocedural alias analysis", in *Proceedings of the 16th Annual ACM Symposioum on Principles of Programming Languages*, Austin, Texas, U.S.A., ACM Press, 1989, pp. 49-59.

[64] P.D. Coward, "Symbolic execution systems – a review", *Software Engineering Journal*, vol. 3, no. 6, Nov. 1988, pp. 229-239.

[65] F. Cutillo, P. Fiore, and G. Visaggio, "Identification and extraction of domain independent components in large programs", in *Proceedings of the 1st Working Conference on Reverse Engineering*, Baltimore, Maryland, U.S.A., IEEE Comp. Soc. Press, 1993, pp. 83-91.

[66] F. Cutillo, F. Lanubile, and G. Visaggio, "Extracting application domain functions from old code: a real experience", in *Proceedings of 2nd Workshop on Program Comprehension*, Capri, Italy, IEEE Comp. Soc. Press, 1993, pp. 186-192.

[67] O.J. Dahl, E.W. Dijkstra, C.A.R. Hoare, *Structured Programming*, Academic Press, London, 1972.

[68] R.B. Dannenberg and G.W. Ernst, "Formal program verification using symbolic execution", *IEEE Transactions on Software Engineering*, vol. SE-8, no. 1, Jan. 1982, pp. 43-52.

[69] U. De Carlini, A. De Lucia, G.A. Di Lucca, and G. Tortora, "An integrated and interactive reverse engineering environment for existing software comprehension", in *Proceedings of 2nd Workshop on Program Comprehension*, Capri, Italy, IEEE Comp. Soc. Press, 1993, pp. 128-137.

[70] A. Delis, "Data binding tool: a tool for measurement based on data and type binding", Ph.D. Thesis, University of Maryland, U.S.A., 1990.

[71] A. De Lucia, A. Imperatore, M. Napoli, G. Tortora, and M. Tucci, "The software development workbench WSDW", in *Proceedings of the 4th International Conference on Software Engineering and Knowledge Engineering*, Capri, Italy, IEEE Comp. Soc. Press, 1992, pp. 213-221.

[72] A. De Lucia, G.A. Di Lucca, and A.R. Fasolino, "Towards the evaluation of reengineering effort to reuse existing software", in *Proceedings of 2nd International Conference on Achieving Quality in Software*, Venice, Italy, 1993, pp. 333-345.

[73] A. De Lucia, M. Napoli, G. Tortora, and M. Tucci, "The tool development language TDL for the software development environment WSDW", in *Proceedings of the 5th International Conference on Software Engineering and Knowledge Engineering*, S. Francisco, California, U.S.A., 1993, pp. 421-428.

[74] A. De Lucia, C. Di Cristo, G. Tortora, and M. Tucci, "Program parallelization in WSDW", in *Proceedings of the 6th International Conference on Software Engineering and Knowledge Engineering*, Jurmala, Latvia, 1994, pp. 462-469.

[75] A. De Lucia and M. Munro, "Program comprehension in a reuse reengineering environment", in *Proceedings of the First U.K. Workshop on Program Comprehension*, Durham, U.K., 1995. Also, Tech. Rep. 5/95, 1995, Dep. of Computer Science, University of Durham, U.K..

[76] M.F. Dunn and J.C. Knight, "Automating the detection of reusable parts in existing software", in *Proceedings of the 15th International Conference on Software Engineering*, Baltimore, Maryland, U.S.A., IEEE Comp. Soc. Press, 1993, pp. 381-390.

[77] H. Ehrig, H.J. Kreowski, A. Maggiolo Schettini, B.K. Rosen, J. Winkowski, "Transformations of structures: an algebraic approach", *Math. Syst. Theory*, vol. 14, 1981, pp. 305-334.

[78] H.A. Ellozy, "The determination of loop invariants for programs with arrays", *IEEE Transactions on Software Engineering*, vol. SE-7, no. 2, 1981, pp. 197-206.

[79] N.E. Fenton and W. Witty, "Axiomatic approach to software metrication through program decomposition", *The Computer Journal*, vol. 29, no. 4, 1986, pp. 330-339.

[80] N.E. Fenton, *Software metrics: a rigorous approach*, Chapman & Hall, London, 1991.

[81] J. Ferrante, K.J. Ottenstain, and J. Warren, "The program dependence graph and its use in optimization", *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, July 1987, pp. 319-349.

[82] R.W. Floyd, "Assigning meaning to programs", in *Proceedings of Symposioum on Applied Mathematics*, vol. 19, J.T. Schwartz (ed.), Amer. Math. Society, 1967, pp. 19-32.

[83] W.B. Frakes and S. Isoda, "Success factors of systematic reuse", *IEEE Software*, vol. 11, no. 5, Sep. 1994, pp. 15-19. Introduction to the special issue on "Systematic Reuse".

[84] P. Freeman(ed), *Tutorial on Software Reusability*, IEEE Comp. Soc. Press, New York, 1987.

[85] H. Gall and R. Klösch, "Finding objects in procedural programs: an alternative approach", in *Proceedings of the 2nd Working Conference on Reverse Engineering*, Toronto, Ontario, Canada, IEEE Comp. Soc. Press, 1995, pp. 208-216.

[86] K.B. Gallagher and J.R. Lyle, "Using program slicing in software maintenance", *IEEE Transactions on Software Engineering*, vol. 17, no. 8, August 1991, pp. 751-761.

[87] G.C. Gannod and B.H.C. Cheng, "Strongest postcondition semantics as the formal basis for reverse engineering", in *Proceedings of the 2nd Working Conference on Reverse Engineering*, Toronto, Ontario, Canada, IEEE Comp. Soc. Press, 1995, pp. 188-197.

[88] M. German and B. Wegbreit, "A synthesizer of inductive assertions", *IEEE Transactions on Software Engineering*, vol. SE-1, no. 1, Mar. 1975, pp. 68-75.

[89] C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of Software Engineering*, Prentice Hall Pub., 1991.

[90] R. Gupta, M.J. Harrold, and M.L. Soffa, "An approach to regression testing using slicing", in *Proceedings of the Conference on Software Maintenance*, Orlando, Florida, U.S.A., IEEE Comp. Soc. Press, 1992, pp. 299-308.

[91] P.A.V. Hall, "Software reuse, reverse engineering and reengineering", in *Software Reuse and Reverse Engineering*, P.A.V. Hall (ed.), Chapman & Hall, London, pp. 3-31.

[92] R.J. Hall, "Automatic extraction of executable program subsets by simultaneous program slicing", *Journal of Automated Software Engineering*, vol. 2, no. 1, Mar. 1995, pp. 33-53.

[93] M.H. Halstead, *Elements of Software Science*, North-Holland, Amsterdam, 1977.

[94] S.L. Hantler and J.C. King, "An introduction to proving the correctness of programs", *Computing Surveys*, vol. 8, Sept. 1976, pp. 331-353.

[95] M. Harman, S. Danic and Y. Sivagurunathan, "Program comprehension assisted by slicing and transformation", in *Proceedings of the First U.K. Workshop on Program Comprehension*, Durham, U.K., 1995.

[96] M.J. Harrold and M.L. Soffa, "Computation of interprocedural definition and use dependencies", in *Proceedings of IEEE International Conference on Computer Languages*, New Orleans, Louisiana, U.S.A., IEEE Comp. Soc. Press, 1990, pp. 297-306.

[97] M.J. Harrold and M.L. Soffa, "Selecting data-flow integration testing", *IEEE Software*, vol 8, no. 2, Mar. 1991, pp. 58-65.

[98] M.J. Harrold and B. Malloy, "A unified interprocedural program representation for a maintenance environment", *IEEE Transactions on Software Engineering*, vol. 19, no. 6, June 1993, pp. 584-593.

[99] H.P. Haughton and K. Lano, "Object revisited", in *Proceedings of the International Conference on Software Maintenance*, Sorrento, Italy, IEEE Comp. Soc. Press, 1991, pp. 152-161.

[100] M.S. Hecht, *Flow Analysis of Computer Programs*, Elsevier North-Holland, New York, 1977.

[101] C.A.R. Hoare, "An axiomatic basis for computer programming", *Communications of the ACM*, vol. 12, no. 10, Oct. 1969, pp. 576-580.

[102] S. Horwitz, P. Pfeiffer, and T. Reps, "Dependence analysis for pointer variables", *ACM SIGPLAN Notices*, vol. 24, no. 7, 1989, pp. 28-40.

[103] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs", *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 1, Jan. 1990, pp. 26-60.

[104] S. Horwitz, "Identifying the semantic and textual differences between two versions of a program", in *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation*, ACM Press, 1990, pp. 234-245.

[105] D.H. Hutchens and V.R. Basili, "System structure analysis: clustering with data binding", *IEEE Transactions on Software Engineering*, vol. SE-11, no. 8, Aug. 1985, pp. 749-757.

[106] J. Jiang, X. Zhou, and D.J. Robson, "Program slicing for C - the problems in implementation", in *Proceedings of the International Conference on Software Maintenance*, Sorrento, Italy, IEEE Comp. Soc. Press, 1991, pp. 182-190.

[107] S.C. Johnson, "YACC: yet another compiler-compiler", Tech. Rep. no. 32, Bell Laboratories, Murray Hills, New Jersey, U.S.A., 1975. Also in *UNIX Programmers' Guide*.

[108] C.B. Jones, *Systematic Software Development Using VDM*, Prentice Hall International, 1990.

[109] S. Katz and Z. Manna, "Logical analysis of programs", *Communications of the ACM*, vol. 19, no. 4, Apr. 1976, pp. 188-206.

[110] B. Kernighan and D. Ritchie, *The C Programming Language*, Englewood Cliffs, New Jersey: Prentice-Hall, second ed., 1988.

[111] J.C. King, "Symbolic execution and program testing", *Communications of the ACM*, vol. 19, no. 7, July 1976, pp. 385-394.

[112] D.A Kinloch and M. Munro, "A combined representation for the maintenance of C programs", in *Proceedings of the 2nd Workshop on Program Comprehension*, Capri, Italy, IEEE Comp. Soc. Press, 1993, pp. 119-127.

[113] D.A Kinloch and M. Munro, "Understanding C programs using the combined C graph Representation", in *Proceedings of the International Conference on Software Maintenance*, Victoria, British Columbia, Canada, IEEE Comp. Soc. Press, 1994, pp. 172-180.

[114] D.A. Kinloch, "A combined representation for the maintenance of C programs", Ph.D. Thesis, 1995, Dep. of Computer Science, University of Durham, U.K..

[115] K. Kontogiannis, R. De Mori, M. Bernstein, E. Merlo, "Localization of design concepts in legacy systems", in *Proceedings of the International Conference on Software Maintenance*, Victoria, British Columbia, Canada, IEEE Comp. Soc. Press, 1994, pp. 414-423.

[116] K. Kontogiannis, R. De Mori, M. Bernstein, M. Galler, and E. Merlo, "Pattern matching for design concept localization", in *Proceedings of the 2nd Working Conference on Reverse Engineering*, Toronto, Ontario, Canada, IEEE Comp. Soc. Press, 1995, pp. 96-103.

[117] B. Korel and J. Laski, "Dynamic program slicing", *Inform. Process. Lett.*, vol. 29, no. 3, Oct. 1988, pp. 155-163.

[118] B. Korel and J. Laski, "Dynamic slicing of computer programs", *The Journal of Systems and Software*, vol. 13, no. 3, Nov. 1990, pp. 187-195.

[119] W. Kozaczynski and N. Wilde, "On the reengineering of transactions systems", *Journal of Software Maintenance: Research and Practice*, vol. 4, no. 3, 1992, pp. 143-162.

[120] W. Kozaczynsky, J. Ning, and A. Engberts, "Program concept recognition and transformation", *IEEE Transactions on Software Engineering*, vol. 18, no. 12, Dec. 1992, pp. 1065-1075.

[121] W. Kozaczynsky and J. Ning, "Automated program understanding by concept recognition", *Automated Software Engineering*, vol. 1, no. 1, 1994, pp. 61-78.

[122] D.J. Kuck, *The Structure of Computers and Computations*, Wiley, New York, 1978.

[123] W. Landi and B.G. Ryder, "Pointer-induced aliasing: a problem classification", in *Proceedings of the 18th Annual ACM Symposioum on Principles of Programming Languages*, Orlando, Florida, U.S.A., ACM Press, 1991, pp. 93-103.

[124] F. Lanubile and G. Visaggio, "Function recovery based on program slicing", in *Proceedings of the International Conference on Software Maintenance*, Montreal, Quebec, Canada, IEEE Comp. Soc. Press, 1993, pp. 396-404.

[125] M.M. Lehman, "Program evolution", *Information processing management*, vol. 20, 1984, pp. 19-36.

[126] B.P. Lientz and E.B. Swanson, *Software Maintenance Management*, Addison-Wesley, Reading Massachussets, U.S.A., 1980.

[127] W.C. Lim, "Effects of reuse on quality, productivity, and economics", *IEEE Software*, vol. 11, no. 5, Sept. 1994, pp. 23-30.

[128] P.E. Livadas and P.K. Roy, "Program dependence analysis", in *Proceedings of the International Conference on Software Maintenance*, Orlando, Florida, U.S.A., IEEE Comp. Soc. Press, 1992, pp. 356-365.

[129] S.S. Liu and N. Wilde, "Identifying objects in a conventional procedural language an example of data design recovery", in *Proceedings of the International Conference on Software Maintenance*, San Diego, California, U.S.A., IEEE Comp. Soc. Press, 1990, pp. 266-271.

[130] A. Maggiolo Schettini, M. Napoli, G. Tortora, "Web structures: a tool for representing and manipulating programs", *IEEE Transactions on Software Engineering*, vol. 14, no. 11, Nov. 1988, pp. 1597-1609.

[131] E. Merlo, P.Y. Gagné, J.F. Girard, K. Kontogiannis, L. Hendren, P. Panangaden, R. De Mori, "Reengineering user interfaces", *IEEE Software*, vol. 12, no. 1, Jan. 1995, pp. 64-73.

[132] T.J. McCabe, "A complexity measure", *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, Dec. 1976, pp. 308-320.

[133] M.D. McIlroy, "Mass produced software components", in *Software Engineering Concepts and Techniques*, J.M. Buxton, P. Naur, and B. Randell (eds), Petrocelli/Charter, New York, 1976, pp. 88-98. Paper presented at the *1968 NATO Conference on Software Engineering*, Garmish.

[134] H. mili, F. Mili, and A. Mili, "Reusing Software: Issues and Research Directions", *IEEE Transactions on Software Engineering*, vol. 21, no. 6, June 1995, pp. 528-561.

[135] E.M. Myers, "A precise interprocedural data flow algorithm", in *Proceedings of the 8th Annual ACM Symposioum on Principles of Programming Languages*, ACM Press, 1981, pp. 219-230.

[136] J.Q. Ning, A. Engberts, and W. Kozaczynski, "Recovering reusable components from legacy systems by program segmentation", in *Proceedings of 1st Working Conference on Reverse Engineering*, Baltimore, Maryland, U.S.A., IEEE Comp. Soc. Press, 1993, pp. 64-72.

[137] L. Ott and J. Thuss, "The relationship between slices and module cohesion", in *Proceedings of the 11th International Conference on Software Engineering*, IEEE Comp. Soc. Press, 1989, pp. 198-204.

[138] K.J. Ottenstain and L.M. Ottenstain, "The program dependence graph in a software developmemnt environment", *Proceedings of the ACM SIGPLAN/SIGSOFT Symposioum on Practical Programming Development Environments*, Pittsburgh, PA, U.S.A., 1984, *ACM SIGPLAN Notices*, vol. 19, no. 5, May 1984, pp. 177-184, and *ACM SIGSOFT Software Engineering Notes*, vol. 9, no. 3.

[139] H.D. Pande, W.A. Landi, and B.G. Ryder, "Interprocedural def-use associations for C systems with single level pointers", *IEEE Transactions on Software Engineering*, vol. 20, no. 5, May 1994, pp. 385-403.

[140] D.L. Parnas, "On the criteria to be used in decomposing systems into modules", *Communications of the ACM*, vol. 15, 1972, pp. 1053-1058.

[141] D.L. Parnas and D.M. Weiss, "Active design reviews: principles and practices", *The Journal of Systems and Software*, vol. 7, no. 4, Dec. 1987, pp. 259-265.

[142] S. Paul and A. Prakash, "A framework for source code search using program patterns", *IEEE Transactions on Software Engineering*, vol. 20, no. 6, 1994, pp. 463-475.

[143] L.C. Paulson, "The foundation of a generic theorem prover", *Journal of Automatic Reasoning*, vol. 5, no. 3, 1982, pp. 363-398.

[144] R.E. Prather and S.G. Guilieri, "Decomposition of flowchart schemata", *The Computer Journal*, vol. 24, 1981, pp. 258-262.

[145] R. Prieto-Diaz, "Making software reuse work: an implementation model", *ACM SIGSOFT Software Engineering Notes*, vol. 16, no. 3, 1991, pp. 61-68.

[146] A. Quilici, "A memory-based approach to recognizing programming plans", *Communications of the ACM*, vol. 37, no. 5, May 1994, pp. 84-93.

[147] A. Quilici and D.N. Chin, "DECODE: a cooperative environment for reverse-engineering legacy software", in *Proceedings of the 2nd Working Conference on Reverse Engineering*, Toronto, Ontario, Canada, IEEE Comp. Soc. Press, 1995, pp. 156-165.

[148] C. Rich and R. Waters, *The Programmer's Apprentice*, Addison Wesley, Reading, MA, 1990.

[149] C. Rich and L.M. Wills, "Recognizing a program's design: a graph-parsing approach", *IEEE Software*, vol. 7, no. 1, 1990, pp. 82-89.

[150] D.J. Robson, K.H. Bennett, B.J. Cornelius, and M. Munro, "Approaches to program comprehension", *The Journal of Systems and Software*, vol. 14, no. 2, Feb. 1991, pp. 79-84.

[151] D. Ross, 1993, "Recognising programs using an effects-based analysis", Ph.D. Thesis, 1993, School of Computer Studies, University of Leeds, U.K..

[152] W.W. Royce, "Managing the development of large software systems: concepts and techniques", in *Proceedings of WesCon*, S. Francisco, California, U.S.A., August 1970.

[153] S. Rugaber, K. Stirewalt, and L.M. Wills, "The interleaving problem in program understanding", in *Proceedings of the 2nd Working Conference on Reverse Engineering*, Toronto, Ontario, Canada, IEEE Comp. Soc. Press, 1995, pp. 166-175.

[154] B.G. Ryder, "Constructing the call graph of a program", *IEEE Transactions on Software Engineering*, vol. SE-5, no. 3, May 1979, pp. 216-225.

[155] R.W. Scheifler and J. Gettys, "The X window system", *ACM Transactions on Graphics*, vol. 5, no. 2, 1986, pp. 79-109.

[156] R.W. Schwanke, "An intelligent tool for re-engineering software modularity", in *Proceedings of the 13th International Conference on Software Engineering*, IEEE Comp. Soc. Press, 1991, pp. 83-92.

[157] H.M. Sneed, "Migration of procedurally oriented COBOL programs in an object-oriented architecture", in *Proceedings of the International Conference on Software Maintenance*, Orlando, Florida, U.S.A., IEEE Comp. Soc. Press, 1993, pp. 396-404.

[158] H.M. Sneed and E. Nyary, "Downsizing large application programs", *Journal of Software Maintenance: Research and Practice*, vol. 6, no. 5, 1994, pp. 105-116.

[159] H.M. Sneed, "Planning the reengineering of legacy systems", *IEEE Software*, vol. 12, no. 1, Jan. 1995, pp. 24-34.

[160] E. Soloway and K. Erdlich, "Empirical studies of programming knowledge", *IEEE Transactions on Software Engineering*, vol. SE-10, no. 5, Sept. 1984, pp. 595-609.

[161] I. Sommerville, *Software Engineering*, Addison-Wesley, 1989.

[162] J.M. Spivey, *The Z Notation: A Reference Manual*, Prentice Hall International, 1992.

[163] T.A. Standish, "An essay on software reuse", *IEEE Transactions on Software Engineering*, vol. SE-10, no. 5, Sept. 1984, pp. 494-497.

[164] L. Sterling and E. Shapiro, *The Art of Prolog*, MIT Press, Cambridge, Massachussets, U.S.A., 1986.

[165] M. Tamir, "ADI: automatic derivation of invariants", *IEEE Transactions on Software Engineering*, vol. SE-6, no. 1, Jan. 1980, pp. 40-48.

[166] L. Torczan and K. Kennedy, "Efficient computation of flow insensitive interprocedural summary information", in *Proceedings of the ACM SIGPLAN Symposioum on Compiler Construction*, Montreal, Quebec, Canada, 1984, *ACM SIGPLAN Notices*, vol. 19, no. 6, June 1984, pp. 49-59.

[167] M. Tortorella, "Identifying reusable abstract data types in code", M.Sc. Thesis, 1995, Dep. of Computer Science, University of Durham, U.K..

[168] W. Tracz (ed), *Tutorial on Software Reuse: Emerging Technologies*, IEEE Comp. Soc. Press, New York, 1988.

[169] G.A. Venkatesh, "The semantic approach to program slicing", in *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation*, Toronto, Ontario, Canada, 1991, *ACM SIGPLAN Notices*, vol. 26, no. 6, 1991, pp. 107-119.

[170] R.J. Waldinger and K.N. Levitt, "Reasoning about programs", *Artificial Intelligence*, vol. 5, 1974, pp. 235-316.

[171] M.P. Ward, "Abstracting a specification from code", *Journal of Software Maintenance: Research and Practice*, vol. 5, no. 2, 1993, pp. 101-122.

[172] R.C. Waters, "A method for analyzing loop programs", *IEEE Transactions on Software Engineering*, vol. SE-5, no. 3, May 1979, pp. 237-247.

[173] B. Wegbreit, "The synthesis of loop predicates", *Communications of the ACM*, vol. 17, no. 2, Feb. 1974, pp. 102-112.

[174] M. Weiser, "Program slicing", in *Proceedings of the 5th International Conference on Software Engineering*, San Diego, California, U.S.A., 1981, pp. 439-449.

[175] M. Weiser, "Programmers use slices when debugging", *Communications of the ACM*, vol. 25, July 1982, pp. 446-452.

[176] M. Weiser, "Program slicing", *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, July 1984, pp. 352-357.

[177] N. Wilde, J.A. Gomez, T. Gust, and D. Strasburg, "Locating user functionality in old code", in *Proceedings of the International Conference on Software Maintenance*, Orlando, Florida, U.S.A., IEEE Comp. Soc. Press, 1992, pp. 200-205.

[178] N. Wilde and M.C. Scully, "Software reconnaissance: mapping program features to code", *Journal of Software Maintenance: Research and Practice*, vol. 7, no. 1, 1995, pp. 49-62.

[179] L. Wills, "Automated program recognition: a feasibility demonstration", *Artificial Intelligence*, vol. 45, no. 1-2, 1990, pp. 113-171.

[180] L. Wills, "Automated program recognition by graph parsing", Ph.D. Thesis, MIT, Cambridge, Massachussets, U.S.A., 1992.

[181] A.S. Yeh, D.R. Harris, and H.B. Rubenstein, "Recovering abstract data types and object instances from a conventional procedural language", in *Proceedings of the 2nd Working Conference on Reverse Engineering*, Toronto, Ontario, Canada, IEEE Comp. Soc. Press, 1995, pp. 227-236.

[182] E. Yourdon and L.L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and System Design*, Prentice Hall, Englewood Cliffs, New Jersey, U.S.A., 1979.

[183] -, "ANSI/IEEE standard no. 729-1983", in *Software Engineering Standard*, IEEE Comp. Soc. Press, 1983.

[184] -, "ANSI/IEEE standard no. 610.12-1990", in *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Comp. Soc. Press, 1991.