A Combined Representation for the Maintenance of C Programs

David A. Kinloch

Ph.D. Thesis

Centre for Software Maintenance
Department of Computer Science
University of Durham

July 1995

Abstract

A programmer wishing to make a change to a piece of code must first gain a full understanding of the behaviours and functionality involved. This process of program comprehension is difficult and time consuming, and often hindered by the absence of useful program documentation.

Where documentation is absent, static analysis techniques are often employed to gather programming level information in the form of data and control flow relationships, directly from the source code itself. Software maintenance environments are created by grouping together a number of different static analysis tools such as program slicers, call graph builders and data flow analysis tools, providing a maintainer with a selection of 'views' of the subject code. However, each analysis tool often requires its own intermediate program representation (IPR). For example, an environment comprising five tools may require five different IPRs, giving repetition of information and inefficient use of storage space.

A solution to this problem is to develop a single combined representation which contains all the program relationships required to present a maintainer with each required code view. The research presented in this thesis describes the Combined C Graph (CCG), a dependence-based representation for C programs from which a maintainer is able to construct data and control dependence views, interprocedural control flow views, program slices and ripple analyses. The CCG extends earlier dependence-based program representations, introducing language features such as expressions with embedded side effects and control flows, value returning functions, pointer variables, pointer parameters, array variables and structure variables. Algorithms for the construction of the CCG are described and the feasibility of the CCG demonstrated by means of a C/Prolog based prototype implementation.

Acknowledgements

The author would like to acknowledge the Engineering and Physical Sciences Research Council (EPSRC) for the award of a research studentship and the European Gas Turbines Ltd. Mechanical Engineering Centre (Whetstone) for their financial and scientific support through a Co-operative Award in Science and Engineering (CASE) studentship.

Special thanks are also due to my supervisor Malcolm Munro for all his help and guidance throughout the course of this research, and to Toni Bünter, Thierry Bodhuin and Gerardo Canfora for their help in the development of the prototype tool. Many thanks must also go to all my colleagues at the Centre for Software Maintenance and especially to Ishbel Duncan for her help and advice in the writing of this thesis. Finally huge thanks are owed to Jenny Newton for her support and encouragement over the past three years.

Statement of Copyright

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As computer hardware becomes more and more powerful, the size and complexity of applications demanded by users continues to increase. The late 1960s saw the beginning of the 'software crisis' - existing software development methods proved to be inadequate when applied to new large scale projects. Consequently systems were often late, over budget, unreliable, difficult to maintain and unsatisfactory to the users.

Over the past twenty-five years, the 'software crisis' has been eased to a limited extent through the use of *software engineering* techniques. Software engineering is defined by Fairley[24] as

> The technological and management discipline concerned with the systematic production and maintenance of software products that are developed and modified on time and within cost estimates.

Software process models such as the *waterfall model* have simplified management problems, whilst techniques such as structured programming, information hiding and formal development methods together with the use of tools and a greater emphasis on training and quality have led to more reliable software. However, the 'software crisis' is still with us and better techniques, tools and education are still required.

1.1 The Software Maintenance Problem

Despite improvements in software engineering processes, the need for *software maintenance*, the modification of a program after delivery, remains unavoidable. Software maintenance can be a major cost for any large organisation. Foster et al[30] provide an estimated figure showing 2% of a typical company's overall expenditure to be on software maintenance. Many other figures have

been produced showing the costs of maintenance during the lifetime of a piece of software. Parikh and Zvegintzov[69] indicate that software maintenance consumes 50% of all computer resources and research by Boehm[11] has shown that maintenance costs can be up to ten times those of initial development. Only recently have organisations begun to recognise software maintenance as a problem and yet it is in this area of the *software life cycle* that the greatest potential for savings exists.

Four categories of maintenance have been described.

- *Perfective maintenance* - improving the functionality of software in response to the user's defined changes.
- *Corrective maintenance* - correction of errors in the software.
- *Adaptive maintenance* - software alterations due to changes in environment.
- *Preventative maintenance* - updating the software to improve its future maintainability, without altering the functionality.

Of the four areas, studies by Lientz and Swanson[59] showed 64% of the costs involved to be in perfective maintenance and only 17% in corrective maintenance.

Many factors have been highlighted as contributing to the software maintenance problem. Lientz and Swanson identify six major causes.

- quality of documentation
- user demand for enhancements
- competing demands for maintenance personnel
- meeting schedules
- inadequate user training
- staff turnover

Sommerville[84] lists several other factors.

- is the application clearly defined or new?
- external environment
- hardware stability

- language and style of code
- quality of validation and testing

Maintenance activities can also render further maintenance more difficult as program structure becomes degraded. Software maintenance is generally perceived as having a low profile, with management often placing little emphasis on the activity. Few tools exist and the quantity of research into the subject, although increasing, is still small.

Some advances have been made in combating these problems. It is now recognised that programs must be designed and implemented in ways that will reduce maintenance problems. Yourdon[95] states that maintenance costs may be reduced by a factor of five through the use of structured analysis, design and programming. The use of quality procedures has become important, leading to traceability through the development life cycle. Research has also been carried out to devise quality metrics to assess code from a maintenance viewpoint. McCabe[65] and Halstead[37] have devised metrics assessing program complexity whilst Kafura and Reddy[49] use seven metrics to assess a program.

Each of these techniques addresses the maintenance problem from the development aspect of the software life cycle. Improvements in the software maintenance phase itself have come through the application of *process modelling* to software maintenance. Boehm[12], Patkau[70] and Foster[29] each describe models for software maintenance. *Reverse engineering* techniques in which representations of the system are created at higher levels of abstraction have also been widely applied, especially in connection with archaic software systems. Reverse engineering may be classified as *inverse engineering* or *design recovery*, allowing the recapture of information at the specification and design levels respectively. Examples of such systems are described by Ward et al[87], who presents an inverse engineering tool based on formal program transformations, and Antonini et al[6], who describe the creation of structure diagrams and design documents from COBOL code. Following a reverse engineering step, 'forward' engineering processes may then be used to produce a completely new 're-engineered' system. A simpler technique is that of *code restructuring*, in which existing systems are restructured producing code in the same language as the original.

Software maintenance is made more difficult because any changes made to the source code may induce unexpected *ripple effects* on other parts of the system. Work on measuring the stability of code against ripple effects has been reported by Yau and Collofello[94]. Ripple effects may also occur at the design and specification levels and may additionally involve system documentation. *Impact analysis*, the task of 'assessing the effects of making a set of changes to a software system', allows a

maintainer to estimate what is needed to accomplish a change or identify possible consequences. By employing impact analysis techniques a maintainer is able to derive a bounded change set, which represents a complete and consistent modification with no undesirable ripple effects. Starting with an initial set of impacted objects, impact analysis identifies iteratively additional candidate objects until no further candidates are found. Current research in impact analysis focusses on the development of automated tools, for example the 'Propagation Control System' developed at Matra Marconi[10][73].

1.2 Program Comprehension

Whenever a change is to be made to a piece of software, it is important that the maintainer gains a complete understanding of the behaviour and functionality involved. This process of *program comprehension* is frequently made more difficult by the absence of program documentation. The maintenance programmer will often not have been involved in the development process or a significant period of time may have elapsed between development and maintenance. Documentation will then become of crucial importance.

Documentation is often found to be out of date, incomplete, difficult to read, difficult to update or not suited to the needs of the programmer. In many cases, particularly with older programs, documentation simply does not exist. As pressure increases through tight development deadlines, 'non-essential' activities such as documentation are postponed or forgotten. Furthermore the variety of available documentation preparation systems means that often within the same project, different systems are used. At later stages updating such documentation can become impractical if the systems are no longer available or understood. It is therefore evident that in many cases the maintenance programmer's only reliable description of the software will be the source code. In this case the maintainer must attempt to comprehend the code itself.

Simple *code reading* can be an effective way of understanding a program but is obviously dependent on the size and complexity of the code. Well structured, consistently indented code with meaningful identifiers and useful comments will be easier to understand than a program with inconsistent style and meaningless names.

Various models have been developed describing the mental processes involved in program comprehension. Brooks[13] suggests comprehension involves the top-down construction of a mental model at different levels from the problem domain to the programming domain, together with the relationships between each domain. Letovsky[56] describes a programmer's mental model as a

series of layers between the specification and implementation. A programmer employs both top-down and bottom-up strategies, forming an 'inquiry' when the current mental model is found to be inadequate. In either case, a sound understanding of the system at the code level is a pre-requisite for successful software maintenance.

The complex nature of data and control dependencies between the modules and functions comprising a software system make the process of program comprehension a difficult task. It is not surprising that Standish[85] reports the time spent on program understanding to be between 50% and 90% of the total time spent on software maintenance. Automatic tools are required if this time is to be reduced and savings made.

An important class of automated maintenance tools are *static analysis tools*, which are able to analyse a program without its execution to extract information of use to a maintenance programmer. This information, in the form of cross references, control flows, data flows, call graphs or program slices, is often presented graphically. By integrating static analysis tools, software maintenance environments have been developed. The source code is analysed and the resulting information stored in a central database for later browsing by the maintainer.

Integrated maintenance environments are able to provide multiple *views* of a software system. The maintainer is presented with information in more than one form and can switch quickly from one view to another, often through the use of hypertext and multiple window environments. Views can also aid program comprehension by concentrating the maintainer's attention on relevant parts of the software. Maintenance environments often present information at different levels of abstraction. The maintainer can gain a general understanding of the program and can then examine parts of the code in closer detail.

An example of such a system is Chen et al's C Information Abstractor[18]. This tool collects information in a relational database which can be accessed by a maintainer. Cleveland[19][20] also describes a tool providing control flows, data flows, call graphs and cross references, together with a user interface based on multiple windows, to aid the comprehension of assembler code.

1.3 Research Problems

Whilst the software maintainer is primarily interested in the views and information provided by a software maintenance environment, and by the usability of its external interface, an important aspect of these tools is the intermediate program representation on which the tool is based. Where a maintenance environment is created simply by grouping a set of unconnected software maintenance

tools, such as a call graph builder, a cross referencer and a program slicer, each tool may require its own intermediate program representation. For example, a slicing algorithm will require the construction of a different representation than would a data flow analysis algorithm. A maintenance environment providing these two static analysis views would require two different intermediate program representations. The need for two (or more) representations will lead to repetition of program information and inefficient use of storage space.

Harrold and Malloy[38][39] first recognised this problem and proposed a single intermediate program representation containing sufficient information to create each of the views required by the software maintainer. This representation unifies the relationships and dependencies between the program components and allows savings to be made in the use of storage space. Any algorithm now only accesses one single program representation.

Harrold and Malloy describe the *Unified Interprocedural Graph* (UIG), a dependence-based intermediate program representation providing four program views - program slices, call graph, data flow and control flow. However, the language modelled by the UIG is restricted to only ordinal types, assignments, while loops, for statements, if statements and procedures with reference parameters.

Any software maintenance or program comprehension tool can only become of real use if it is practically applicable to large programs written in real programming languages. Such programs are the domain in which software maintenance and program comprehension become truly problematic and hence costly. The aim of this research is to extend the ideas of Harrold and Malloy to allow the modelling of programs written in the C programming language [51]. The C language presents a number of additional language constructs and features which are not part of the language considered by Harrold and Malloy. The primary features to be addressed in this thesis follow.

Value-returning functions Functions in the C language may or may not return a value to the caller function.

Embedded side effects Expressions in C may contain embedded side effects through the use of the postfix and prefix increment/decrement operators, assignments and function calls.

Embedded control flow Expressions in C may additionally contain embedded control flow due to the conditional expression operator or short-circuit evaluation of boolean expressions.

Pointers C permits arbitrary assignment of pointer variables and dynamic allocation of memory, leading to complex *aliasing* problems.

Structures The combination of pointers and structures permits the use of self-referential structures such as lists and trees.

Arrays There is a strong relationship between arrays and pointers - operations written using arrays can also be achieved using pointer variables.

Value parameters Parameters in C are passed by value. To achieve the effects of *call by reference* a programmer must make use of pointers as actual parameters.

Control structures C provides the `switch` control structure and allows the creation of unstructured programs through the use of `break`, `continue` and `goto`.

An intermediate program representation, the Combined C Graph (CCG), will be developed to represent programs involving these language features. Algorithms will be created to construct the following program views from the new intermediate representation.

Call graph A representation of the call relationships between the program's functions.

Control dependence A representation of control dependencies between the program's statements.

Definition-use pairs A representation of data dependencies between the program's statements.

Data flow information Information on the definition and use of global variables and 'pointer' parameters.

Program slices All statements and predicates of the program that might affect the value of a variable v at statement s.

The intermediate representation and view construction algorithms will each be demonstrated by means of a prototype implementation.

1.4 Criteria for Success

1. Different views of a C program are to be made available to a maintainer and these views should help the comprehension process. The interface presented to the maintainer should allow quick switching between views and should allow the maintainer to concentrate on areas of the program which are of particular interest. Views available should include call graphs, definition-use, data flow, control dependence and program slices. These views should be created quickly and should provide the maintainer with accurate and useful information.

2. The level of coverage of the C language that is provided by the representation. Of particular importance are features such as pointers, embedded side-effects, embedded control flows and value-returning functions.

3. The accuracy of the representation. Language features with dynamic effects, such as self-referential structures and arrays will require approximations in order to be modelled statically. These approximations must still provide the maintainer with useful information.

4. Practical application of any program comprehension tool requires that the tool be able to deal with large programs, since it is precisely with such systems that the most significant problems in understanding occur. The new intermediate representation must enable large programs to be modelled, both in theory and in any practical implementation. Construction algorithms must not be prohibitively expensive whilst the resulting representation should be space efficient.

1.5 Thesis Outline

The remainder of this thesis is organised as follows.

Chapter 2 reviews the field of program comprehension. Program views are introduced and a number of software maintenance environments evaluated. Research into the mental processes involved in program comprehension is also discussed.

Chapter 3 examines in further detail a number of existing intermediate program representations (IPRs), assessing the strengths and limitations of each. The representations are drawn from the fields of data flow analysis and program slicing, with more recent developments unifying a number of individual representations to provide a maintainer with different program views. An important feature of many of these IPRs is the representation of the subject program's data dependencies. The presence of pointer variables and dynamic allocation within a programming language such as C make the calculation of these dependencies a difficult task. Current research in this field from both the software engineering and compiling communities is discussed.

Chapter 4 introduces a new fine-grained dependence-based program representation, the Combined C Graph. The new graph enhances and modifies previous IPRs to allow many features of the C language to be modelled. Embedded side effects and control flows, the C parameter interface, control structures, pointer, structure and array variables, external and static variables and standard library routines are each addressed. The program views made available by the CCG are described

and a formal outline of the vertices and edges of the CCG given. The CCG representation is finally demonstrated by means of a theoretical example for a small C program.

Chapter 5 describes a prototype CCG system. The CCG system analyses the subject C program to produce a Prolog fact base. Prolog meta programs enhance this fact base to give a complete CCG representation. A maintainer can then perform queries on the CCG representation to create a number of program views. The Prolog representation may also be translated allowing the maintainer to view the CCG using a graphical display tool.

Chapter 6 details the results achieved using the prototype CCG system. CCGs are constructed firstly for four small C programs of up to 121 lines of code, involving different features of the C language. Two programs of up to 1000 lines of code are then analysed to demonstrate the applicability of the CCG to the representation of larger programs. Empirical results for the construction times and space requirements of the CCGs are given. Examples of the views and information made available to maintainers are then outlined, and finally two scenarios are described illustrating the use of the CCG system in software maintenance tasks.

Chapter 7 presents an evaluation of the CCG representation. The language coverage provided by both the theoretical CCG and the prototype implementation is first addressed. The program views made available are then discussed. The algorithms used in each stage of the construction of the CCG are then evaluated, both in terms of the results achieved and the theoretical complexity. Finally the space requirements of the CCG representation are analysed.

Chapter 8 finally presents a summary of the research in this thesis, and addresses the criteria for success defined above. A number of areas of further work are also outlined.

Chapter 2

An Overview of Program Comprehension

This chapter reviews work in three areas of program comprehension. Firstly a variety of program 'views' are discussed, in particular data and control flow analysis and program slicing. Secondly, existing software maintenance environments are described and evaluated with respect to the views provided, applicability to real programming languages and external user interface. Finally, differing theories of the mental processes of program comprehension are analysed.

2.1 Program Views

2.1.1 Data Flow

Data flow analysis is the process of collecting information on the flow of data through a program, in particular the definition and use of variables. The original use of data flow analysis was as a means of detecting safe conditions for optimisations within compilers. Information collected using data flow analysis may be in the following forms:

- reaching definitions - variable definitions reaching a program statement.
- reachable uses - variable uses reachable from a program statement.
- live variables - variables whose value at a program statement may be used at some following statement.
- available expressions - expression evaluations which reach a program statement without any intervening definitions of the variables involved in the expression.

A compiler can make use of such information to perform optimisations such as constant folding and dead code elimination. Two families of algorithms have been developed to solve these data flow problems, iterative algorithms and elimination algorithms. A summary of each and their use in compiler optimisations is given by Aho et al[3].

The use of data flow analysis has more recently extended beyond compilers. Software tools providing information to programmers on variable definition and uses, particularly across procedure boundaries (interprocedural data flow analysis), have been used in error detection during debugging and as an aid to program comprehension and impact analysis during maintenance. Data dependence information, indicating relationships between statements which provide and use data is of particular importance in helping a maintainer's understanding of the effects of a maintenance change. Data dependence information is typically presented in the form of definition-use graphs.

Error Detection

Data flow analysis is a technique which can be a powerful method for detecting errors in software and improving its quality. An early paper linking data flow analysis and software reliability is by Fosdick and Osterweil [28].

Data flow in a program is expected to be consistent in various ways. If the pattern of usage of a variable is abnormal in any way, this is said to be an anomaly. Anomalies are commonly caused by programming errors such as confusion of names, incorrect parameter usage and omission of statements. The aim of the analysis is to find anomalies in large bodies of code with arbitrarily complex data flow.

Algorithms are presented to detect anomalies involving the use of sets based on three events - variable definitions, undefinitions and references. Anomalies are identified by unexpected combinations of events, for example two consecutive definitions without an intervening reference. A tool DAVE is described which implements the algorithms to find these anomalies. Array variables present serious problems since subscript values cannot be evaluated. Array elements are therefore treated as a single 'aggregate' variables. The programmer must also still determine the actual underlying errors that are the cause of the anomalies.

A second prototype tool Omega, based on DAVE, is described by Wilson and Osterweil[93]. Omega detects anomalies in C code, although operations allowed on pointers are restricted and the algorithms employed may become exponential.

Each of the static data flow analysis tools described so far suffer from drawbacks. They are slow and restrictions are placed on the code that can be analysed. Array analysis for example would

require run time knowledge and pointer analysis is difficult. Dynamic data flow analysis is an alternative data flow analysis method which provides more accurate pointer and array information. The use of variables is analysed as the program is executing by instrumenting the program with extra code involving state variables and state transfer functions. State variables record the last action performed on a variable, which may be either 'reference', 'define', 'undefine' or 'anomalous'. A state transfer function takes as input the current value of a state variable and the action being performed on the variable and returns the appropriate new value of the state variable. This may be 'anomalous' if the action violates the variable usages described earlier. Early work on this method was carried out by Huang[46] who considered Fortran 66 programs. Calliss and Cornelius[16] extend this work to C programs, considering array elements, structured variables, pointers and dynamic variables.

Aliases

In order for data flow analysis tools to compute accurate solutions, the problems of variable aliasing must be overcome. An alias exists when a single storage location may be accessed by more than one name. Parameter passing by reference and pointer assignments are possible sources of aliasing. To accurately determine variable definitions and uses in interprocedural data flow analysis, the aliases existing within each procedure must be calculated.

Early work on the detection of aliases is described by Banning[8]. Banning's work deals with ordinal types, and the creation of aliases through reference parameters and Pascal-like nested procedure declarations. The more complex issue of dealing with pointers is not addressed. Banning describes a simple recursive depth first search algorithm to detect all possible aliases. Starting with trivial alias pairs created whenever a call site has a repeated actual parameter, e.g. $P(z, z)$, the algorithm descends call chains finding possible alias pairs arising from the original site.

Further work describing the detection of aliases is reported by Cooper[21]. Unlike Banning's paper which deals with aliasing as an issue in finding procedure side effects, Cooper's paper is concerned with the aliasing problem itself. Cooper considers initially only the two level name scoping of Fortran and not nested procedures.

Cooper's algorithm commences by detecting induced aliases at each call site directly from the source code. An iterative data flow algorithm propagates the sets of introduced aliases throughout the program's call graph, halting when no further propagation is possible. The algorithm has worst case complexity $O(n^2)$ but the actual behaviour will depend on the frequency of alias introduction and propagation. Optimisations are possible by effectively reducing the size of the call graph by

detecting call sites which do not introduce aliases or pass formal parameters. A simple extension to the algorithm permits analysis of nested procedures where aliases can be propagated into a nested block in which both aliased variables are visible. Cooper provides a detailed discussion of aliasing problems but, like Banning, does not attempt to deal with pointers.

2.1.2 Control Flow

Control flow analysis provides information to a maintainer on the control paths through a program. These paths are typically represented as graphs or tables and help a maintainer in understanding the control flow within a piece of software.

Control dependence information represents directly relationships between program statements, indicating that the execution of one statement is conditional on the execution of another statement. Construction of control dependencies is discussed by Ferrante et al[25] who present an algorithm based on dominance relations.

Interprocedural control flow is typically presented in the form of a program call graph, indicating direct call relationships between procedures. Vertices of the call graph represent procedures and edges of the graph represent calls between them. This information is particularly useful in the understanding of large systems and can also be used as a basis for the propagation of data flow information between procedures. For programs without procedure parameters, the construction of the call graph is simple. The vertices and edges of the call graph may be determined by analysing each procedure in turn to find references to other procedures. When procedure parameters are present, the order in which procedures are analysed becomes important and a reference to a formal procedure parameter may represent the invocations of distinct procedures. Construction of the call graph for a language with procedure parameters is described by Ryder[80]. The algorithm is proved to be correct, has complexity $O(n^3)$, but cannot be applied to programs containing recursive procedures.

2.1.3 Program Slicing

Static slicing

In order to aid understanding, large programs are decomposed into smaller parts. Examples of decomposition are procedures and abstract data types which both allow understanding independent of the context within the program. Program slicing, first introduced by Weiser[89], is a decomposition method based on both control and data flow analysis. A slice is a reduced program which

restricts the behaviour of a program to a subset of interest. More specifically, a slice $S(v, n)$ of a program P on a variable or set of variables v at statement n, gives the portions of the program P which contribute to the value(s) of v before n is executed. Various uses of program slicing have been described. These include debugging, testing, maintenance, parallel processor distribution and code understanding.

Weiser presents a formal definition of a program slice and describes an iterative algorithm to find an intraprocedural slice i.e. a slice within a single procedure. Corrections to this algorithm are given by Leung and Reghbati[58]. The complexity of the algorithm is $O(ne \log e)$. The algorithm is then extended by Weiser to include procedures calling or called by the original procedure in which the slice is taken, giving interprocedural program slicing. Weiser describes four advantages of program slices.

- They can be found automatically. This gives possible uses in calculating metrics.
- They are smaller than the original program so they are easier to understand.
- They execute independently of each other, allowing parallel execution.
- They reproduce exactly a subset of the program's original behaviour. This gives uses in verification and testing.

Three disadvantages are also given.

- They can be expensive to find.
- A program may have no significant slices.
- Their total independence may cause additional complexity in each slice that could be cleaned up if simple dependencies could be represented.

An earlier paper by Weiser[88] shows that programmers make use of slices when debugging. Weiser suggests that programmers attempt to reason backwards through the flow of control from a point where an error becomes manifest, constructing a slice mentally as they do so.

An experiment was constructed in which programmers debugged three programs and were then asked to recognise five types of code fragments - relevant slices, irrelevant slices, relevant contiguous regions, irrelevant contiguous regions and random statements. The results showed that the programmers recognised slices as often as relevant contiguous fragments, indicating that they had in fact abstracted the slices when debugging.

Weiser's slicing algorithm has some problems when applied to C code. It does not cater for arrays and pointer variables and does not deal with statements such as `break`, `continue` and `goto` which have effects on the slice. Jiang et al[47] describe some of these problems and present enhancements to the algorithm to allow slicing with pointer and array variables and `break`, `continue` and `goto` statements. An interprocedural slicing algorithm for C is also given.

Ottenstein and Ottenstein[67] describe the construction of program slices using an alternative method based on an intermediate program representation known as the Program Dependence Graph (PDG). The PDG represents explicitly the data and control dependencies which are the bases of program slicing and is consequently ideal for constructing slices. The vertices of the graph represent the statements of the program and the connecting edges represent either data dependencies or control dependencies between the statements. A slice is constructed by selecting the appropriate vertex in the PDG and from this vertex traversing the graph backwards. Visited vertices represent the source lines of the slice. The slice is found in linear time, in comparison to Weiser's $O(ne \log e)$ algorithm. Input/output is also accounted for correctly and irrelevant statements on multi-statement lines are not included. However, the slices found using this method are more restricted than those of Weiser. A slice must be taken with respect to a variable defined or used at a particular statement, rather than an arbitrary variable at a statement.

The slicing algorithm described by Ottenstein and Ottenstein discusses only the case of programs consisting of a single monolithic procedure. Their work is extended by Horwitz et al[45] who describe a new graph, the System Dependence Graph (SDG) which models multiple-procedure programs with parameters passed by value-result. Horwitz et al show that Weiser's original interprocedural algorithm is imprecise since no account is taken of calling context; a called procedure may return to any callee procedure, not only the one from which the call was made. A new two-stage traversal-based interprocedural slicing algorithm is described, with phase one not descending into called procedures and phase two not ascending into callee procedures. In each case the effects of such procedures are summarised by a new transitive dependence edge which solves the problems of calling context. Again the complexity of the algorithm is linear in the size of the graph.

The graph traversal program slicing methods are intuitively much simpler than the iterative algorithm presented by Weiser and are additionally cheaper in terms of complexity, although the costs of graph construction must be considered. A limiting factor in each case is the language that can be represented using the graphs. Neither Ottenstein and Ottenstein nor Horwitz et al deal with pointer variables, self-referential structures or unstructured programs.

Gallagher and Lyle[32] present a paper in which program slicing is applied to the software

maintenance process. Slicing is used to obtain a decomposition of the program. The first step is to build for one variable a 'decomposition slice' which is the union of certain slices taken at certain program statements on the given variable. A second slice, the 'complement slice', is obtained by removing certain statements of the decomposition slice from the original program. The complement slice must always remain fixed when changes are made to the decomposition slice. The contents of the decomposition slice are independent of the slicing method employed. Restrictions are presented as to the possible changes that may be made to the decomposition slice, and a linear time algorithm given for merging the modified slice back into the original program.

The authors claim that the changes made will have no impact on the complement slice, and therefore only the modified slice need be tested. This gives rise to a new software maintenance process model in which regression testing is no longer necessary and consequently cost savings can be made. As yet the decomposition method described has not been applied to a large software system and its effects on real integration testing must be investigated.

Dynamic slicing

One problem of the program slicing methods described so far is that the slices produced may not be significantly smaller than the original program. The usefulness of a slice will decrease as its size increases. These 'static' slices contain all the statements that might affect the value of a given variable occurrence for any input values. Agrawal and Horgan[1] investigate 'dynamic slicing' in which the statements contained in a slice are those that actually affect the value of a variable for a given program input. This approach is of use in debugging and testing where specific program inputs are generally available.

Several approaches to deriving intraprocedural dynamic slices are described by Agrawal and Horgan. The first two make use of the PDG described by Ottenstein and Ottenstein[67]. These are simple and efficient methods but the slices resulting may be larger than necessary. The third approach uses a Dynamic Dependency Graph in which a new vertex is created for each occurrence of a statement in the execution history. This method produces accurate slices but the size of the graphs may be unbounded. A reduced Dynamic Dependency Graph is finally described in which a vertex is added only if it can cause a new dynamic slice to be introduced.

Whilst dynamic slicing is of value in debugging and testing where the programmer is dealing with specific program inputs. the use of the technique in relation to software maintenance has not been addressed. Also, no empirical studies have been undertaken to compare the size of dynamic slices to that of static slices. The technique of dynamic slicing is new and further work is required

to evaluate the different algorithms with respect to time complexity, applicability to real languages and ease of implementation.

2.1.4 Other Views

Component dependencies

In addition to data and control relationships described earlier, program understanding will be made easier by the provision of component dependence information, i.e. relationships between source code components. Cross reference tools collect all references to a software object, presenting this information to a maintainer in the form of listings of declarations and subsequent references. Wilde and Huitt[91] describe 'definition dependencies' which represent the use of a program object in the definition of another object, for example the use of a symbolic constant to set the dimension of an array or of a user defined type in a variable definition.

Decision views

Wild and Maly[90] identify that a significant missing aspect of current software documentation is an explanation of why decisions are made. Most documentation describes what a system does and how it does it. The authors propose a complimentary system to standard static analysis tools which can document design decisions. A knowledge base stores decision objects which represent particular decisions, alternate choices, complexity analyses and reasons for selecting the choices made. A decision dependency graph links justifications of decision objects and makes it possible to

- trace from a result all the decisions that support it.
- find all the results that depend on particular decisions.

The environment allows the user to find relevant information and understand the system, assess the impact of design choices and design and implement solutions. As yet no prototype tool exists although future work on a hypertext, multiple windows system is indicated.

High level views

Whilst the aim of this research is to provide a software maintainer with programming level information, Letovsky[56] recognises that the process of program comprehension relies on building a mental representation both bottom-up from the code level and top-down from the problem domain level.

Program understanding can be aided by the provision of higher level information and in particular by providing traceability from these higher level views to those at the programming level.

Von Mayrhauser and Vans[86] identify the lack of tools at levels other than the programming level. Suggesting the importance of tools supporting rather than forming the comprehension process, experiments with an integrated code comprehension model lead to a series of tool requirements. Tools should present information at each level of the comprehension model and should aid in switching between its components, with connections between these levels.

Research in this direction is described by Avellis et al[7] who propose an enhanced view of a software system, the Software System Model (SSM) and a Software Evolution Expert System (SEES) with four basic functionalities.

1. Find code to be changed.
2. Advise maintainer on how to make changes.
3. Find impact of change.
4. Organise 1-3.

The authors indicate that for effective maintenance additional views of a system are required. Two properties are given to define a useful view.

- The view focusses the attention of the maintainer on a small portion of the system.
- The view facilitates defining a map between change descriptions and implementations.

Six views are described from which the SSM may be constructed.

programming and structural views Low construction costs but no links between change descriptions and code.

architectural views Standard architectures can allow the SSM to be partitioned into subsets and changes can be indexed to system parts.

waterfall views Waterfall information is available, understood and provides good indexing between change descriptions and implementations.

domain views Provide fine granularity for change indexing.

domain network views Expensive to construct but are the most powerful views. Each of the other views except waterfall views are included. Indexing is refined even further.

A prototype has been implemented with a generic 3GL programming language view with change propagation and change performance knowledge bases. Presently the only implemented view is at the programming level. The most interesting section of the paper is the description of the different views. Domain, standard architectures and domain network views are providing information at higher levels of abstraction and have much in common with the results of research into the theory of the program comprehension process, as reported by Brooks[13] and Letovsky[56].

2.2 Tools for Program Comprehension

Various tools exist which are specifically aimed to help with the program comprehension process. The method employed by most tools is to perform static analysis on the source code and to store the resulting information in a central database. This gathering of information, either automatically or manually, is often known as *redocumentation.* The maintainer is then able to access the information as required. This section describes various environments which extend this basic idea, each providing the maintainer with a variety of programming level views.

Documentation useful during program comprehension may also be created during the development phase of the software life cycle. Research into documentation environments supporting the production, management and use of program documentation is described briefly.

2.2.1 C Information Abstractor

The C Information Abstractor described by Chen et al[18] is a static analysis tool which stores the information collected in a relational database. This tool provides information with the following qualities:

- global information abstraction - emphasis is on global references rather than local references. This reduces database size and increases speed.
- database support
- simple database queries
- efficiency

The authors suggest enhancements to the system are possible in the way of an incremental database (avoiding the need for complete re-analysis of the system in response to system changes) and the addition of structured comments to record information that cannot be extracted from the code.

The tool provides simple cross referencing information but without a graphical user interface. The authors suggest the use of windowing systems to provide 'multiple views' of the software. The description of views however goes no further than the use of multiple windows to display cross references.

2.2.2 DOCMAN

Foster and Munro[31] describe a documentation system built on cross referencing. The tool is aimed at maintenance programmers working on large software systems and allows documentation produced by the programmers to be linked with cross referencing information. Three types of documentation are included:

- encyclopaedia - descriptive comments about objects in the source code.
- glossary - descriptions of special words which appear frequently in the documentation.
- overview - high level descriptions of the system.

DOCMAN allows the user to cross reference between documentation entities, cross referencer output and the source code itself and consequently ease the understanding process.

The tool is suitable for use with undocumented systems since documentation can be added incrementally; as a maintenance programmer works on an area of the code the area is documented.

2.2.3 Redocumentation of Systems Using Hypertext Technology

The concepts introduced by Foster and Munro[31] are expanded further by Fletton[26] and Fletton and Munro[27] to give a hypertext tool linking cross references, source code and documentation. Links between source code and cross references are generated automatically whilst links in the documentation are generated as the documentation is created.

Eight desirable properties of a redocumentation tool are listed:

- incremental documentation - it should not be necessary to document the entire system at once.
- casual update - it must be easy to update the documentation as the programmer examines the code.
- quality assurance checks should be possible on documentation updates.
- team use

- configuration management - document different versions of the system.
- integrated source code
- integrated automatic documentation - static analysis
- information hiding - the documentation can be read at different levels of abstraction.

2.2.4 An Environment for Understanding Programs

Cleveland[19] describes an environment to aid the understanding of old assembler code. The system presents various views of the software - control flows, data flows, call graphs and cross references. This information is collected using various static analysis tools and is stored in a database. Non-verifiable or 'soft' information may also be added by the user. This is similar to the encyclopaedia information described by Foster and Munro[31]. Cleveland[20] also describes a user interface based on the use of a large colour screen, multiple windows and an intelligent cursor. 'Window containment' assures that information relating to an object is grouped together. 'Linkage marking' through the use of colour highlighting is used to relate representations of an object.

Only a simple prototype tool has been implemented and the papers contain no description of how information within each view is related. The views presented are limited to those only provided by static analysis tools and the resulting environment is very similar to that described by Fletton[26] and Fletton and Munro[27].

2.2.5 VIFOR

Rajlich et al[74] describe a maintenance environment for Fortran, VIFOR (Visually Interactive Fortran). The tool provides a visual form for Fortran programs in addition to the source code. This visual form is a simple two column graph with vertices indicating source files, subroutines and commons and the edges calls, references and 'belong to' for file contents. The authors describe views of the system but in actual fact these are only subsets of the database formed by queries.

2.2.6 CARE

Linos et al[60] extend the VIFOR system to facilitate the comprehension of C code, producing a software environment CARE (Computer-Aided Re-engineering). The CARE system maintains a repository of program dependencies with a data model comprising five entity types and four relationships. Program understanding is achieved via a graphical model combining a hierarchical

control flow display (i.e. call graph) and a new 'colonnade' representation of the program's data relationships. This 'm-level graph' extends the two column graph of VIFOR. The user is provided with functions to display monolithic representations of the hierarchical or colonnade graphs or to display multiple views comprising fragments of the code, colonnade or call graph. Transformation tools allow the user to derive control flow or data flow views from the colonnade or hierarchical graphs, for example to pop-up the data flow graph for a specified function in the call graph. Various graphical operations are made available to provide information abstraction and to improve the display's readability. Simple empirical tests demonstrate the benefits of using the CARE system. CARE provides the programming level views seen in other maintenance environments but with a more advanced graphical user interface.

2.2.7 Microscope

Microscope described by Ambras and O'Day[4] is a knowledge based tool for use in software maintenance. The system can perform static and dynamic analysis on programs written in CommonObjects and Common Lisp. The system is able to perform impact analysis and execution monitoring and can record execution histories for browsing. Views at different levels of abstraction are also permitted. These may be module hierarchies, cross references, call graphs or execution histories.

The prototype implementation analyses only a subset of CommonObjects and Common Lisp and provides cross references, some execution monitoring and a graphical browser. The authors suggest that the user interface and the response times of queries are critical. Microscope is a more ambitious project than the environments described so far but the prototype system does not appear to provide any significant improvements over the other systems.

2.2.8 PECAN

Although a software development tool rather than a maintenance tool, the provision of multiple views of a software system is the basis of the PECAN system reported by Reiss[75]. The internal representation used by the system is the abstract syntax tree, which the programmer can view in a variety of different forms through the use of multiple windows.

Possible program views described in the paper are a syntax directed editor, Nassi-Schneiderman flow graphs, data flows, module hierarchies and declaration views. The user can perform editing operations using a displayed view which will be reflected in the internal form. Any other displayed program views will consequently be updated. Whenever the abstract syntax tree is updated, an incremental compiler will update a corresponding semantic representation incorporating a symbol

table view, data type view, expression view and flow view. Execution views of data structures and the stack can similarly be updated as the program is executed.

The current prototype only allows changes to be made in the structured editor view and additionally provides only Nassi-Schneiderman flow graphs. The incremental compiler and interpretor are not yet completed and no descriptions are given of how a code change is incorporated into the abstract syntax tree and from there into the other program and semantic views. However a tool providing the variety of views described and allowing incremental updates of the views could be applied to uses in software maintenance.

2.2.9 Software Documentation Environments

Various tools exist which provide support for the production, management and use of documentation during each phase of the development life cycle. Such information will be of considerable value during the maintenance phase. The tools provide facilities such as central storage, traceability and easy access and update. However, undocumented systems are less suitable to this type of tool. Redocumentation of a previously undocumented system is a large task and in most cases is not economically feasible. FORTUNE[66], the Document Integration Facility (DIF) [33][34] and SODOS[41] are similar integrated documentation tools based on the waterfall life cycle model.

2.3 Theories of Program Comprehension

In order to carry out a maintenance task, a programmer must first acquire an understanding of the subject system. This stage of program comprehension is often the most time consuming of the entire maintenance process and consequently is an area in which savings can be made. Methods of avoiding errors in program comprehension are also required to enable more successful and less costly software maintenance.

This chapter discusses research into the problems of program comprehension. The research is divided into three areas. The first deals with methods by which information can best be presented to a software maintainer. The second describes the mental processes involved in the comprehension of source code. The third area discusses research based on the theory of *programming plans*, which considers expert programmers to make use of generic computational structures rather than primitive programming elements.

2.3.1 Information Presentation

A maintenance programmer will spend a large amount of time in examining the source code of the subject system in order to gain an understanding of its functionality and the relationships and dependencies between its components. Savings will result if this process is made less time consuming and more error free. Research has consequently been carried out in the areas of how best to present information to a maintenance programmer and of which information is of most use in code understanding.

Schneiderman et al[81] identify the importance of presentation of information to maintainers. Their study concentrates on the use of new larger, faster display screens and on strategies for displaying more information using efficient display formats. The authors deal with coordinated window systems in which windows and their contents appear and scroll automatically as a result of user activities. The user is freed from the chore of creating, positioning and manipulating windows.

Four strategies for coordinated windows are presented.

- Fusion - Many lines of code are displayed in sequence in multiple windows. This technique is used with large sections of code to reduce 'page turning'.
- Synchronised scrolling - Two or more files in different windows are scrolled together. This is useful for comparing versions of code, evaluating test cases, interpreters etc.
- Embedded selection - Names can be selected from the source code and extra information, for example manual pages, declarations and comments, may be displayed in another window.
- Hierarchical browser - A representation of the high level structure that may be used to access the source code or other text. The implemented system has two windows, one displaying program structure at different levels of abstraction and the other associated source code. Other views can be added such as cross references, execution histories, data flows etc. Program comprehension is helped by showing structured information and the underlying design. An empirical test shows that maintainers perform better with the browser than with simply the source code.

The strategies outlined in this paper are very similar to those employed in maintenance environments such as those described by Cleveland [19] and Fletton and Munro[27]. Hypertext systems give embedded selection and generally involve some form of hierarchical browser. Fusion and synchronised scrolling do not appear to have been used elsewhere.

2.3.2 Comprehension Theories

Before attempting to make a change to a piece of code, a maintenance programmer must gain an understanding of that code. Various models exist to explain the cognitive processes that programmers go through when performing this task. A better understanding of how comprehension occurs may lead to code which is more suited to the maintainer's needs. Summaries of research into program comprehension are given by Robson et al[79] and Deimal and Naveda[23].

Letovsky[56] reports an empirical study of the cognitive processes involved in program comprehension. Six professional programmers were video-taped whilst enhancing an existing Fortran 77, 250 line program. The subjects were additionally supplied with some program documentation and were asked to 'think aloud' whilst examining the code.

Letovsky finds that the programmers often form 'inquiries' concerned with the same topic. An idealised inquiry is made up of four parts.

1. The subject encounters a fact and asks a question.
2. The subject conjectures an answer.
3. The subject attempts to find an answer in the code or documentation, or by detailed reasoning about the program.
4. The subject finally draws a conclusion and resumes the previous activity.

Letovsky presents taxonomies of questions and conjectures and gives examples from the thinking aloud protocols. He views programmers as knowledge based understanders with

- a knowledge base of previous expertise and background knowledge.
- a mental model representing the current understanding of the program.
- an assimilation process - how the programmer builds the mental model.

The mental model consists of layers. At the top level is the program specification and at the bottom the implementation. The intermediate layers are annotations which link the specification goals and implementation. Letovsky describes how this model is built both top-down and bottom-up and how questions arise when the programmer finds the mental model to be incomplete.

One interesting conclusion of the paper is the application of the research to the design of program documentation and documentation standards. Documentation should facilitate easy answering of the questions posed by programmers.

An alternative model of the comprehension process is described by Brooks[13]. When a programmer understands a program, a mental model is built up of successive knowledge domains, from the problem domain to the programming domain. The relationships between these domains must also be understood. Brooks' theory differs from that of Letovsky in how the mental model is actually constructed. Brooks describes this process as being largely top-down. The programmer generates hypotheses about the program and attempts to verify these from the code. These hypotheses are generated in a hierarchical manner. Brooks claims that the bottom-up technique of code reading considered by Letovsky is less powerful and less important than the top-down method.

Pennington[71] provides support for the mental models of both Letovsky and Brooks. She suggests that programmers build at least two mental models, in particular a program model and a domain model. The program model relates to the program's textual structures and the domain model to the objects and functions in the problem domain. For effective comprehension a programmer must be able to cross reference between the two models.

2.3.3 Programming Plans

A significant body of research has been carried out in the area of programming plans. Rather than thinking in terms of low level primitive elements such as assignments and tests, expert programmers instead build up a knowledge of commonly used computational structures called programming plans, which can be combined to implement higher level abstractions. Examples of plans are list enumerations, binary searches and successive approximation loops. The research can be divided into two distinct areas, firstly experiments to investigate the cognitive basis for the theory of programming plans and secondly the development of plan-based tools for code analysis and synthesis.

Theory of programming plans

Soloway and Ehrlich[82] suggest that programmers have two basic forms of programming knowledge.

- Programming plans - program fragments that represent stereotypic action sequences in programming, for example a running total loop plan, an item search loop plan.
- Rules of programming discourse - rules that specify conventions in programming such as the name of a variable should usually agree with its function. These rules set up expectations in the minds of the programmers about what should be in a program.

The authors suggest that programs are composed from programming plans modified to fit the needs of specific problems. The composition of the plans are governed by rules of programming discourse.

Two empirical studies are described which evaluate the hypothesis that expert programmers possess programming plans and rules of discourse. The first study asked programmers to fill in blanks in programs. The second asked programmers to recall the programs. In both cases plan-like and unplan-like programs were used. In the first study expert and novice programmers, and in the second only experts, were used as subjects.

The results of the first study showed that the performance of experts was reduced to that of the novices with unplan-like programs. The second study showed that plan-like programs were easier to remember and that the critical lines, i.e. the plans, were remembered first.

The authors conclude that advanced programmers have strong expectations about what programs should look like. When these expectations are violated their performance drops drastically. Style is not just a matter of aesthetics; there is a cognitive basis for writing programs in the conventional manner.

The results show plans and discourse rules to have a powerful effect on comprehension. Surface complexity measures would be unable to detect the difficulties inherent in unplan-like programs.

Further work on program plans is described by Letovsky and Soloway[57]. The authors suggest that maintainers have difficulties understanding code containing delocalised plans, i.e. a plan realised by lines scattered in different parts of the program. When a plan is close together it is easy to recognise. When it is split up partial understanding of the program can result. Purely local understanding may lead to inaccurate understanding of the program as a whole.

The authors describe the task of program understanding as that of uncovering the intentions behind the code - the 'goals'. A plan is a technique for realising a goal in a particular implementation.

Six programmers were video-taped making a program enhancement, 'thinking aloud' as they did so. Four examples are described of comprehension failures due to delocalised plans and possible solutions given to avoid these misconceptions. These generally involve the programmer being more explicit in comments and other documentation. For example the 'role' and 'goal' of a variable and any 'non-normal' updates of a variable should be documented. The authors suggest documenting plans, indicating the purpose and implementation of the plan with pointers to the code. Other solutions are to use symbolic execution and data flow analysers. The latter are particularly suitable since delocalised plans can be considered as plans with data flow links.

Soloway et al[83] describe further work in the area of dealing with the problems of delocalised plans. The authors describe an experiment involving the performance of an enhancement to a 250 line Fortran program, given the provision of typical program documentation. Again the 'thinking

aloud' protocol was used as the programmers carried out the task and the 'inquiry' episode originally described by Letovsky[56] observed when a programmer's expectations were not met. Particular breakdowns in expectation were found with a delocalised plan involving the retrieval and processing of a record. Two higher level strategies were observed, *systematic* and *as-needed.* The systematic strategy involves tracing the flow of the entire program from the beginning whereas the as-needed strategy involves only studying the areas of the program considered useful. Programmers employing the as-needed strategy were found to have difficulties with delocalised plans whilst the systematic strategy becomes unworkable with large programs.

The authors develop a new type of program documentation explicitly identifying causal interactions in the delocalised plans and relating the interactions to a listing of the source code. Improvements in comprehension of the delocalised plan result with this new documentation, although problems are still found with programmers who actually misunderstand the code rather than those simply unable to form any understanding.

Plan-based tools

The Programmer's Apprentice The Programmer's Apprentice project is a long term project involving the use of artificial intelligence and software engineering techniques. The aim of the project has been to develop an interactive knowledge based tool to provide assistance to programmers in both the construction and maintenance of programs. At the basis of this work has been the use of 'inspection methods', that is performing recognition of 'cliches' (i.e. plans) from a known library to allow program synthesis, analysis and verification.

Plan Calculus A formalism for the representation of program plans has been developed by Rich[76][77]. This formalism is the basis of the whole Programmer's Apprentice project. The plan calculus allows plans to be represented independently of the original source language and to be combined in a straightforward manner. Plans can be verified and relationships between one plan and another made explicit through the use of 'overlays'.

Rich reports that translators from source text to the plan calculus representation have been constructed for subsets of Lisp, Fortran and Cobol. Whilst problems such as side effects and aliasing can be represented by the plan calculus, the language features permitted by the translators are much more limited.

Recognizer The Recognizer[78][92] developed at MIT is a prototype tool to detect plans within code using a library of standard plans and to build hierarchical descriptions of the plans found. The code is first translated into the plan calculus, this step being the only language dependent part of the tool. The possible use of different syntaxes to code a plan and the use of delocalised plans is hidden by this representation. The plan calculus produced is then encoded as a flow graph, subgraphs of which will comprise the specific plans. The Recognizer has a library of several hundred plans and overlays which are each coded as flow graph grammar rules. The flow graph of the input program is parsed and any matched right hand side of a grammar rule replaced by the corresponding left hand side. In this way a plan in the code is replaced by a more abstract operation. A hierarchy of plans will result which represents the program's design tree. From this hierarchy textual documentation can be generated by inserting the actual identifiers from the code into slots in standard templates associated with each plan. Unrecognisable code, i.e. that not made up of known plans, is dealt with in two ways. Firstly the parse can be started at each intermediate position of the flow graph and ended before the complete flow graph has been parsed. Secondly, low level plans can be recognised even if they cannot be combined using higher level plans.

The prototype Recognizer has been demonstrated with only small Common Lisp programs. Data plans for modelling data structures and plans involving side effects are not dealt with. For the Recognizer to be of value in the maintenance of real software, complex data structures and side effects must be handled. The current system additionally performs an exhaustive search which is purely based on the structure of the flow graph and the encoded plans. Ways of limiting the exhaustive search are required if this technique of plan recognition is not to prove too expensive. For this purpose the authors suggest the possibility of additionally performing a top-down specification driven analysis to produce expectations by which the search can be limited.

PROUST The PROUST system developed by Johnson and Soloway[48] performs a plan-based analysis of novice Pascal programs with the aim of reconstructing the design and implementation steps originally performed by the programmer. In this way, bugs in the program can be understood and an explanation given to the programmer.

PROUST is supplied with the subject Pascal program and a specification in the form of a set of goals. Using a library of standard plans and common bugs, PROUST attempts to find a mapping between the requirements and code - a program interpretation. The space of possible interpretations is increased by the possibility of programs containing bugs. The system must therefore be able to generate a wide variety of programs. PROUST maintains an agenda of goals, selects the first and

from its knowledge base and determines if the goal selected requires decomposing. If not, the plan library is searched for appropriate plans and matches made with the source code. Various heuristics are used to constrain the generation of program interpretations. Each candidate interpretation is evaluated by examining how well other parts of the program conform to expectations based on the current interpretation.

The system's knowledge base has been tailored to analyse a particular programming problem and its performance evaluated on 206 student programs. A complete analysis resulted in 72% of the cases, of which 95% were found to be correct.

The PROUST system appears to have been reasonably successful in its task of analysing simple programs. However, fundamental limitations will arise with unusual bugs, novel plans and ambiguous cases where human interaction is required. A practical student tutoring system also requires the ability to handle a variety of problems and as indicated in the paper, would need some way of effectively interacting with the students.

Whilst there is considerable supporting evidence for the theory of programming plans, only limited success has been achieved with these practical applications. The need for large plan libraries and the problems of delocalised plans and 'unplan-like' code make the technique difficult to apply to large-scale programs.

2.4 Summary

This chapter has reviewed literature in three areas of program comprehension. Software views, in particular at the programming level, have been discussed in terms of their construction and usefulness for program understanding. Existing software maintenance environments were then described, focussing on the views provided and languages covered. Finally the cognitive processes involved in program comprehension were described, leading to an investigation of the theory of programming plans.

Chapter 3

Current Techniques

This chapter discusses research in two areas. Section 3.2 analyses a number of existing intermediate program representations (IPRs) and assesses the strengths and limitations of each. This is preceded by a discussion on graph terminology since it is on this theory that each of these IPRs are based. An important component of many of these IPRs is a representation of the program's data dependence information. The computation of this information is complicated by the presence of pointer variables and dynamic memory allocation and is an ongoing area of research. The theory and problems of this work are outlined in section 3.3.

3.1 Graph Terminology

A graph is made up of a set of elements known as *vertices* together with a set of arcs connecting the vertices known as *edges*. Formally, a graph G is represented by the relation $G(V, E)$, where V is a set of vertices $\{v_1, ..., v_n\}$ and E is the set of ordered pairs called edges, $\{(x, y) \mid (x, y) \in V \times V\}$. The number of vertices in G is represented by $|V|$ and the number of edges by $|E|$. Given any graph edge (v_1, v_2), then v_1 is the *source vertex* of the edge and v_2 the *sink vertex* of the edge. The vertices v_1 and v_2 are said to be *adjacent.*

The graph $G_s(V_s, E_s)$ is a *subgraph* of $G(V, E)$ if $V_s \subseteq V$ and $E_s \subseteq E$.

A *path* is a sequence of vertices such that each successive pair of vertices is adjacent. If two vertices are adjacent or are connected indirectly through one or more intermediate vertices, there is said to be a path between the two vertices. If on a given path, each vertex is visited only once, the path is said to be a *simple path.*

Graph edges may be either *directed* or *undirected.* A directed edge indicates that information flows in only one way along the edge from the source vertex to the sink vertex. An undirected edge

allows information to flow in both directions along the edge and can be defined as a pair of directed edges (v_1, v_2) and (v_2, v_1).

3.2 Program Representations

This section describes various existing intermediate program representations and discusses the associated benefits, limitations and drawbacks of each. The representations considered are drawn from two techniques used in program comprehension: data flow analysis and program slicing. Graphical representations have been used in the construction of data flow information whilst dependence-based representations are used as a basis for program slicing. More recent research has focussed on the possibility of unifying the representations from each area to allow the creation of a variety of program views from a single intermediate program representation.

3.2.1 Call Graph

A simple representation of a program is the program call graph. Vertices of the call graph represent procedures whilst edges represent possible calling relations between procedures. Edge labels may be added to represent the actual parameters of each call. An example call graph for the program in table 3.1 is shown in figure 3.1. In the absence of procedure parameters the construction of the call graph is simple. Where procedure parameters are present the call relationships can no longer be determined statically from the code itself and the $O(n^3)$ construction method described by Ryder[80] may be used.

Interprocedural data flow analysis based on the call graph is possible but this is limited to *flow-insensitive* algorithms i.e. the control flow within procedures is not considered. Information is gathered at each vertex as to the variables possibly defined or used within each procedure. This information can then be propagated throughout the graph to produce a solution for the desired data flow problem.

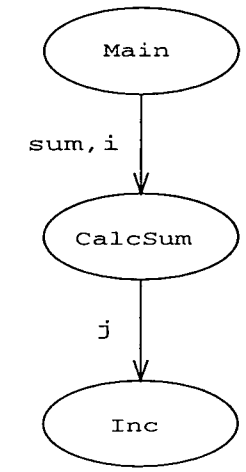
Whilst the call graph itself provides important information for program comprehension, only limited data flow information can be generated. Intraprocedural control flow information is not available and thus control flows and program slices cannot be constructed. The absence of specific locations for variable definitions and uses similarly prevents the calculation of cross reference information and more accurate flow-sensitive data flow analysis.

```
1. program Main()
2.    sum = 0;
3.    i = 0;
4.    while i<20 do;
5.      CalcSum(sum,i);
6.    endwhile
7.    i = i;
8.    sum = sum;
9. end.
```

```
10. procedure CalcSum(s,j)
11.    Inc(j);
12.    if s<100 then
13.       s = s + j;
14.    endif
15. return
```

```
16. procedure Inc(x)
17.    x = x + 1;
18. return
```

Table 3.1: Example program.



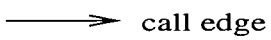


Figure 3.1: Example Call Graph.

3.2.2 Interconnection Graph

The *Interconnection Graph* (IG) described by Debnath and Bieman[22] allows the analysis of the interprocedural structure of a program. By analysing paths in the IG explicit and implicit interactions between procedures can be identified, enabling the development of interconnectivity measures, testing strategies covering interprocedural interactions and a basis for debugging tools.

Debnath and Bieman first describe the generalised program graph (GPG) which represents individual procedures written in an imperative language such as Pascal, Fortran or C. The GPG synthesises the control flow graph and data dependence graph - nodes represent variable definitions and edges control flow and data dependencies. A reduced GPG (RGPG) is then described which contains only nodes for parameter definitions and edges to represent control and data dependency relationships between the nodes. The interconnection graph (IG) is formed by adding 'interaction edges' to connect nodes denoting the definitions of actual parameters with nodes representing the definition of formal parameters. Debnath and Bieman then define control and data interaction paths between a pair of procedures and use this notion to determine both explicit and implicit interactions, where the interaction between two procedures cannot be identified by examining the procedures in isolation.

The interconnection graph is useful in allowing a maintainer to analyse the effects of a code change in other procedures, although the absence of control dependence information prevents the construction of program slices and the lack of local definitions prevents the computation of definition-use information.

3.2.3 Program Summary Graph

The *Program Summary Graph* (PSG) devised by Callahan[15] is an extension of the basic call graph which additionally permits *flow-sensitive* data flow analysis, that is control flow internal to procedures is considered. The PSG was originally developed as a representation to allow data flow analysis of do loop structures to enable parallelisation of Fortran programs. The control flow internal to each procedure is summarised and hence the graph is much more compact than the full control flow graph, growing linearly with program length.

The graph is made up of four types of vertices. These are 'entry' and 'exit' vertices for every formal reference parameter of every procedure and 'call' and 'return' vertices for every actual reference parameter of every call site. The vertices thus represent procedure entry, exit, call and return. Global variables are represented as additional formal and actual reference parameters.

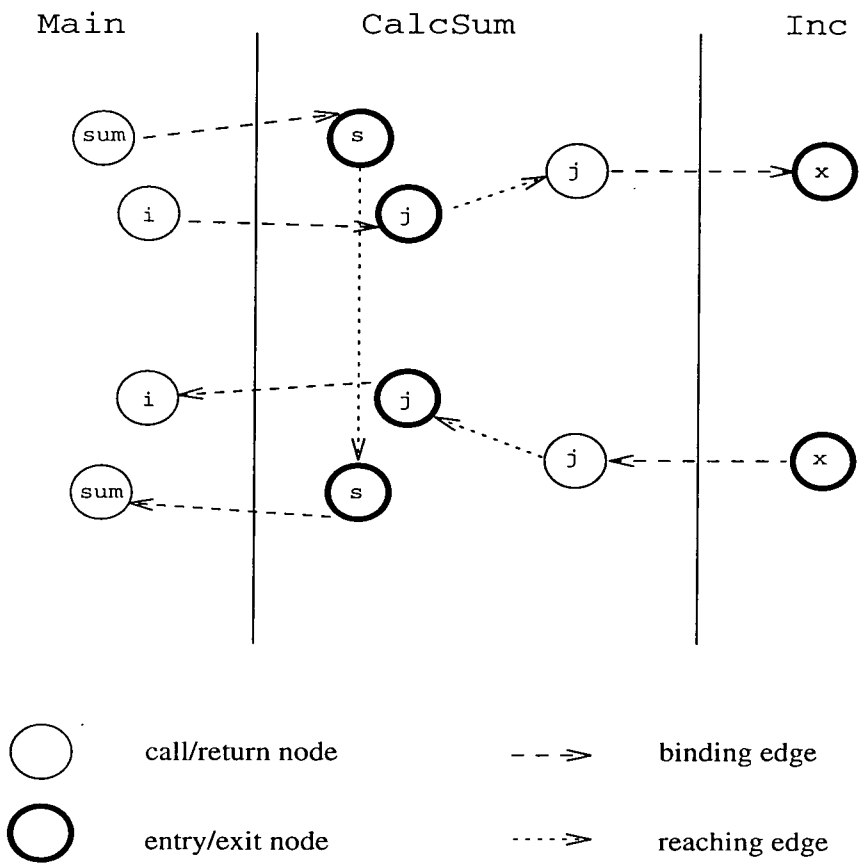


Figure 3.2: Example Program Summary Graph.

The PSG vertices are connected by two different edge types. The first, 'binding edges', connect actual parameters to formal parameters at procedure invocations and formal parameters to actual parameters at procedure returns. The call structure of the program is represented by these edges. The second edge type, the 'reaching edge', summarises the control structure internal to a procedure by indicating the flow of data between procedure control points. These definition free execution paths within procedures are calculated using standard data flow analysis algorithms. Edges on the graph indicate where definitions from an entry vertex reach a call site or a return statement, or where definitions from a call site reach another call site or return statement. An example PSG is shown in figure 3.2 representing the program in table 3.1.

Callahan presents *iterative analysis* algorithms using the PSG to solve interprocedural data flow problems such as *may be preserved*, i.e. there is a path on which the parameter is preserved unchanged over the call. For example, a reference parameter may be preserved over a call site if there is a path on the graph from the appropriate call vertex to the associated return vertex. These algorithms are less precise in the presence of aliases which are not addressed in detail.

The specific locations of definitions and uses within procedures are not represented and thus information such as definition-use associations, program slices and cross references cannot be calculated. Additionally, the graph does not represent calling context. Following paths on the graph can lead to a procedure returning to a call site other than the one at which the execution was invoked and hence inaccurate data flow solutions.

3.2.4 Interprocedural Flow Graph

Harrold and Soffa[40] extend the PSG to allow the computation of interprocedural definition-use and use-definition chains, defining the *Interprocedural Flow Graph* (IFG). In addition to the vertices and edges of the PSG, the IFG has interprocedural reaching definition and reachable use sets attached to each vertex. These indicate variable definitions that can reach these vertices and variable uses that can be reached from these vertices. This information is constructed by first performing intraprocedural data flow analysis on each routine to gather local information and then propagating the results of this analysis throughout the graph. During the propagation phase, it is important that the calling context of procedures is maintained. The provision of a new 'inter-reaching' edge summarising reference parameters that may be preserved across call sites allows the use of a two phase propagation algorithm. This eliminates the problem of the PSG where a procedure may return to an out of context call site. The authors present algorithms to enable the calculation of interprocedural definition-use and use-definition chains based on the reaching

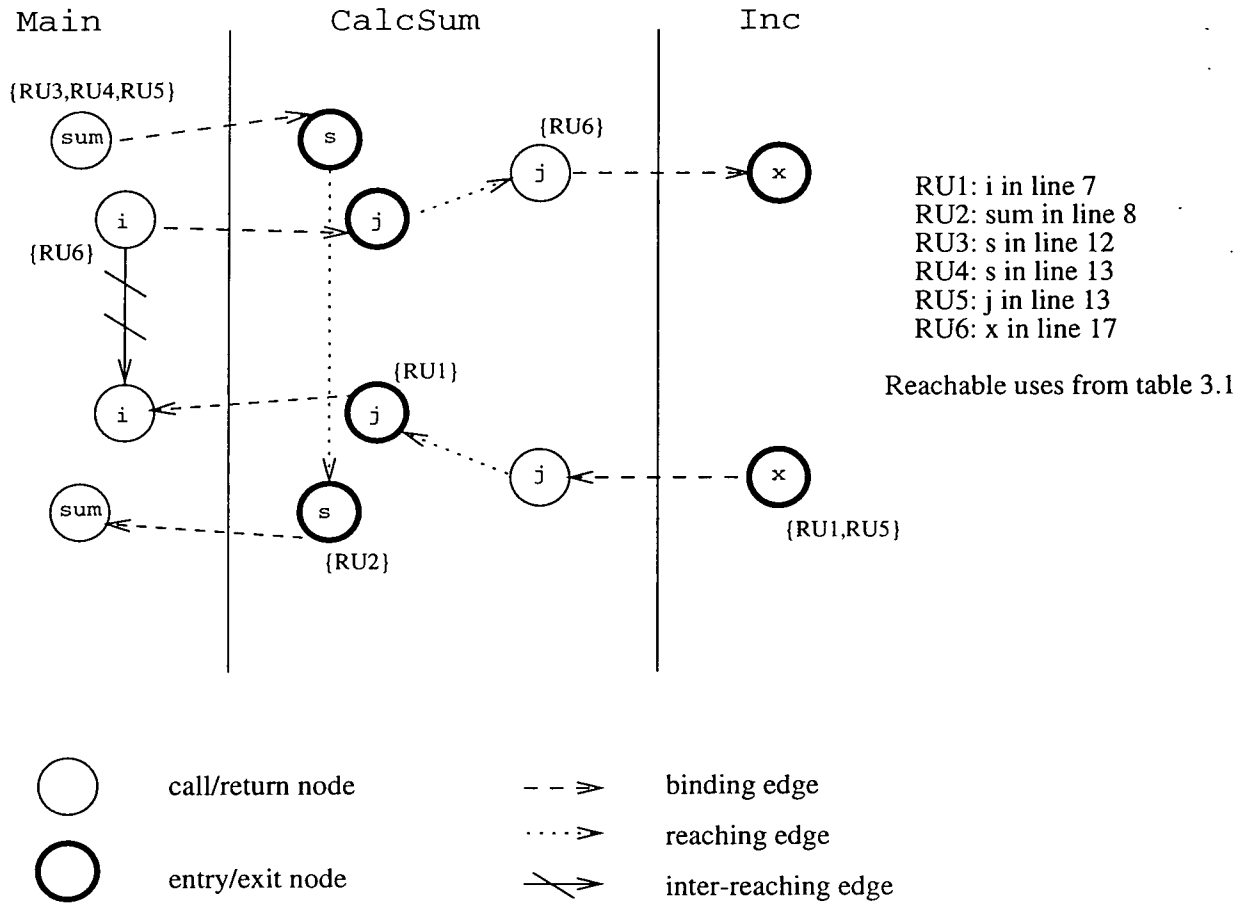


Figure 3.3: Example Interprocedural Flow Graph.

definition and reachable use sets at each node. Again reference parameters and global variables are supported but the problem of aliasing is not addressed. Space requirements for the IFG are similar to those of the PSG, that is linear in the length of the program. IFG construction has complexity $O(n^2)$. An example IFG for the program in table 3.1 is shown in figure 3.3.

The IFG contains sufficient information to allow the analysis of interprocedural data dependencies. Cross reference information is also available by gathering variable definitions and uses from the representation. The program's control dependencies are not fully represented and consequently control flows and program slices cannot be formed.

3.2.5 Program Dependence Graph

The *Program Dependence Graph* (PDG) is a dependence based IPR originally introduced by Ferrante et al[25] and also discussed by Ottenstein and Ottenstein[67] and Horwitz et al[45]. The PDG consists of vertices representing program statements and edges representing control and data dependencies. A definition of the PDG is given by Horwitz et al and has been taken as the basis for three extensions discussed in sections 3.2.6, 3.2.7 and 3.2.8. The language represented involves only scalar variables, assignments, conditionals and while loops. An 'end' statement indicates the final value of a variable. The vertices of the graph represent assignment statements and control predicates. The edges of the graph represent dependencies between program components. An example PDG for the monolithic program in table 3.2 is shown in figure 3.4. Construction of the PDG is $O(n^2)$ in complexity.

A control dependence edge is labelled true or false and indicates that whenever the predicate at the source vertex evaluates to the label on the edge, then the program component at the target will eventually be executed if the program terminates. Given the restricted language, these control dependencies reflect the nesting structure of the program and are easy to determine.

Two kinds of data dependencies are considered, *flow dependencies* and *def-order dependencies*, and are computed using standard data flow analysis techniques described by Aho et al[3]. A flow dependence edge indicates the definition of a variable at the source vertex which reaches a use at the target vertex along some path in the control flow graph. Flow dependencies are further classified into loop-carried and loop-independent dependencies. For example the flow dependence in figure 3.4 from `i = i + 1` to `sum = sum + i` is loop independent since the definition and use each occur within the same loop iteration. The dependence `i = i + 1` to `i = i + 1` is a loop carried dependence since the definition and use occur on subsequent iterations. A def-order dependence between two vertices represents definitions of the same variable at the two vertices which are both

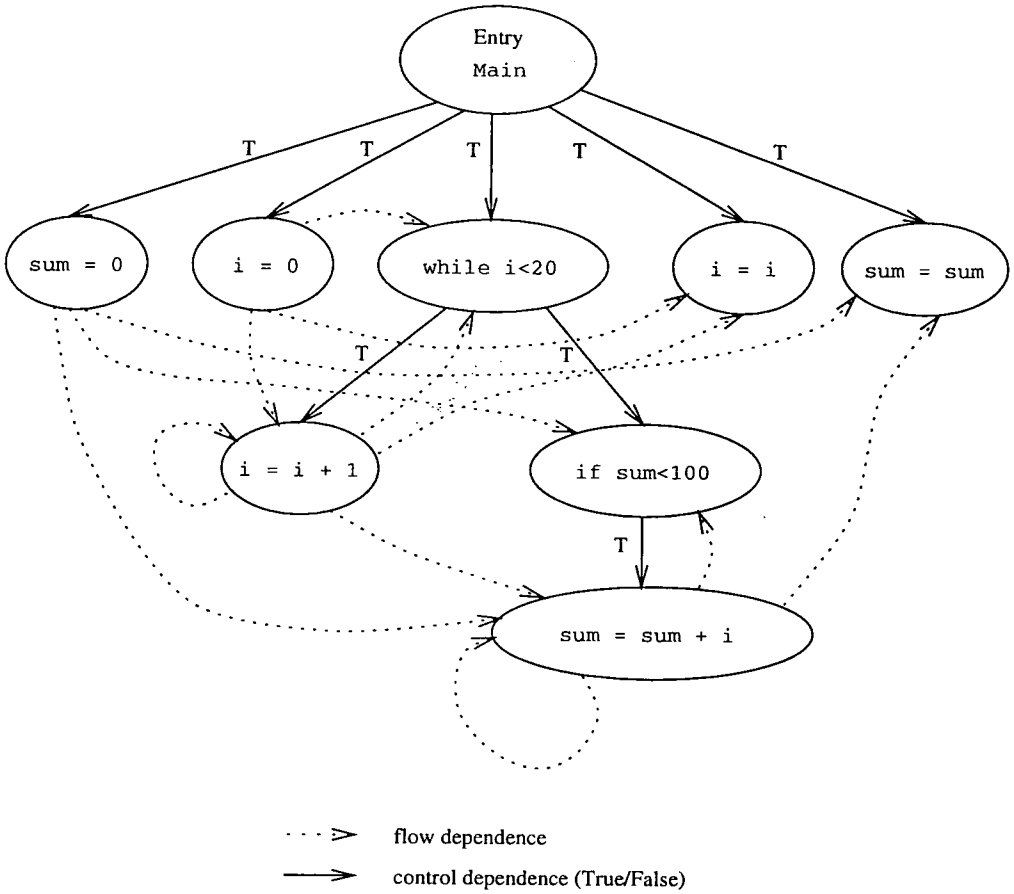


Figure 3.4: Example Procedure Dependence Graph.

```
 1.  program Main()
 2.    sum = 0;
 3.    i = 0;
 4.    while i<20 do;
 5.      i = i + 1;
 6.      if sum<100 then
 7.        sum = sum + i;
 8.      endif
 9.    endwhile
10.    i = i;
11.    sum = sum;
12.  end.
```

Table 3.2: Example monolithic program.

in the same branch of any enclosing conditional statement, with a third program component which is flow dependent on both. This dependence constrains the order in which values may be read from a variable.

Ferrante et al's[25] initial application for the PDG was as an internal representation for an optimising compiler. Various optimisations are found to operate efficiently on the PDG, for example detection of parallelism, node splitting, code motion and loop fusion. Limited incremental update of the PDG is described in response to compiler optimisations. As described in section 2.1.3, the PDG has more recently been used as a representation for the construction of program slices in linear time. A simple backwards traversal of the control and data dependencies from the required statement produces an accurate program slice. Other uses of the PDG have been described by Ottenstein and Ottenstein[67] in the calculation of program complexity metrics, by Griswold and Notkin[36] as a basis for program restructuring and by Horwitz et al[44] in the integration of program versions.

Whilst the PDG has many applications and in the area of program comprehension provides useful information in the form of control and data dependencies and program slices, the PDG in the form described is unable to model real programs.

3.2.6 System Dependence Graph

The work on slicing and PDGs by Ottenstein and Ottenstein[67] considered only single monolithic programs. Slicing across procedure boundaries using similar dependence-based methods has been tackled by Horwitz et al[45]. This involves the use of an extension to the PDG, the *System Dependence Graph* (SDG). This graph will model a similar language to the PDG, but with programs made up of a collection of procedures. Each procedure ends with a return statement and parameters are passed by *value-result.*

The SDG consists of a 'Procedure Dependence Graph' for each of the program's procedures. A non-standard two stage mechanism is used for run-time parameter passing. When procedure P calls procedure Q, values are transferred by means of intermediate 'call' temporary variables, one for each parameter. Similar 'return' temporaries are used when values are copied back on returning to P from Q. Five new vertices are required to represent this scheme:

- call site vertex - represents a call site.
- actual-in vertex - control dependent on the call site, copies the value of the actual parameter to the call temporary variable.
- actual-out vertex - control dependent on the call site, copies the value of the return temporary variable to the actual parameter.
- formal-in vertex - control dependent on the called procedure's entry vertex, copies the value of the call temporary variable to the formal parameter.
- formal-out vertex - control dependent on the called procedure's entry vertex, copies the value of the formal parameter to the return temporary variable.

The use of temporary variables forces the creation of data dependencies at the actual and formal vertices.

Three new interprocedural graph edges connect the procedure dependence graphs.

- call edge - from each call site to the corresponding procedure entry vertex.
- parameter-in edge - from reach actual-in vertex at a call site to the corresponding formal-in vertex in the called procedure.
- parameter-out edge - from each formal-out vertex in the called procedure to the corresponding actual-out vertex at the call site.

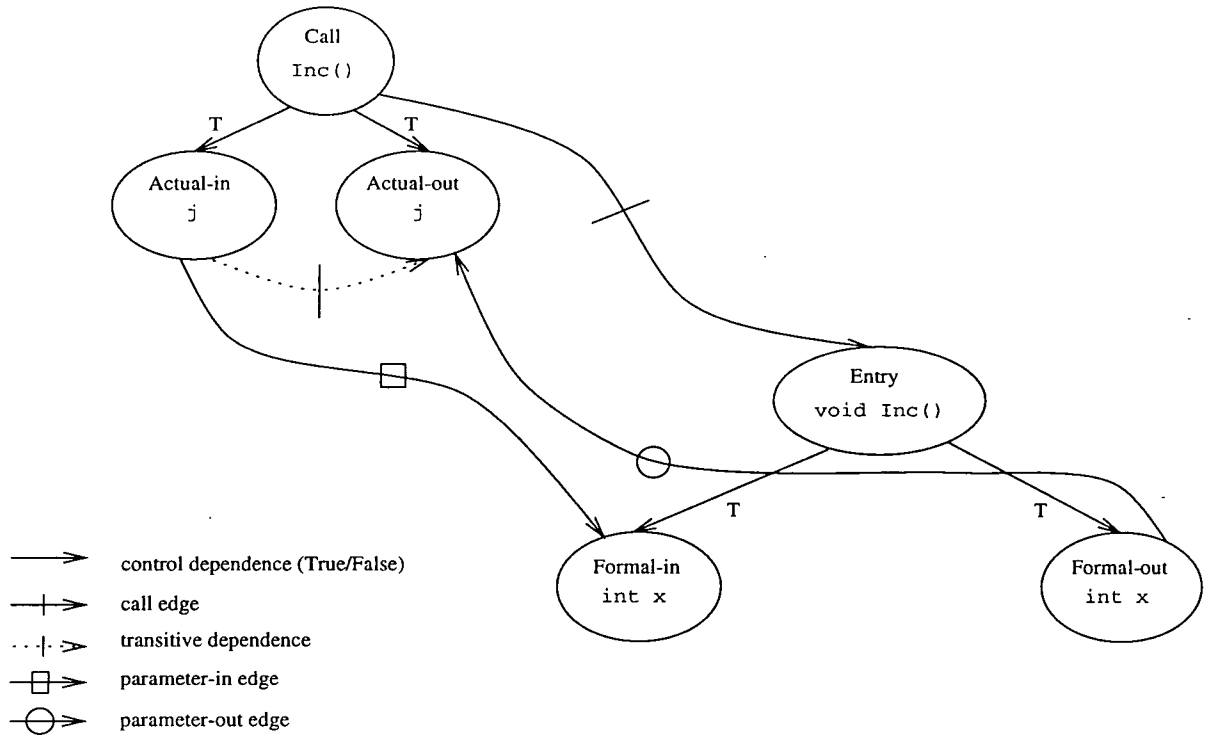


Figure 3.5: Example System Dependence Graph 'encapsulated' interface.

This model restricts data dependencies between procedures to be from actual-in to formal-in vertices and from formal-out to actual-out vertices, producing an 'encapsulated' interface between procedures, This is shown in figure 3.5

A fourth new edge, the interprocedural flow edge, represents transitive dependencies between actual-out and actual-in parameters across call sites. The edge serves a similar purpose to the inter-reaching edge of the IFG in helping to preserve procedure calling context during graph traversals. For a language without recursion, the problems of calling context can be overcome by introducing a separate copy of a procedure dependence graph for each call site. This solution is obviously undesirable for large programs. For a language with recursion, transitive dependence edges are calculated by defining an attribute grammar, the 'linkage grammar' to model the call structure of each procedure and any intraprocedural transitive flow dependencies between the parameter vertices. Interprocedural transitive edges are then found by calculating the corresponding 'subordinate characteristic graphs' of the linkage grammar's non-terminals.

Horwitz et al present a two stage interprocedural slicing algorithm making use of the interproce-

dural transitive edge, based on the graph traversal method described by Ottenstein and Ottenstein [67]. Each phase traverses only a subset of the SDGs edges to avoid ascending or descending the graph respectively.

Two methods are described by which the SDG can be adapted to call by reference parameters. The first is to effectively remove the problem by examining the aliases that may exist at each instance of a procedure call. For each different alias configuration a new procedure is created, the call site adjusted and the variables renamed so that each alias set becomes a single variable.

The second method is to generalise the definitions of data dependencies to create edges for the dependencies arising as a result of possible aliases. For ordinal types potential aliasing occurs whenever a procedure has more than one formal reference parameter or global of the same type.

Both methods have limitations. The former may lead to the creation of many procedure dependence graph copies for large programs with many aliases. The latter generalised definitions will produce many extra spurious data dependencies, giving both inaccurate program slices and inefficiencies in the usage of storage space.

Whilst simple value parameters of the C language can be adequately modelled by the SDG representation by employing a subset of the value-result mechanism, the use of pointer parameters in C presents greater difficulties. The SDG explicitly names each actual parameter at each call site, defining actual-in and actual-out vertices together with corresponding formal-in and formal-out vertices accordingly. For parameters with bounded size, for example of type `int *`, it is possible to in the same way define actual and formal vertices for the pointer and the referenced object. However, when a recursive structure appears as a parameter, it is no longer possible to define appropriate actual/formal vertices since the structure is not bounded in size.

Like the earlier IPRs considered, the language represented by the SDG is currently too restricted. For example value-returning procedures, pointer and structure variables are not tackled. Program understanding information in the form of call graphs, control and intraprocedural data dependencies and program slices may be constructed. However the absence of the 'inter-reaching' edge of the IFG means that local data flow information cannot be propagated throughout the SDG to construct interprocedural definition-use information.

3.2.7 C System Dependence Graph

Based on Horwitz et al's SDG[45], Livadas[61][62][63] describes an extended SDG allowing the handling of constructs forming a subset of ANSI C[5]. Extra features modelled by Livadas are declarations of local and global variables, pass by value parameters, pointer operations limited

```
 1. void main()                  13. int CalcSum(int s,int *j) 21. void Inc(int *x)
 2. {                            14. {                         22. {
 3.   int sum;                   15.   Inc(j);                  23.   *x = *x + 1;
 4.   int i;                     16.   if (s<100) {             24. }
 5.   sum = 0;                   17.     s = s + *j;
 6.   i = 0;                     18.   }
 7.   while (i<20) {             19.   return s;
 8.     sum = CalcSum(sum,&i); 20. }
 9.   }
10.   i = i;
11.   sum = sum;
12. }
```

Table 3.3: Example C program.

to one level of indirection (i.e. `*x`), arbitrary return statements, value-returning functions and control constructs such as `switch`, `for` and `do..while`. Structures are decomposed into primitive components, globals represented as extra reference parameters and static variables represented as 'limited-scope globals'.

Livadas' SDG is based on the program parse tree, i.e. the graph's vertices represent nodes in the parse tree rather than statements of the program. Three new edge types are introduced.

- affect-param edge - represents dependence between an actual parameter and the function's return value.
- return-link edge - represents dependence between the return nodes in the function and a corresponding call site.
- return-control edge - indicates the dependence between the return statement and other following statements not executed when the function exits on the return statement.

Livadas tackles the aliasing problem using the alias-removing transformations/procedure copying method described by Horwitz et al[45]. A C System Dependence Graph representation of the program in table 3.3, a C version of the program in table 3.1, is shown in figure 3.6. The graph shown is, for simplicity, resolved at the statement level rather than the token level.

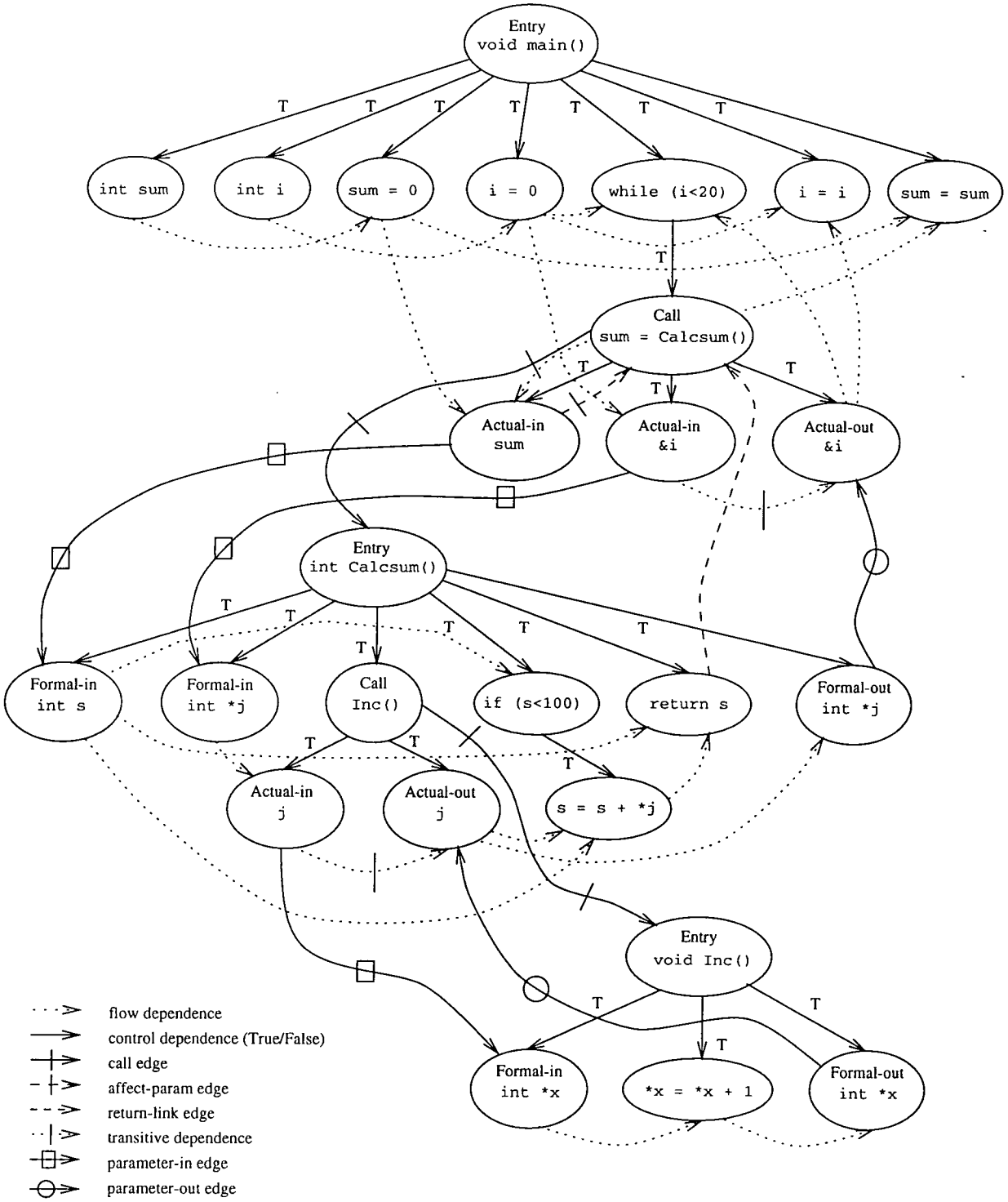


Figure 3.6: Example C System Dependence Graph.

A program slicing algorithm similar to the two phase algorithm described by Horwitz is applied to the new SDG. Enhanced slicing accuracy is reported via two means. Firstly, by determining during SDG construction whether an actual parameter is never, sometimes or always modified during procedure execution, the architecture of the SDG is adjusted accordingly. Secondly by resolving the SDG nodes at the token level, in certain cases smaller and therefore more accurate slices will result. For example slicing on variable i at statement 4 of the program fragment

```
1. i = 0;
2. sum = 0;
3. sum = sum + Alpha(&i);
4. a = i;
5. b = sum;
```

produces the slice

```
1. i = 0;
3. Alpha(&i);
4. a = i;
```

By employing a finer-grained SDG, Livadas is able to omit the dependencies involving sum at statement 3 from the resulting slice.

Livadas' SDG is constructed using a new bottom-up method in which only one copy of a procedure dependence graph is required for any number of recursive call sites. The algorithm is conceptually simpler than that employed by Horwitz et al, no longer requiring the use of attribute grammars and subordinate graphs to determine transitive dependencies.

Livadas describes the Ghinsu environment[61] which currently provides program slicing, inter-procedural definition-use analysis, call graph and data flow dependence information, together with an object finding mechanism[63] based on the SDG IPR.

Livadas' SDG is closer to the representation of real programs than the other IPRs considered so far, although the language covered still omits some features of the C language. In particular restriction of pointers to only one level of indirection means that Livadas keeps the encapsulated procedure interface of Horwitz' SDG and simplifies data dependence analysis. It is not clear how this representation can be extended to deal with arbitrary pointer parameters and pointer assignments.

Livadas has made good progress in the areas of graph construction and program slicing. The use of a 'refined' dependence-based representation to allow more accurate slicing is a new and beneficial

approach although will lead to less space-efficient graphs, causing problems in the representation of large-scale programs.

3.2.8 Unified Interprocedural Graph

Given the applicability of the IFG to interprocedural data flow calculations and of Horwitz' SDG to program slicing, Harrold and Malloy[38][39] identify the possibility of merging the two representations to create the *Unified Interprocedural Graph* (UIG). Numerous redundancies exist between the vertices and edges of these two graphs. The call/return vertices of the IFG are equivalent to the actual-in/actual-out vertices of the SDG. Similarly, the entry/exit vertices of the IFG are equivalent to the SDG's formal-in/formal-out vertices. Finally, the IFG's binding edges are equivalent to the parameter-in/parameter-out edges of the SDG. These redundancies are eliminated in the UIG and consequently savings made in terms of space. A UIG representation of the program in table 3.1 is contained in figure 3.7.

The algorithms designed to operate on each component subgraph remain applicable to the UIG by simply considering only subsets of the available edges and vertices. The provision of the inter-reaching edge of the IFG means that intraprocedural reaching definition and reachable use sets can be propagated to allow the calculation of interprocedural definition-use associations, whilst the control information of the SDG permits interprocedural program slicing. Flow-insensitive data flow analysis is also possible given that the call graph of the program is represented by the procedure entry vertices and call edges of the SDG. The complexity of these algorithms remains identical to the complexity of the algorithms when applied to the original graphs. Given that only a single representation need be accessed, the authors claim that savings can be made in the access times of these algorithms.

The UIG provides many different views of a program yet suffers from the same limitations as Horwitz' SDG in the representation of C programs. The language modelled by the UIG is currently too restricted and the addition of features such as arrays, structures, pointers and value-returning procedures is required. The methods employed for parameter passing must also be addressed, the UIG suffering from the same limitations as the SDG in the representation of recursive structure parameters used in C. Harrold and Malloy nevertheless present a useful approach and starting point for the development of a unified IPR for a real programming language. The benefits of having a single representation make this a good approach for the development of a multiple program view maintenance environment.

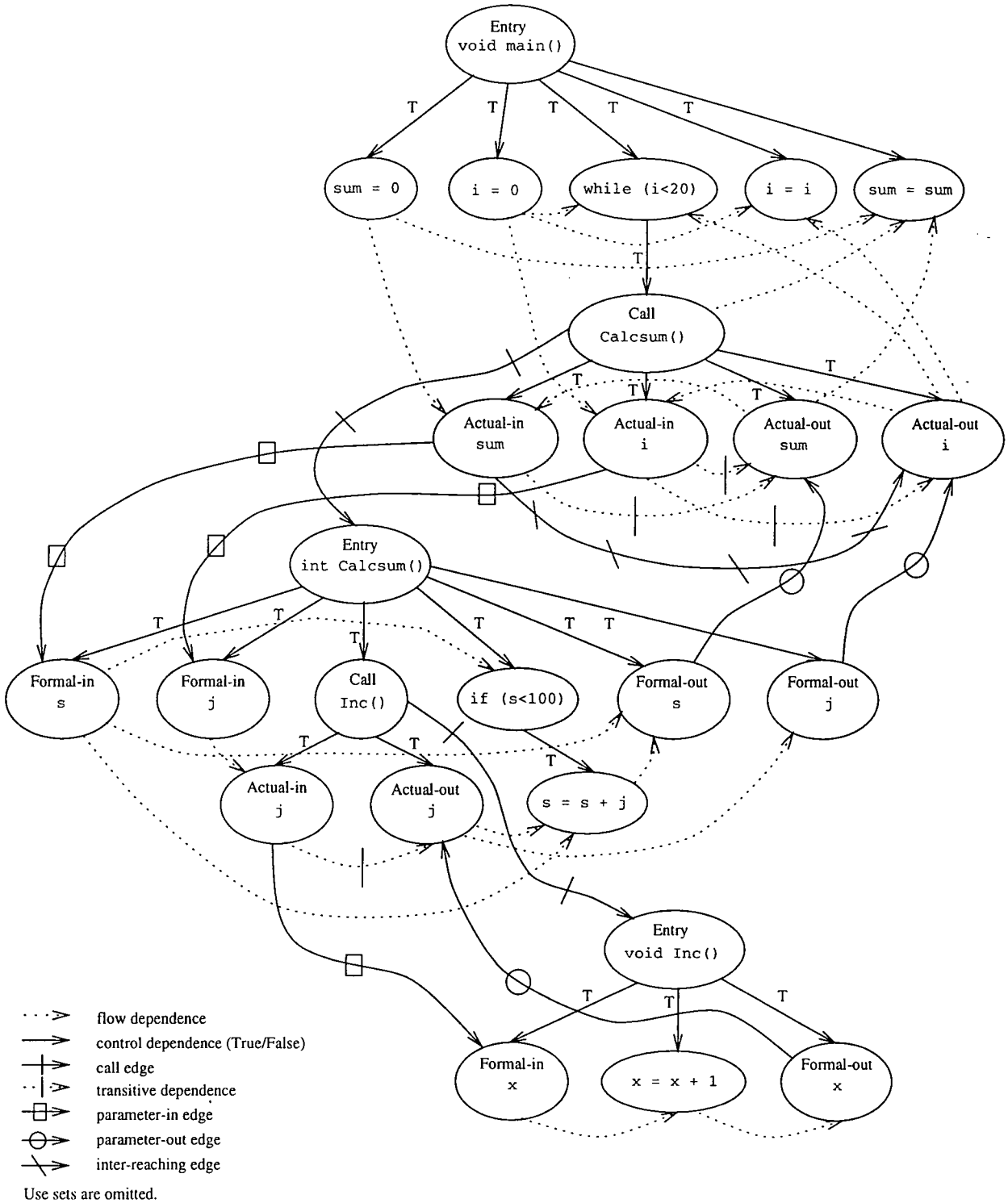


Figure 3.7: Example Unified Interprocedural Graph.

3.2.9 Maintainer's Assistant Program Representation

Platoff et al[72] describe the *Maintainer's Assistant Program Representation* (MAPR), an IPR designed to provide source code, architecture, syntax, static semantics, control and data flow views of C programs. MAPR consists of abstract syntax graphs, cross reference graphs, control flow graphs and data flow graphs, allowing the implementation of tools such as restructuring tools, impact analysers, architecture analysers and testing tools.

Abstract syntax graphs are the foundation of MAPR. C preprocessor information is maintained within the graphs and annotations indicate expression types and the presence of side effects. Cross reference information is implemented as attributes on the vertices of the abstract syntax graphs with identifier declarations linked to subsequent uses. The control flow and data flow graphs are similarly linked to the abstract syntax graph by pointers between associated vertices. Data flow analysis employs the techniques of Landi and Ryder[54][55] and hence provides approximate solutions in the presence of multi-level pointers.

Although the representation is integrated by simple links between its components, the approach is different to that of the UIG. MAPR is essentially a collection of linked representations rather than a single representation with an associated set of view-forming algorithms. MAPR may be less efficient in its use of storage space since information will be repeated between the MAPR subcomponents.

The authors claim that MAPR is able to represent all of the C language but no formal definition of the representation or examples are given. Data and control dependence information are not represented explicitly in MAPR and hence traversal-based program slicing algorithms cannot be applied. No mention is given to any program slicing tool based on MAPR. The specific construction costs and space requirements of MAPR are not described.

3.2.10 Standard Representation of Imperative Language Programs

Bieman et al[9] identify the problem that many software tools and measures are defined in terms of different program representations, such as the source code, program flow graph or data dependencies. In many cases algorithms are defined in a language dependent fashion and are hence difficult to compare. Bieman et al's solution to this problem is a standard language independent representation protecting the program's semantics and containing sufficient information to permit control flow and data dependence analysis.

Hiding program semantics by encoding variables is suggested as a way to gain access to propri-

etary software in order to validate tools and measures on large projects. The language independent representation may be constructed from any imperative language - a translator must be made available to map the language to the standard representation whilst tools and measures are defined in terms of the representation itself.

The standard representation, like the UIG, incorporates control flow, data dependencies, procedure interfaces and the use of operators but unlike the UIG is not a graph-based representation but is defined using a formal specification language. Using the standard representation Bieman et al define the computation of the basic software science measures (i.e. number of distinct operators and distinct operands, total number of operator and operand occurrences), the cyclomatic number and information flow measures.

The authors recognise similar problems to Harrold and Malloy[39] but coming from a metrics background concentrate more on the computation of measures. Program comprehension based on the standard representation will require other program views such as program slices or data flow information. Translators from languages such as C may also be difficult to construct, particularly the less restrictive use of pointer variables.

3.3 Data Dependence Analysis of Pointer Variables

When statically analysing a program the effect of aliasing must be taken into account. The precision by which these effects can be determined will be a significant factor in the usefulness of the static analysis. This is especially true of data dependence analysis, where the presence of aliasing creates additional dependencies. Imprecise aliasing information will lead to further spurious dependencies.

The *flow dependence* is an important component of each of the dependence-based IPRs described earlier. For a language without aliasing Horwitz et al[45] define a flow dependence from vertex v_1 to vertex v_2, $v_1 \rightarrow_f v_2$, to exist when:

- v_1 is a vertex that defines variable x.
- v_2 is a vertex that uses variable x.
- Control can reach v_2 after v_1 via a path in the control flow graph along which there is no intervening redefinition of x.

This corresponds to the computation of a definition-use association from v_1 to v_2. Simple algorithms to calculate definition-use associations are described by Aho et al[3] and may be employed in the construction of SDG and UIG intraprocedural flow dependence edges.

For languages without pointer variables, aliases can be created through the use of reference parameters and non-local variables. An alias will result in procedure P whenever a call is made to procedure P of the form $P(x, x)$, where the actual parameter is repeated, or $P(g)$, where g is a non-local variable accessed within P. Further aliases can then be created within procedures called from P by calls such as $Q(a, b)$, where a and b are aliases within P. Any alias created will hold throughout the execution of a callee. Horwitz et al extend the original definition of flow dependence to deal with 'potential aliasing' which exists in a language with reference parameters whenever a procedure has two or more parameters of the same type, two or more non-local variables of the same type or a parameter and non-local variable of the same type:

- v_1 is a vertex that defines variable x.
- v_2 is a vertex that uses variable y.
- x and y are potential aliases.
- Control can reach v_2 after v_1 via a path in the control flow graph along which there is no intervening definition of x or y.

More accurate alias analysis in the presence of reference parameters and global variables has been described by Banning[8] and Cooper[21].

The definition presented by Horwitz et al for definition-use associations in the presence of aliasing relies on the fact that an alias holding when a variable is defined will also hold when that variable is used. There is a direct relationship between variable names and memory locations which does not change intraprocedurally. This approach, although not discussed by Harrold and Malloy[38][39], could be applied to the reference parameters of the UIG.

In C, assignments between pointer variables allow aliases to be created intraprocedurally and allow the aliases holding in a caller to be affected by assignments within a callee. Aliasing information can no longer be calculated for an entire procedure but must be updated after each pointer operation. The presence of self-referential structures further complicates the aliasing problem. Linked data structures may be potentially unbounded and must be represented in some finite way. More precise aliasing solutions are required if useful data dependence information is to be achieved.

Recent work by Landi and Ryder[54][55] presents an algorithm to approximate safely interprocedural pointer-induced aliasing, based on the use of conditional analysis techniques - assuming an alias pair exists at entry to a procedure P, then can variables x and y be aliased at statement n within P? A *may-hold* predicate indicates whether this is true or false for a given statement, an

assumed alias pair and a pair of variables x and y. A *must-hold* function indicates the set of aliases that must exist at a procedure entry point to ensure an alias pair exists at a statement n in the procedure P.

Pande et al[68] extend this work to give an approximate algorithm for obtaining interprocedural definition-use associations in the presence of single level pointers. Pande first calculates the interprocedural reaching definitions, again using the conditional analysis technique. The aliasing information computed by Landi and Ryder is then used to account for the generation and killing of reaching definitions

The point specific aliases are said to allow sufficient accuracy for the definition-use information to be useful to a maintenance programmer, although it appears that the work has not yet been applied to large scale C programs.

The compiling community is a source of research into the data dependence analysis of pointer variables. By modelling the effects of dynamic variables, possible sources of parallelisation can be determined. A method for the calculation of data dependencies for programs with pointers and heap allocated storage is presented by Horwitz et al[42]. Horwitz et al address the reaching definitions problem in terms of memory locations, rather than variable names and aliases:

> Program-point q has a flow dependence on program-point p if p writes into a memory location *loc* that q reads, and there is no intervening write into *loc* along the execution path by which q is reached from p.

Horwitz et al's algorithm is divided into two phases. The first phase, the 'reaching-stores phase', computes at each program-point a set of store graphs that approximate the possible memory layouts that could arise during execution. Program variables, together with any dynamic variables allocated during execution, are represented by abstract memory locations. A store graph consists of vertices representing abstract memory locations and edges representing pointer relationships between the abstract locations. An initial store graph is iterated around the program's control flow and call graph until a fixed-point solution is achieved. The contents of each abstract location and pointer relationships between abstract locations are updated at each program-point to reflect the semantics of the statement. Each abstract memory location is labelled by the program-point which last wrote to that location.

Three approximations are used to ensure that the set of store graphs at each program-point is effectively computable whenever the program contains a loop.

- A 'one-element domain' of atoms prevents state sets from differing only on the value of an

atom.

- A 'condense' operation limits store size by replacing a non-empty region of a store with a single vertex. This *k-bounding* limits acyclic paths within self-referential data structures to depth k.

- A 'collapse' operation uses an equivalence relation to limit the size of a set of states.

The second phase, the 'inference phase', examines the set of stores reaching each program-point and determines the locations read. A flow dependence $p \rightarrow_f q$ exists if q reads a location labelled p in any store graph reaching q.

Horwitz et al's algorithm is defined using an abstract semantics and only a small practical Lisp example is described. Use of the algorithm with C programs will require a method to deal with the more complicated dynamic allocation statements and a means of approximating the iteration throughout the control flow/call graph during the reaching stores phase. The number of possible control paths through a large program can preclude *all-paths* analysis, hence some method of path limitation may be required.

A similar pointer/structure dependency analysis algorithm is described by Chase et al[17]. A 'storage shape graph' (SSG) is constructed at each statement of the program. An SSG consists of vertices representing simple variables and heap-allocated storage with a special vertex representing all atoms. Edges in the SSG represent pointer relationships.

Chase et al's basic algorithm produces at each statement SSGs containing the same vertices, one for each simple variable and one for each statement in a program allocating a data structure. Initially each SSG has no edges. The analysis proceeds by iterating throughout the program's call/control flow graph in the same way as Horwitz et al. At each statement, edges are added to the SSG to represent the semantic effects of the statement. When a fixed point is reached, edges in an SSG represent an approximation to the possible pointer relationships into and through allocated storage, that could arise by executing any path to the statement for which the SSG was computed. The most accurate solution occurs whenever an SSG comprises the least number of edges that is still a conservative approximation to actual storage.

An extension to the algorithm allows multiple vertices within an SSG for each dynamic allocation statement. These vertices may be merged under certain conditions. This operation is the equivalent to Horwitz et al's k-limiting approximations. The k-limiting method has two main limitations. Firstly, information on a structure beyond depth k is lost, Secondly, unless k is small, k-bounded approaches can be inefficient. Chase et al's merging operation maintains more structural

information and allows the detection of acyclic structures such as lists and trees. By never merging vertices from different allocation statements, Chase et al assume that heap space allocated at the same statement is liable to be treated in the same way. For a type-based language this is equivalent to never merging vertices that could be given different type definitions.

Like Horwitz et al, Chase et al describe only a small Lisp-based example and the control paths problem still exists. The actual computation of data dependencies is also not described, only the computation of program aliases.

3.4 Summary

This chapter has evaluated a number of intermediate program representations, selected from the fields of data flow analysis and program slicing. The common limitation of each representation is the language modelled. In most cases only 'toy' languages are addressed. Harrold and Malloy's UIG[38][39] allows the creation of a variety of program views but is unable to model many features of the C language. Livadas' refined SDG[61][62][63] allows more accurate program slicing, but is unable to deal with the arbitrary pointer usage of C.

Algorithms have been described for the data dependence analysis of languages with pointer variables, self-referential structures and heap-allocated storage. The algorithms are largely theoretically based and have yet to be demonstrated with large programs. However, useful information can be achieved in the data dependence analysis of C using these methods.

Chapter 4

Combined C Graph

This chapter describes the Combined C Graph (CCG), a dependence-based intermediate program representation (IPR) allowing the modelling of many features of the C language. A brief introduction to some of the features of the C language is given in section 4.1. The approaches taken in the development of the CCG are described in section 4.2, which addresses the additional language features modelled and outlines the vertices and edges comprising the CCG. Using earlier dependence-based intermediate program representations as a basis, embedded side effects and control flows, pointer parameters, value-returning functions, structures, arrays and pointer variables are introduced. The CCG representation makes available to the software maintainer a variety of programming level views. Section 4.3 goes on to describe these views and their construction from the CCG. A more formal description of the vertices and edges of the CCG is given in section 4.4, whilst algorithms for the construction of the CCG are contained in section 4.5. Section 4.6 finally contains an example CCG for a small C program.

4.1 The C Programming Language

The C Programming Language[51] was originally designed and implemented in 1972 by Dennis Ritchie on the DEC PDP-11 for the UNIX operating system. The language has its history in the BCPL language developed by Martin Richards and the B language designed by Ken Thompson in 1970. The language is a general purpose programming language and although useful as a language for writing compilers and operating systems has grown in popularity and been used to write large systems in many application domains. C is a concise, small language which contains a mixture of low level assembler-style commands together with higher level commands. For many years the *de facto* standard for C was taken from the first edition of Kernighan and Ritchie's *The C Programming*

Language[50] and known as 'K&R' C. In 1988, the American National Standards Institute completed a standard definition for the language, known as 'ANSI C'[5]. The two standards differ only slightly, the most important changes being the provision in ANSI C to describe function arguments and the definition of a standard library. The following subsections give a brief flavour of the C language, describing features which add to the simpler declarative languages modelled by other IPRs.

4.1.1 Variables and Types

C is a typed language. The basic types available are characters, integers and floating point numbers of different lengths. From these basic types it is then possible to derive other data types using pointers, arrays, structures and unions. For example variables of the same type can be put into arrays:

```
char months[12];
```

Multi-dimensional arrays can also be created:

```
char coords[50][100];
```

Variables of different types can be grouped into structures. For example:

```
struct address {
  int number;
  char street[30];
  char town[20];
} my_house;
```

The `typedef` operator can be used to create new types. Union variables are declared in a similar way to structures but memory is only assigned for the largest item. The programmer must keep track of what the union is being used for. For example:

```
union person {
  int age;
  float height;
} bill;


bill.age = 30;
bill.height = 1.71;
```

Pointer variables are widely used in C. The * operator is used to dereference a pointer variable. The & operator returns the address of a variable. A pointer is constrained to point to a given type. Pointers to structures are commonly used, particularly in connection with dynamic memory allocation, to create recursive structures such as lists or trees. The -> operator is provided as an abbreviation to access the fields of a referenced structure. For example, `(*address_ptr).street` becomes `address_ptr->street`.

Pointers and arrays in C are closely related. Operations making use of array variables may also be carried out using pointer variables. For example, the notation `month[3]` refers to the fourth element of the array `month`, array subscripts starting at 0. By declaring a pointer variable `char *p` and assigning `p = &month[0]`, the pointer `p` now points to the start of the array `month`. The equivalent notation `*(p + 3)` can then be used to access the fourth element of the array. The compiler ensures that the pointer is incremented in relation to the type and size of the referenced object.

Another use of pointer variables is to create pointers to functions. A pointer to a function can be used like any other variable.

4.1.2 Expressions

An expression in C is constructed from operands and operators. C has many different operators - arithmetic operators, increment and decrement operators, boolean operators, bitwise operators, relational operators and assignment operators. Any expression can be used as a statement in the language. For example, the expression `x = y` is a statement which assigns the value of `y` to `x`. All expressions return a value, which in the case of assignment is the value assigned to the left hand side argument. By combining expressions it becomes possible to create larger expressions containing embedded side effects. For example, the statement:

```
x = (y + z++);
```

will have the side effect of incrementing `z` in addition to assigning a new value `(y + z)` to `x`.

The order of evaluation of sub-expressions is unspecified with the exception of `&&` (logical and), `||` (logical or), `?:` (conditional operator) and `,` (the comma operator).

4.1.3 Control Flow

The C language has a wide variety of control constructs. In addition to `if` statements, `while` and `for` loops, C provides the `switch` multi-way decision statement and the `do` loop, which tests at the

bottom of the loop. The non-structured `break` statement enables early exit from the innermost enclosing loop or `switch` whilst `continue` causes the next iteration of any enclosing `for`, `while` or `do` loop to begin. Finally a `goto` statement is also provided.

In addition to these control constructs, control flow can also exist within expressions. The boolean operators `&&` (logical and) and `||` (logical or) each make use of short circuit evaluation. Hence a boolean expression such as `a && b && c` will contain control flows between the sub-expressions `a`, `b` and `c`. Similarly the conditional operator `?:` creates control flows between its sub-expressions. For example the expression:

```
(i > 100) ? i++ : i--;
```

evaluates either `i++` or `i--` dependent on the outcome of `(i > 100)`.

4.1.4 Functions

C provides no procedures, only functions. Functions may not be nested but with the exception of `main` (the start function) may be called recursively. Execution of a function continues until the end of the function is reached, a `return` statement is reached, or an `exit` statement is reached terminating the program. The `return` statement is used to return a value to the calling function. If no `return` statement is present, the value returned is undefined.

Parameters are passed to a function using the call by value mechanism. The actual parameter value is copied to the formal parameter and therefore cannot be modified by changes to the formal. To modify an actual parameter, the programmer must pass a pointer value. The called function can then dereference the formal parameter to access the object to be modified.

4.1.5 External Variables, Scoping Rules and Block Structure

The C language has complex scoping rules which become of great importance with larger programs for which the source is kept in several files or indeed in separately compiled libraries.

Variables defined within a function may be accessed only within that function. These local variables come into existence when the function is called and disappear when the function is exited and are known as *automatic* variables. An alternative is to define variables which are external to all functions. These variables remain in existence permanently. A variable defined in global space may be accessed within a function by declaring the variable inside the function using an `extern` statement. For example an externally defined integer `i` may be declared and then accessed within a function by declaring the variable within the function using `extern int i`. Space for the variable

is however only assigned by the single external definition.

Preceding an external variable definition with the keyword `static` limits the variable's scope to the rest of the source file being compiled. Variables local to a function may also be defined as `static`. Static internal variables remain in existence and preserve their value between function calls.

It is possible to define variables in C in a block-structured manner, despite the absence of nested procedures. Variables can be declared at the start of any compound statement such as a `while` or `if` statement. Any variables declared within an inner block in this manner hide any similarly named variables defined in outer blocks or as `extern`.

Variables may also be initialised as they are defined. For example, the following statement defines the integer `i` and initialises it to the value 100:

```
int i = 100;
```

4.1.6 Standard Library

The C language is provided with a set of standard library routines. This library provides functions to perform tasks such as input and output, mathematical routines, string handling and storage management. The standard library is not part of the C language as such but the definition provided by the ANSI standard allows for compatibility and ease of porting.

4.1.7 C Preprocessor

Facilities such as file inclusion, macro substitution and conditional inclusion are also not part of the language itself but are provided by the C Preprocessor. The preprocessor forms a separate grammar to the language itself and is implemented as a first stage in compilation. The most commonly used features are `#define` to replace a token with a given character string and `#include` to include the contents of another file.

4.2 CCG Overview

This section describes the approaches taken in the development of the CCG to allow the modelling of many features of the C language. An earlier IPR, the UIG described by Harrold and Malloy[38][39], provides useful information yet is unable to model C features such as embedded side effects and control flows, pointer parameters, value-returning functions, structures, arrays, pointer variables or

C control constructs such as `break`, `continue`, `switch` or `goto`. The CCG representation refines and extends the UIG to enable the modelling of each of these constructs and language features[52][53].

The CCG is a dependence-based IPR containing explicit representations of the program's data and control dependencies.

The CCG is a directed graph comprising dependence graphs for each function making up the subject C program. These subgraphs are known as Function CCGs (FCCGs). The FCCGs making up a CCG are connected by a variety of interprocedural graph edges representing call relationships, parameter passing and interprocedural data dependencies.

Each FCCG is a directed graph where, at the most basic level, vertices represent the statements of the program, such as assignments and control predicates. Each FCCG has a special 'entry' vertex which is the root of the directed graph representing the body of the function. Refinements to this basic model are introduced to represent expressions with embedded side effects or embedded control flows and are discussed in detail below.

The edges of an FCCG represent the control and data dependencies between the statements. A control dependence edge indicates that the execution of the statement at the sink vertex of the edge is determined by the execution of the predicate at the source vertex of the edge. For C programs involving only `if..then..else` and `while` control constructs, the control dependence edges of an FCCG will reflect directly the nesting structure of the control predicates. Each statement immediately nested within the loop or conditional whose predicate is at vertex v will be control dependent on v. Control dependence edges are labelled 'true' or 'false' indicating the outcome of the predicate at the source vertex. Control dependencies for other C control structures are discussed in section 4.2.1.

A data dependence edge between two vertices indicates that a program's computation may change if the relative order of the two vertices were changed. The data dependence edges of the CCG are more accurately flow dependencies. A flow dependence corresponds to a definition-use association from the source to sink vertex. A variable defined at the source vertex is used at the sink vertex with no intervening redefinition.

Each FCCG has annotations describing the solutions to three flow-sensitive data flow analysis problems:

- *May be preserved* - variables whose value is maintained unchanged on some path through a function.
- *Live on entry* - variables used on some path through a function before being redefined.

- *Live on exit* - variables which are used on some path on leaving a function before being redefined.

Variables involved are either objects referenced by pointer parameters or external variables. Sets representing the solutions to these problems are attached to each FCCG.

4.2.1 Embedded Side Effects, Embedded Control Flows and Value-returning Functions

Side effects occur when a variable is altered during the evaluation of an expression. In C, side effects can arise as a result of assignment statements, increment and decrement operators and function calls. Wherever a statement may contain an expression, side effects are possible. For example, the variable y is defined as a side effect of the test:

```
if (x == (y = 5)) ...
```

The increment of variable i in:

```
while (a[i++] == 0) ...
```

similarly is a side effect. A function call involves side effects if any variables are defined during the execution of the function. For example:

```
if (x == f()) ...
```

may involve the definition of variables within function f.

As a result of embedded side effects, data dependencies can exist between the individual sub-expressions of a program statement. For example, the statement:

```
y = f() + x;
```

may lead to data dependence from any definition of x within f, where x is a global variable, to the use of variable x in the addition expression.

An expression in C may also contain embedded control flow. This occurs with the conditional expression operator ?:. The conditional expression:

```
(a > b) ? a : b;
```

evaluates either a or b depending on the outcome of (a > b). The use of short-circuiting in evaluating boolean expressions similarly leads to embedded control flow. For example, in the expression:

```
if (x && y && z) ...
```

if x is false, the value of the entire expression is false and y and z will not be evaluated. There is consequently a possible change in control flow associated with the && (logical and) and || (logical or) operators and hence control dependencies between the sub-expressions involved.

A dependence-based representation resolved at the statement level will be unable to model these embedded side effects and control flows. Data and control dependencies which cannot be represented at the statement level will exist between sub-expressions.

A finer-grained representation is necessary and has two advantages:

- A maintainer is presented with a more intuitive representation of the code.
- As described by Livadas[61][62][63], it becomes possible to construct more accurate program slices.

The solution presented by Livadas is to resolve the IPR at the parse tree level, such that each vertex of the IPR represents a node in the parse tree. Whilst this undoubtedly provides fine grained information, the number of program vertices required will become large.

The approach taken in the CCG is to wherever possible resolve the representation at the statement level. However, where embedded side effects or control flows exist, extra vertices are created for each sub-expression containing a side effect or possible change in control flow. This approach gives two benefits.

- The CCG contains refined information at the sub-expression level.
- The number of graph vertices is reduced.

For example, the statement:

```
a = b;
```

will produce only a single program vertex.
However the statement:

```
a = b++;
```

will produce two vertices, representing b++ and a =.
A function call followed by an assignment:

```
x = f();
```

gives two vertices `call f()` and `x =`.

The conditional expression:

```
a > b ? c : d;
```

gives three vertices `a > b`, `c` and `d`.

A boolean operator produces a vertex for each operand. For example:

```
a && b;
```

produces two vertices `a` and `&& b`.

More complicated expressions combining side effects, function calls or control flows are broken down to separate the relevant sub-expressions. For example:

```
*y++ = *x++ + f() + (z = a && b);
```

gives seven vertices. These are `a`, `&& b`, `z =`, `call f()`, `*x++`, `*y++` and a final assignment vertex.

Three new edges are introduced to connect the vertices created when sub-expressions involve side effects, function calls or control flows. The first of these edges is the *expression-use edge.* This edge connects a sub-expression vertex evaluating a value which is then used at another program vertex. The statement:

```
x = (y = 5);
```

will produce two vertices, one to represent the embedded side effect `y = 5` and the other to represent the final assignment `x =`. This assignment uses the value produced by the expression `y = 5`. An expression-use edge from `(y = 5)` to `(x =)` indicates this relationship. Figure 4.1 contains an expression-use edge representing the use of the expression `y = 5` at an assignment node `x =`.

An *lvalue* is an expression referring to a named region of storage, its name being derived from the assignment statement, the left hand side of which must be an lvalue. The second new edge, an *lvalue-definition edge* is added whenever a program vertex evaluates an lvalue expression which is then defined at a second program vertex. This situation arises when side effects occur on the left hand side of an assignment statement.

The C language, with the exception of boolean, comma and conditional operators, imposes no order of evaluation rules on expressions. The expression:

```
*p++ = 100;
```

may as a result be evaluated as follows.

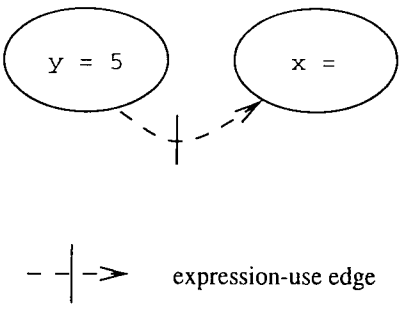


Figure 4.1: Expression-use edge.

1. Evaluate the lvalue `*p`.
2. Increment the pointer `p`.
3. Evaluate the right hand side expression `100`.
4. Assign the value of the expression `100` to the lvalue `*p`.

An lvalue definition edge from `(*p++)` to `(= 100)` will result. This is shown in figure 4.2.

The third new edge type is the *return-expression-use edge*. This edge allows the representation of value-returning functions, indicating the relationship between the expression evaluated at a return statement and its use within any calling function. Return-expression-use edges pass from each vertex evaluating a return value to each vertex at which that value may be used. For example:

```
x = f();
```

where `f` contains:

```
return a * b;
```

will produce a return-expression-use edge from `(return a * b)` to `(x =)`. This edge is shown in figure 4.3.

4.2.2 Parameter Interface

The C language uses the pass by value parameter mechanism. Pointer parameters are used to effect call by reference. A new parameter interface is required between the function subgraphs to represent each of these schemes.

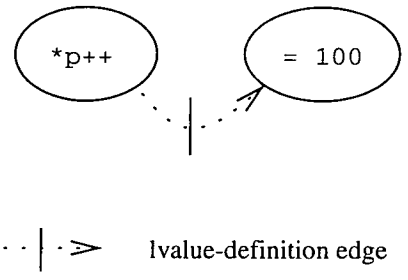


Figure 4.2: Lvalue-definition edge.

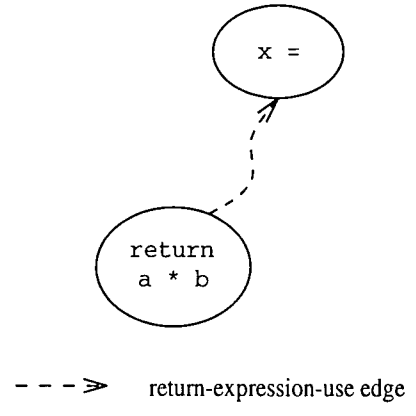


Figure 4.3: Return-expression-use edge.

Ordinal types

The pass by value parameter scheme used by the C language requires a new parameter interface to represent procedure calls and to connect the FCCGs. As seen in section 3.2.6, the SDG models pass by value-result parameters, creating a graph vertex for every actual and formal parameter at each procedure call, entry, exit and return point. The vertices created are known as *actual-in*, *formal-in*, *formal-out* and *actual-out* vertices respectively. The UIG uses the same scheme to model pass by reference parameters in the absence of aliasing.

Since pass by value is a subset of pass by value-result, pass by value parameters can be represented using a subset of these vertices, i.e. only the actual-in and formal-in vertices. These are known simply as *actual* and *formal* parameter vertices. The actual-out and formal-out vertices do not play any part in pass by value.

Pointer parameters

Pointer parameters present more complex problems. The SDG/UIG representation creates vertices for each reference parameter, which can always be enumerated and remain the same on each path to a given call site. For example, the call `P(a, b)` always requires actual-in vertices for the two reference parameters `a` and `b`.

The use of pointer parameters in C means that the actual parameters at a call site can no longer be represented explicitly. Where a pointer is passed as a parameter, the objects referenced by the pointer may no longer be bounded, or may be different on alternative paths to the same call site. Appropriate actual vertices for the referenced objects cannot be constructed in this case. Actual vertices are only created for the parameters explicitly passed by value and not implicitly as referenced objects.

Like the UIG, the CCG includes call and entry vertices, together with call edges to represent function calls. Parameter binding edges connect actual and formal parameters.

An example parameter interface is shown in figure 4.4, representing the call `Inc(j)`. In this case, `j` is an integer and actual and formal parameters are created for the actual `j` and formal parameter `x`. If `j` was of some pointer type, only the pointer `j` would be given an actual vertex, whilst a formal vertex would exist only for the formal pointer parameter `x`. Any objects referenced by `j` would not be represented in the call interface.

This new parameter interface shows a major difference between the UIG and the new CCG. In the UIG only intraprocedural data dependencies are represented explicitly whilst all interprocedural

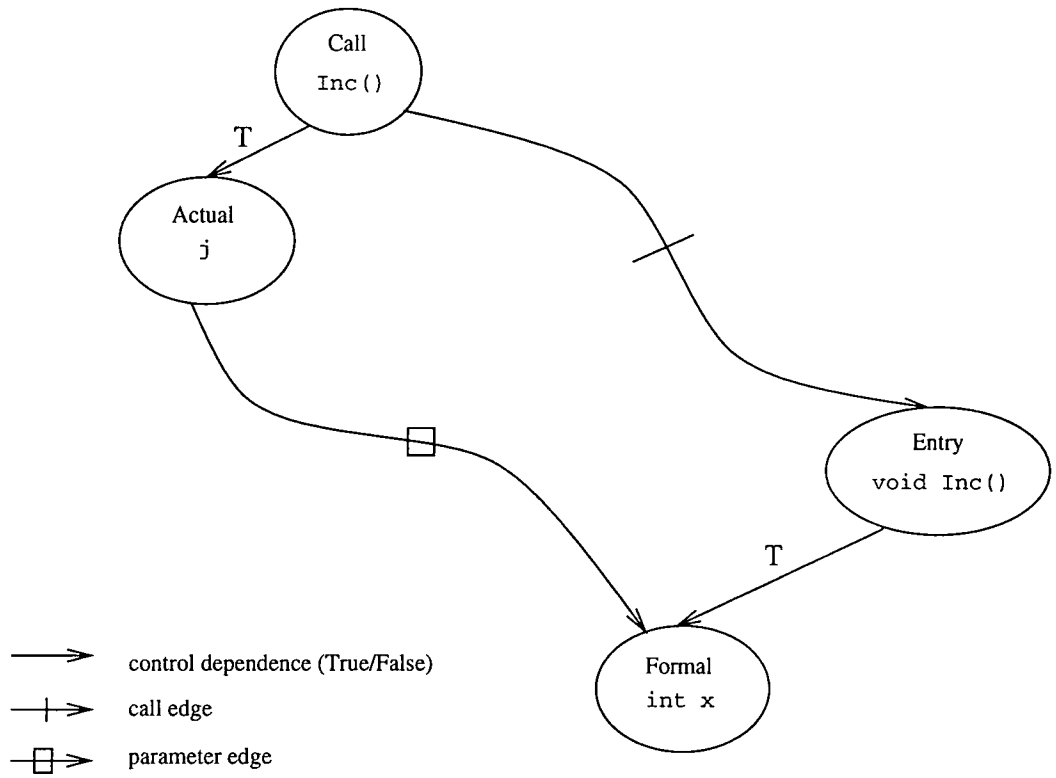


Figure 4.4: CCG parameter interface.

information is encapsulated within the call interface. Interprocedural definition-use associations arise as a result of the call by reference parameter passing scheme. A location may be defined within one procedure and later read within another procedure only where that location is visible as a result of its use as a reference parameter. This interprocedural definition-use information is not explicitly contained in the UIG. Instead, the IFG subcomponent of the UIG allows reachable use information to be gathered intraprocedurally and then to be propagated throughout the graph, making use of the encapsulated parameter interface. Interprocedural definition-use relationships can then be determined.

In C, the combination of pass by value and pointer parameters means that referenced objects can be accessed within a callee function without explicitly appearing in the parameter interface. Interprocedural dependencies can no longer be encapsulated within the call interface but must instead be represented explicitly in the CCG.

The CCG fragment in figure 4.5 shows such an explicit interprocedural data dependence. The variable `pj` is passed to the function `Inc` which increments the integer referenced by the formal parameter `x`. The binding effects of the call to `Inc` mean that `x` will reference and hence increment the object `*pj`, which is not involved in the CCG parameter interface. An interprocedural data dependence exists from the definition of `*pj` to the use of `*x` within `Inc`.

'Anonymous' parameters

The C language allows the value of an expression to be used as an actual parameter. It is possible to construct function calls of the form `f(g())`, where the actual parameter is the value returned from the function `g`. For calls of this format, the parameter interface is modified by the addition of a 'dummy' vertex. This new vertex represents the evaluation of the actual parameter and serves to maintain the 'shape' of the parameter interface where there is no explicit actual parameter vertex.

Figure 4.6 shows the CCG subgraph for the call `Inc(f())`. A dummy node is added to represent the evaluation of the actual parameter `f()`, which is derived from the return expression within `f`.

4.2.3 Control Structures

The CCG introduces the control structures `for`, `do..while`, `switch`, `break`, `continue` and `goto`, in addition to the `if..then..else` and `while` loops of the SDG/UIG representations.

For

The `for` statement:

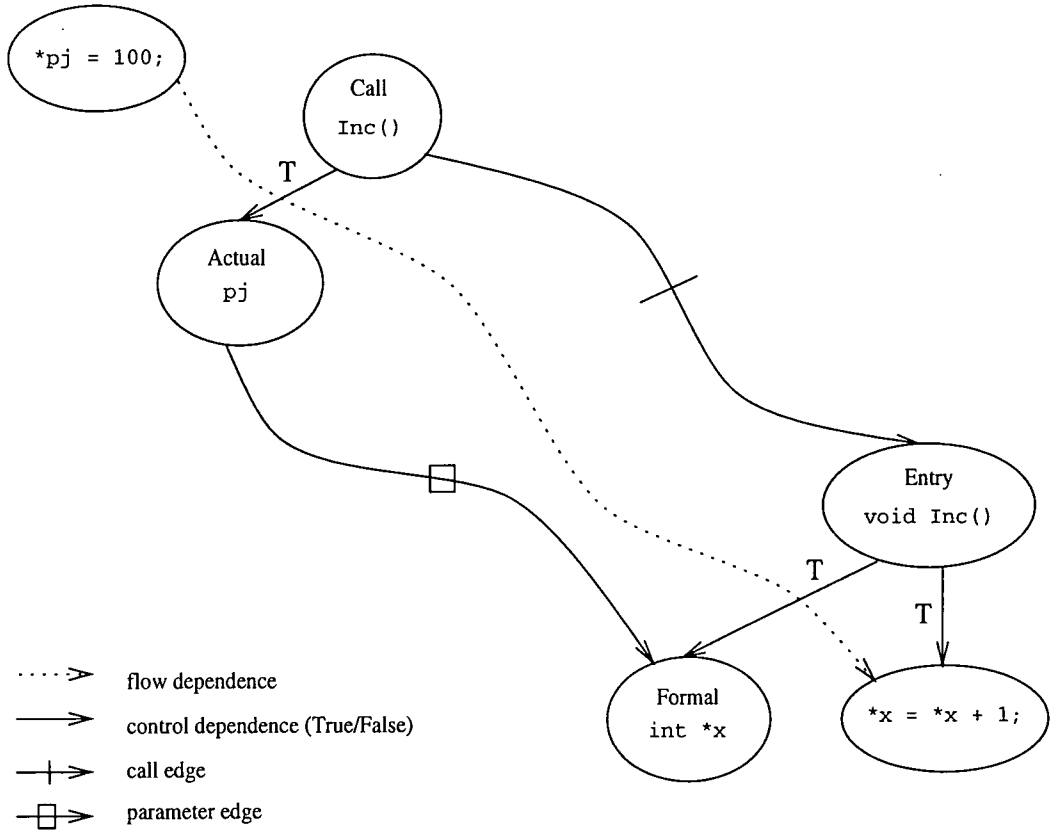


Figure 4.5: Explicit data dependence edge across parameter interface.

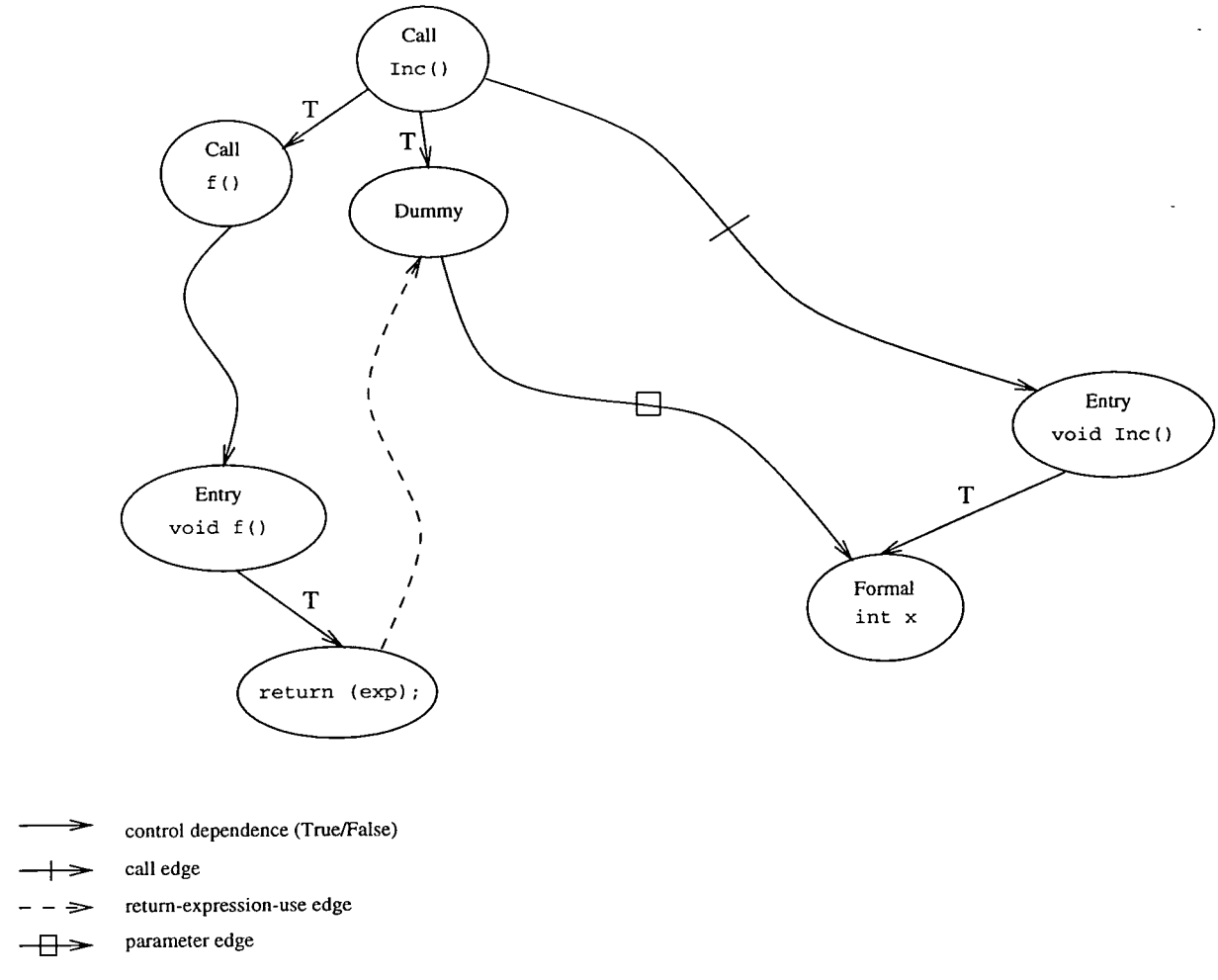


Figure 4.6: Dummy node for 'anonymous' parameter.

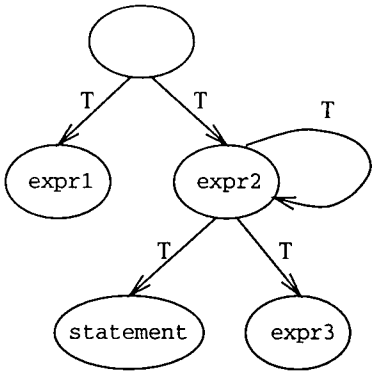


Figure 4.7: CCG control structure for `for` statement.

```
for (expr1; expr2, expr3)
   statement
```

can be represented by the while construct

```
expr1;
while (expr2) {
   statement
   expr3;
}
```

This gives rise to the control dependencies shown in figure 4.7.

Do..while

The `do..while` construct of the C language:

```
do
   statement
while (expr);
```

is represented by a similar pattern of control dependencies to the more common `while` loop. However the statements of the loop body are additionally control dependent on any enclosing condition. This is shown in figure 4.8.

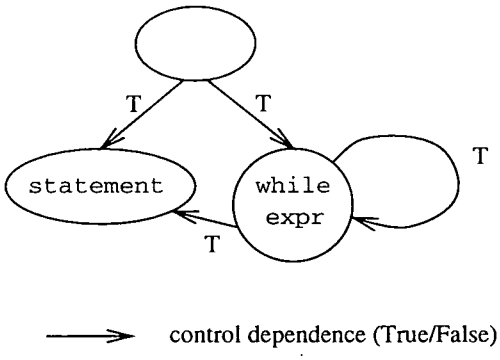


Figure 4.8: CCG control structure for `do..while` statement.

Switch

The multi-way decision `switch` statement is modelled using a new *switch* control dependence edge. Each constant expression of the `switch` statement is *switch dependent* on the `switch` expression. The statements following each constant expression are in turn control dependent on the constant expression vertex. Where the execution of the following statements does not terminate with a `break` statement, execution 'falls through' to the next `case` arm and these statements are also control dependent on the former constant expression. The representation for the following `switch` statement is shown in figure 4.9.

```
switch (expr) {
    case const-expr1: statements1;
                      break;
    case const-expr2: statements2;
    case const-expr3: statements3;
                      break;
    default: statements
}
```

`statements1` are control dependent on `case const-expr1`, and `statements2` are control dependent on `case const-expr2`. `statements3` are control dependent on both `case const-expr1` and `case const-expr2` since `statements2` do not terminate with a `break` statement, instead 'falling through' to `statements3`.

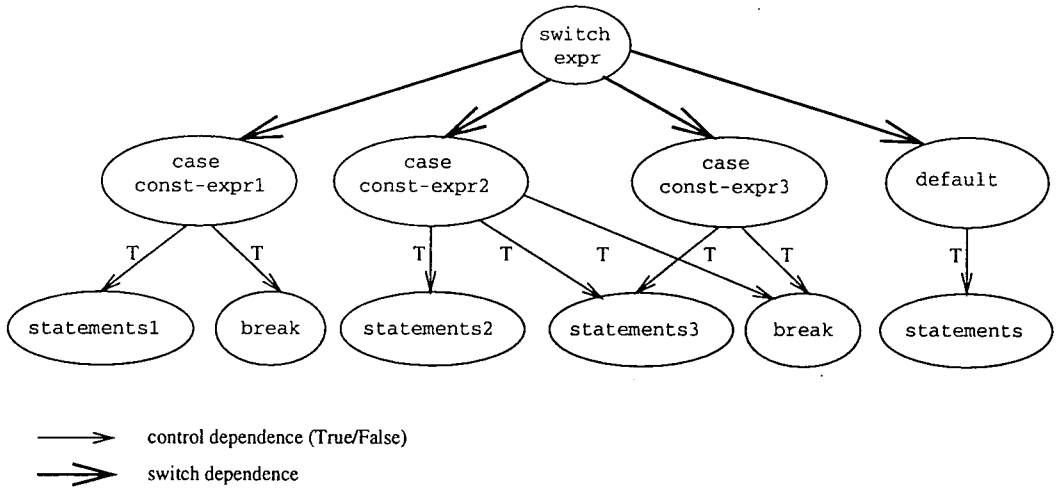


Figure 4.9: CCG control structure for `switch` statement.

Break, continue, goto

`Break` and `continue` statements, in providing early exits from loops, together with the `goto` statement, allow the construction of unstructured C programs. By defining control dependence in terms of a control flow graph and dominators, as reported by Ferrante et al[25], rather than simple nesting of control structures, as reported by Horwitz et al[43], control dependence information can be obtained for unstructured programs. No extra graph features are necessary.

4.2.4 Pointer, Structure and Array Variables

An accurate representation of the program's data dependencies requires analysis of any pointer, structure or array variables involved.

Pointer variables are effectively 'decomposed' to allow the construction of data dependencies through the pointer itself and also any objects referenced by the pointer. Any pointer variables referenced in the computation of an expression may give rise to data dependencies. For example, the statement `x = *p`, where `x` is of type `int` and `p` of type `*int`, references both the object `p` and the referenced object `*p`, before defining `x`. Flow dependencies incident on the vertex `x = *p` may consequently be through either the variable `p` or the referenced object `*p`. Similarly an lvalue expression may lead to dependencies through any pointer variables involved. For example, the assignment `*p = y`, where `p` is of type `*int` and `y` of type `int` will use both `y` and the pointer

p before defining object *p.

Structure variables are also 'decomposed' into primitive components to facilitate more accurate data flow analysis. Consider the following declarations.

```
struct point {
    int x;
    int y;
};

struct rect {
     struct point pt1;
     struct point pt2;
};

struct point pt;
struct rect screen;
```

The variable pt is decomposed into the primitive components `pt.x` and `pt.y`. The variable screen is decomposed into the primitive components `screen.pt1.x`, `screen.pt1.y`, `screen.pt2.x` and `screen.pt2.y`. Dependence analysis is carried out in terms of these primitive components. The structure assignment `pt = screen.pt1` uses both `screen.pt1.x` and `screen.pt1.y` and defines `pt.x` and `pt.y`.

More complicated data structures may involve both pointer and structure variables. For example:

```
struct node {
    int data;
    char *name;
};

struct node *item;
```

defines a pointer to a structure, a field of which is itself a pointer variable. This data structure can be decomposed into `item`, `(*item).data`, `(*item).name` and `*((*item).name)`.

Array variables present difficulties for static analysis techniques. Firstly in most cases the array element referred to by the expression `a[i]` cannot be determined since the value of the subscript `i` is unknown until run-time. In this case approximations to the possible elements involved must be made. Secondly arrays often consist of large numbers of elements and therefore using 'decomposition' techniques similar to those used with pointer and structure variables will be expensive in

terms of the space required. The approach taken is therefore to represent arrays as 'aggregates', i.e. a single variable, and not to attempt analysis of the individual elements. An assignment to an array element is considered to conditionally define the entire array. A reference of an array element is treated as a use of the entire array.

4.2.5 Block Structure, External and Static Variables

Variables in C may be declared at the start of any block, i.e. at the entry to any compound statement, and not only at the entry to a function. Declaration of a variable will hide any variables of the same name declared in outer blocks. The CCG reflects the scoping rules of the language and the visibility of variables in the creation of data dependence information.

Variables may also be declared in a C program external to all functions. These external variables may be defined within one function and referenced within another function, leading to an interprocedural data dependence. In the same way as the objects referenced by a pointer parameter, interprocedural data dependencies through global variables are represented explicitly in the CCG.

Where a local variable is declared as `static`, its value is preserved across invocations of the function or block in which the variable is declared. Static variables are treated in the same way as external variables, but with visibility limited according to the language's scoping rules.

4.2.6 Standard Library Functions

Standard library routines where the body of the function is unknown may be represented in the CCG by forming 'stub' routines having the same interprocedural effects. It is not necessary that a 'stub' routine model the intraprocedural effects of the actual function. Entry and actual vertices are created for the unknown routine, together with a return statement vertex if necessary. Any definitions of objects referenced by pointer parameters or external variables must also be represented to accurately model the function's interprocedural effects. This approach can also be employed where the subject C program contains as yet unimplemented functions. A CCG can be constructed by creating 'stub' routines to model the effects of these unknown functions.

An example stub routine for the standard library routine `labs`, defined in `stdlib.h`, is shown in table 4.1. The `labs` function returns the absolute value of its `long` argument. Its interprocedural effects are confined to call and parameter bindings, together with a value returned to the caller function. The actual value returned by `labs` is unimportant since the CCG ignores all constant values.

```
long labs(long n)
{
  return n;
}
```

Table 4.1: 'Stub' routine for `labs` standard library function.

4.3 Program Views

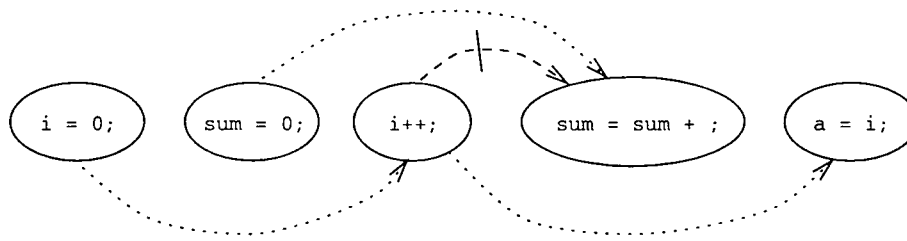
The following section describes the different program views which can be created from the CCG. These views are program slices, definition-use pairs, call graphs, control dependencies and flow sensitive data flow information.

4.3.1 Program Slicing

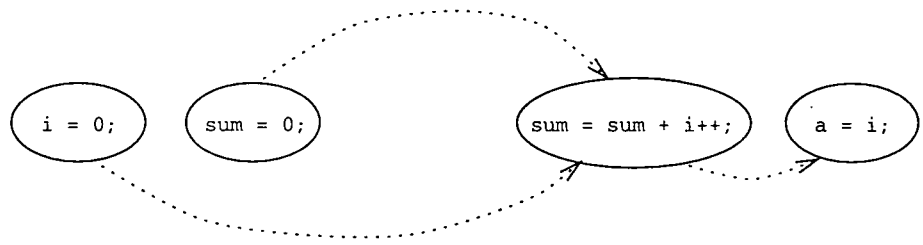
A simple program slice can be achieved on a dependence-based IPR by traversing backwards the control and data dependence edges of the IPR. This algorithm can be applied to the CCG to construct a program slice at any vertex of the graph. A slice at statement p on variable v is obtained by traversing backwards the edges of the CCG from p. Where v is defined at p, the program slice should include statements contributing to the values of any variables used to compute p. Hence all flow dependencies are traversed from p. Where v is used at p, the program slice should only include statements actually contributing to v and not to other variables defined or referenced at p. In this case only flow dependencies involving v are followed from p.

By producing a more refined program representation to deal with expressions with embedded side effects or control flows, as discussed in section 4.2.1, more accurate program slices can be constructed. In the following example, by refining the embedded side effect `i++`, a slice on `a` at statement 4 includes only those statements which contribute to `a`. Statements contributing to `sum` are not included.

```
1.  i = 0;
2.  sum = 0;
3.  sum = sum + i++;
4.  a = i;
```



(a) Refined CCG



(b) Embedded side effect

flow dependence

expression-use edge

Figure 4.10: Enhanced slicing accuracy with refined CCG.

A CCG for this program fragment is shown in figure 4.10(a). The resulting slice obtained from a backwards traversal of edges from `(a = i)` follows.

```
i = 0;
i++;
a = i;
```

If no extra node is created for the embedded side effect `i++` the graph in figure 4.10(b) results and the program slice becomes the entire program.

4.3.2 Ripple Analysis

By reversing the direction of the program slicing algorithm, rather than forming a solution showing the set of statements possibly affecting the value of a variable, the solution obtained represents

the potential effect of changing a variable at the given statement. A maintainer can form a 'ripple analysis' view by constructing a 'forward', rather than the more usual 'reverse' program slice. Again, the 'refined' CCG will give enhanced accuracy over a statement-based IPR.

4.3.3 Definition-use Pairs

Both intraprocedural and interprocedural definition-use associations are represented explicitly in the CCG. The maintainer may then view a CCG subgraph showing only these relationships, or may construct a more precise query to generate more specific definition-use information, such as definition-use relationships within a single function or from a single program statement.

4.3.4 Call Graph

Like the definition-use information, call relationships are contained explicitly within the CCG. A maintainer can query the representation to produce a complete program call graph, or more refined information such as the callees of a given function or call paths from one function to another.

4.3.5 Control Dependence Information

Control dependence information is similarly represented explicitly within the CCG. Intraprocedural control dependence views may be formed by taking simple subgraphs of the CCG representation.

4.3.6 Flow-sensitive Data Flow

Each FCCG is annotated with solutions to the flow-sensitive data flow problems *may be preserved*, *live on entry* and *live on exit*. A maintainer is able to form views showing this information for given functions or variables.

4.4 CCG Description

This section describes formally the vertices and edges comprising the CCG. The CCG is composed of a collection of FCCGs each representing an individual function of the C program. Each individual FCCG is a directed graph containing the vertices and edges described below.

4.4.1 FCCG Vertices

Each FCCG contains a unique *entry* vertex. Each formal parameter of the function is also represented by its own *formal parameter* vertex. Call sites within a function are represented by a

special *call vertex*. Any other program statement without embedded side effects or control flow is represented by an FCCG vertex.

Any statements containing embedded side effects or control flows generate vertices for each sub-expression containing a side effect or possible change in control flow. A binary expression containing side effects

`x ⊕ y`

gives the following vertices.

- `y, x ⊕` : side effect in `y`
- `x, ⊕ y` : side effect in `x`
- `x, y, ⊕` : side effects in `x` and `y`.

A boolean expression with short-circuit evaluation

`a ⊕ b ⊕ c`

with precedence `(a ⊕ b) ⊕ c` gives three vertices

`a , ⊕ b, ⊕ c`

A C conditional expression

`a ? b : c`

gives three vertices

`a, b, c`

Any FCCG vertex which evaluates an actual parameter is termed an *actual vertex*. A *dummy vertex* is created whenever an actual parameter value is the return value of a function call. The function call

`f(g())`

gives vertices `call f`, `call g` and `dummy`.

4.4.2 FCCG Edges

Intraprocedural flow dependencies

The definition of intraprocedural flow dependence is given in terms of memory locations.

> Given program-points p and q within function f, program-point q has a flow dependence on program-point p, $p \rightarrow_f q$, if p writes into a memory location *loc* that q reads, and there is no intervening write into *loc* along the execution path by which q is reached from p.

An intraprocedural flow dependence $p \rightarrow_f q$ corresponds to a definition-use association from vertex p to vertex q.

Control dependencies

Control dependence is defined in terms of the program control flow graph and dominators.

Control flow graph A directed graph G with a unique entry vertex $START$ and unique exit vertex $STOP$. Each vertex has up to two adjacent vertices. Edges are labelled *true*, *false* or *unconditional*. For any vertex N in G there exists a path from $START$ to N and from N to $STOP$.

Post-dominator A vertex V is *post-dominated* by a vertex W in G if every directed path from V to $STOP$ (not including V) contains W.

Control dependence Let X and Y be vertices in G. Y is *control dependent* on X iff

- there exists a directed path P from X to Y with any Z in P (excluding X and Y) post-dominated by Y, and
- X is not post-dominated by Y.

A control dependence from vertex p to vertex q, $p \rightarrow_c q$, is labelled true or false, giving $p \rightarrow_c^T q$ or $p \rightarrow_c^F q$.

> Whenever the predicate represented by vertex p is evaluated and its value is equal to that of the label, the program component executed by vertex q will be executed if the program terminates.

Formal parameter vertices are each control dependent on the entry vertex. Any actual vertex or FCCG vertex derived from an actual parameter expression and not enclosed by any conditional is control dependent on the corresponding call vertex.

A switch dependence $s \rightarrow_s c$ exists from any switch vertex s to each case constant-expression c.

Expression-use edge

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. A binary expression containing side effects

x $\oplus$ y

gives the following expression-use edges:

- $\mathtt{y} \rightarrow_{eu} \mathtt{x} \oplus$: side effect in y
- $\mathtt{x} \rightarrow_{eu} \oplus \mathtt{y}$: side effect in x
- $\mathtt{x} \rightarrow_{eu} \oplus$, $\mathtt{y} \rightarrow_{eu} \oplus$: side effects in x and y

Lvalue-definition edge

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 write to the corresponding storage location at vertex q. An assignment expression:

$\mathtt{x} \oplus \mathtt{y}$

with side effects within x gives the lvalue-definition edge:

$\mathtt{x} \rightarrow_{ld} \oplus \mathtt{y}$: side effect in x

4.4.3 Interprocedural CCG Edges

Interprocedural edges are separated into two classes, those associated with the call interface and those which represent interprocedural data dependencies.

Call interface edges

Call edge A call from a function f to a function g is represented by a *call edge*, $c \rightarrow_{call} e$, from the call vertex c within f to the entry vertex e of g. A recursive function will contain a call edge from the recursive call vertex to its own entry vertex.

Parameter binding edge A parameter binding edge, $a \rightarrow_{bind} f$, connects each actual parameter vertex a and the corresponding formal parameter vertex f within the callee.

Return expression-use edge A return-expression-use edge from return vertex p to vertex q, $p \rightarrow_{reu} q$, indicates the evaluation of an expression at the return statement p, the value of which is subsequently used in an expression at statement q.

Interprocedural flow dependencies

An interprocedural flow dependence is defined as follows.

> Program-point q within function h has an interprocedural flow dependence on program-point p within function g, $p \rightarrow_f q$, if p writes into a memory location *loc* that q reads,

and there is no intervening write into *loc* along the execution path by which q is reached from p.

The flow dependence $p \rightarrow_f q$, corresponds to an interprocedural definition-use association from vertex p within function g to vertex q in function h. Interprocedural flow dependencies are described using the same notation as intraprocedural flow dependencies.

4.4.4 Graph Annotations

Each FCCG is annotated with solutions to three flow-sensitive data flow problems.

- *may be preserved* - a variable v may be preserved across a call to function f if there is a path through f along which v is visible after formal/actual parameter binding and at the return vertex of f and v is not defined on that path.
- *live on entry* - a variable v is live on entry to function f if v is visible after formal/actual parameter binding and there is a path through f along which v is used either within f, or in any function transitively called from f, before being defined.
- *live on exit* - a variable v is live on exit from function f if v is visible at the return vertex of f and there is a path on exiting f along which v is used before being defined.

Three sets are associated with each function f.

- $preserved(f)$
- $live\_on\_entry(f)$
- $live\_on\_exit(f)$

Each variable in $preserved(f)$ and $live\_on\_entry(f)$ is given its name after the formal/actual parameter binding within f, and also at the point at which it is initially defined, in the form $function/name/lineno$. Each variable in $live\_on\_exit(f)$ is given its name at a return vertex of f, and also at the point at which it is initially defined, in the form $function/name/lineno$.

4.5 CCG Construction

This section describes the construction of the CCG. The CCG is constructed in three steps. The first step is to determine the vertices, expression-use edges and lvalue-definition edges for each function of the subject C program. These edges are intraprocedural and may be computed for each

FCCG independently of information from any other function. Intraprocedural control dependence analysis is then performed giving a 'partial' FCCG for each function.

The second step of the CCG construction is to connect each partial FCCG by adding call, parameter binding and return-expression-use edges.

The final step involves the computation of both intraprocedural and interprocedural data dependence information, employing data flow analysis techniques. Existing dependence algorithms are extended to allow the computation of flow-sensitive data flow information.

4.5.1 Partial FCCG Construction

FCCG vertices, expression-use edges, lvalue-definition edges

The vertices of an FCCG are derived directly from the statements making up the subject C function. However, analysis of each expression is necessary to detect any embedded side effects or embedded control flows and to refine the FCCG accordingly.

An abstract syntax tree is first constructed for the subject function. A depth-first traversal of this tree is performed to determine embedded side effects and control flow within the expressions of each statement and to construct the vertices of the FCCG. Operators of interest during the traversal of an expression tree are: assignment operators, postfix increment and decrement, prefix increment and decrement, logical and, logical or, the conditional operator, the comma operator and function calls.

Where a subtree of an expression has at its root vertex an assignment operator, postfix increment or decrement, prefix increment or decrement or function call, this subtree forms a sub-expression with embedded side effects. Such sub-expressions are identified during the depth-first traversal of the abstract syntax tree and refined vertices created.

Expression-use edges are added from the sub-expression vertex to the parent vertex, except in the case where the sub-expression is the left-child of an assignment operator. This sub-expression evaluates an lvalue and consequently an lvalue-definition edge connects the sub-expression and parent vertices.

The abstract syntax tree for the expression

```
*p++ = x + (y = 100);
```

is shown in figure 4.11. The depth-first traversal of this tree identifies sub-expressions with side effects `*p++` and `y = 100`. Three vertices are created - `*p++`, `y = 100` and `= x +`. These vertices are connected by the expression-use edge `(y = 100)` $\rightarrow_{eu}$ `(= x +)` and lvalue-definition edge

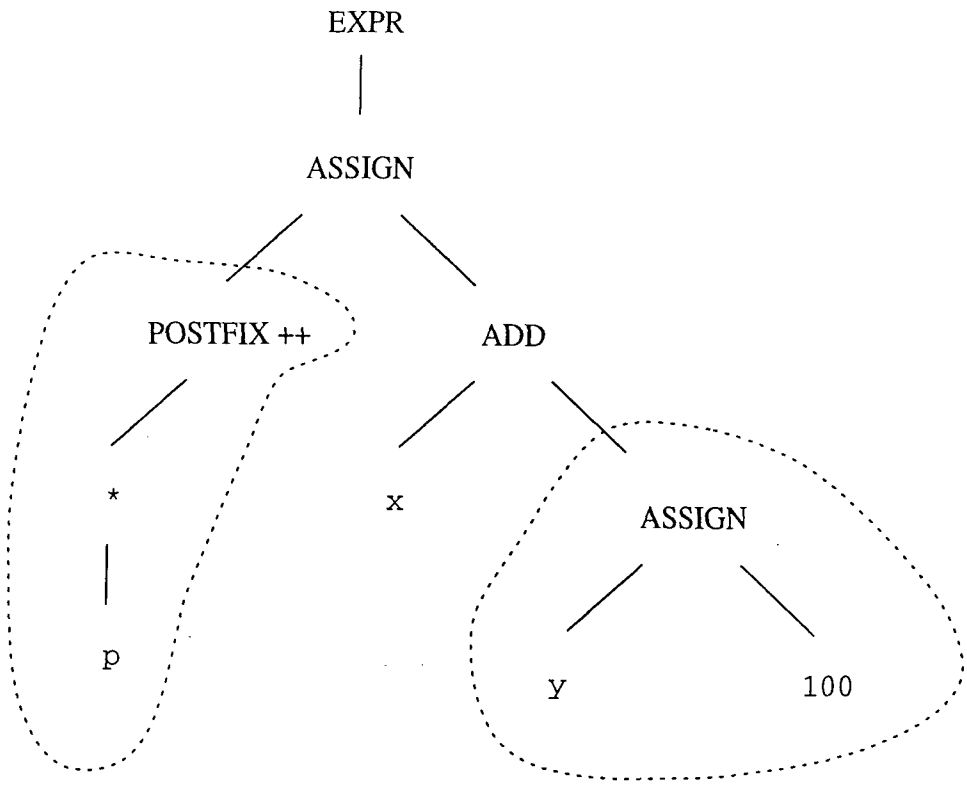


Figure 4.11: Abstract syntax tree for expression `*p++ = x + (y = 100);`

`(*p++)`$\rightarrow_{ld}$`(= x +)`. The resulting vertices and edges are shown in figure 4.12.

Where a subtree of an expression has as its root vertex a logical and, logical or, conditional operator or comma operator, this subtree forms a sub-expression with embedded control flow. In each case the left and right sub-expressions of the root vertex form refined FCCG vertices. The abstract syntax tree for the expression

```
a && (b || c)
```

is shown in figure 4.13. Three vertex-forming sub-expressions are formed by a depth first traversal of this tree, giving vertices `a`, `b` and `c`.

A special entry vertex is created for the FCCG and a formal vertex for each formal parameter. Actual parameters are identified during the abstract syntax tree traversal and abstract and dummy vertices created accordingly. The function's control flow graph is also created during the traversal of the abstract syntax tree.

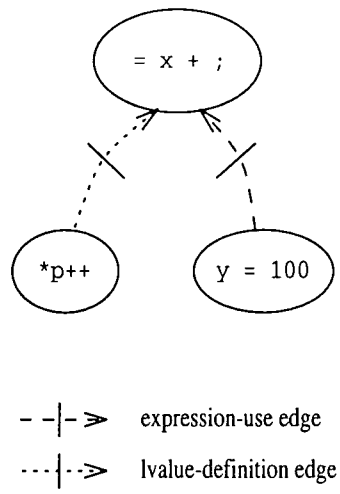


Figure 4.12: Vertices and edges derived from expression `*p++ = x + (y = 100);`

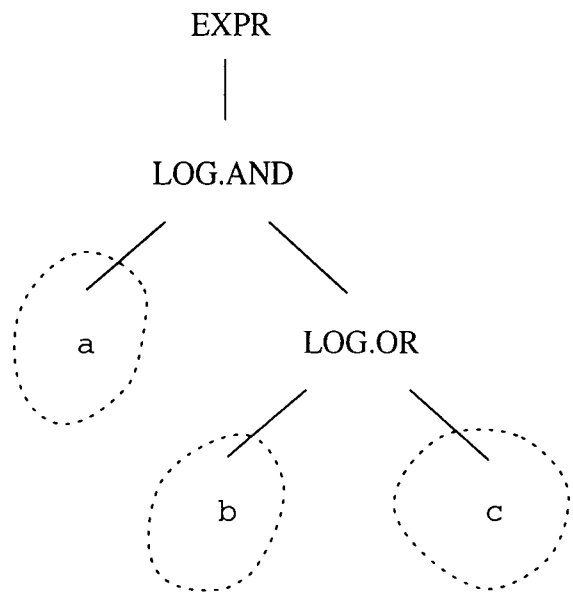


Figure 4.13: Abstract syntax tree for expression `a && (b || c);`

Control dependence analysis

Control dependence information is constructed for each FCCG using an algorithm derived from Ferrante at al[25]. The algorithm allows the computation of control dependencies based on the program control flow graph constructed earlier. The algorithm is applied initially on the flow graph for each function, producing a control dependence graph rooted at the function's entry vertex. The algorithm is then applied on the control flow subgraph for each actual parameter list of each function call. This creates a control dependence subgraph rooted at each function call node, representing control dependencies within the call's actual parameters.

The definitions of control flow graph, post-dominator and control dependence used in this algorithm are those given in section 4.4. The first stage in the calculation of control dependencies is to construct the post-dominator relations for an augmented control flow graph. The control flow graph has a special predicate vertex $ENTRY$ with outgoing edges to $START$ (true) and $STOP$ (false), representing the external conditions required for program execution.

The computation of post-dominator relationships can be achieved by computing dominators on the 'reverse' control flow graph. This graph is created by simply reversing the edges of the control flow graph. The notion of dominance is defined as follows.

> Let X and Y be vertices in the control flow graph G. Vertex X dominates vertex Y if every path from $START$ to Y includes X.

Dominator sets, $Dom(n)$, the set of vertices dominating a vertex n, may be constructed for each vertex using the algorithm defined in table 4.2. Post-dominator sets, $PostDom(n)$, the set of vertices post-dominating n, are calculated by performing this algorithm with $n_0 = STOP$ and employing the reverse control flow graph.

The post-dominator sets are now converted to a post-dominance tree representation, in which each vertex post-dominates only its descendents. Each vertex n in the control flow graph has a unique *immediate post-dominator*.

> The immediate post-dominator m of vertex n, $immed\_post\_dom(n)$, is the last post-dominator on any path from $STOP$ to n.

In terms of the post-dominator relation,

$$If \;\; d \neq n \; and \; d \in Postdom(n), \; then \; d \; postdom \; m$$

The post-dominance tree is constructed such that

$$Parent \; of \; n \; = \; immed\_post\_dom(n).$$

Let -

n_0 = *start vertex of control flow graph G*

$Dom(n)$ = $\forall\ m \in Dom(n) \mid m$ *dominates* n

N = *Set of vertices in control flow graph G*

Initialisation -

$D(n_0) := \{n_0\}$

For $n \in N - \{n_0\}$ *do*

$D(n) := N$

Update -

While changes in any $D(n)$

For $n \in N - \{n_0\}$ *do*

$D(n) := \{n\} \cup \bigcap_{p \text{ is a predecessor of } n} D(p)$

Table 4.2: Computation of dominator sets.

Self post-dominators are then removed from the resulting sets and a control flow graph edge set S is calculated such that:

$$S = \{edge(A,B) \mid B \textit{ is not an ancestor of } A \textit{ in the post} - \textit{dominator tree} \\ \&\& \; edge(A,B) \in \textit{control flow graph } G \}$$

Control dependence edges are then determined using this set S as follows.

For $edge(A,B)$ *in* S

find L, *the least* $-$ *common parent of* A *and* B *in the post* $-$ *dominator tree*

L will either be A itself, or the parent of A in the post-dominator tree. The proof of this is given by Ferrante at al[25].

Case 1 : L = *parent of* A

All nodes in the post $-$ *dominator tree on the path from* L *to* B, *including* B *but not* L, *are control dependent on* A.

Case 2 : $L = A$

All nodes in the post $-$ *dominator tree on the path from* A *to* B , *including* A *and* B, *are control dependent on* A.

The required nodes can be found by performing a backwards traversal of the post-dominator tree from B until reaching the parent of A (if a parent exists). All vertices visited before the parent of A become control dependent on A. The label of the control dependencies formed will be equivalent to the label of the control flow graph edge $edge(A,B)$.

An example control flow graph and its post-dominator relations are shown in figure 4.14. Figure 4.15 illustrates the post-dominator tree constructed from these relations. Examining the control flow graph to determine S, the set of edges A, B such that B is not an ancestor of A in the post-dominator tree gives:

$$S = \{(ENTRY, START), (1,2), (4,5), (4,6), (7,4), (8,9)\}$$

Examining these edges produces the following control dependencies.

$ENTRY \rightarrow_c^T START$, $ENTRY \rightarrow_c^T 1$, $ENTRY \rightarrow_c^T 3$, $ENTRY \rightarrow_c^T 4$, $ENTRY \rightarrow_c^T 7$
$ENTRY \rightarrow_c^T 8$, $ENTRY \rightarrow_c^T 10$

$1 \rightarrow_c^T 2$

$4 \rightarrow_c^T 5$, $4 \rightarrow_c^F 6$

$7 \rightarrow_c^T 4,\ 7 \rightarrow_c^T 7$

$8 \rightarrow_c^T 1,\ 8 \rightarrow_c^T 3,\ 8 \rightarrow_c^T 4,\ 8 \rightarrow_c^T 7,\ 8 \rightarrow_c^T 8,\ 8 \rightarrow_c^T 9$

Switch dependencies may also be constructed along with the control dependence edges by modifying the input control flow graph. Each switch vertex has flow edges labelled *switch* to each `case` and `default` vertex. These vertices have *true* control flow edges to the first statement of the corresponding statement list and additionally a *false* control flow edge to the statement following the entire construct. This *false* edge forces the case statement lists to be control dependent on the `case` or `default` vertices, and these in turn to be switch dependent on the switch vertex, as defined in section 4.4.

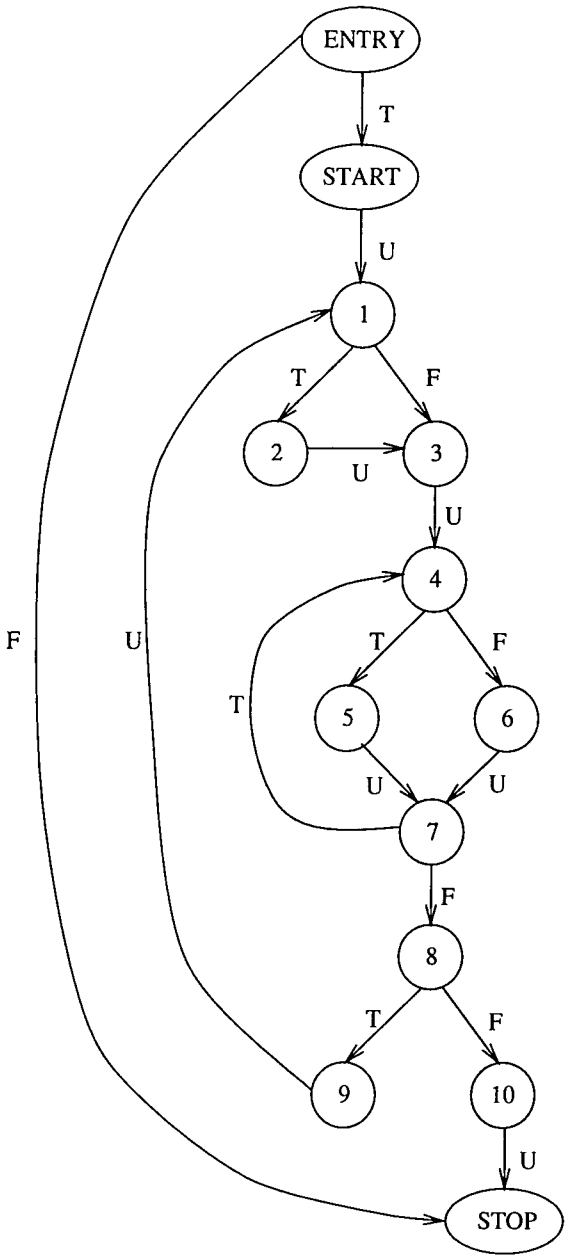
4.5.2 Connecting the Partial FCCG Subgraphs

The second step in the construction of the CCG is to connect the partial FCCG subgraphs. For C programs not making use of pointers to functions, call edges can be constructed by simply connecting any call vertices created in step one with the corresponding entry vertices. Binding edges are constructed to connect actual and formal parameter vertices, representing the associations between formal and actual parameters. Finally return-expression-use edges connect vertices evaluating the return value of a function to vertices at which this value is referenced.

4.5.3 Data Dependence Analysis

The computation of data dependence information for programs with pointer variables is an area of ongoing research. Methods are derived from the compiling community, where research attempts to model statically the effects of pointer variables associated with recursive data structures to determine possible parallelisation. Examples of such work is by Horwitz et al[42] and Chase et al[17]. Alternative data dependence analysis techniques have been described by Landi and Ryder[54][55] and Pande et al[68] who use conditional analysis techniques to determine pointer-induced aliasing and reaching definition information for C programs.

The method employed in the construction of the CCG is based on that described by Horwitz et al[42] and extended to allow the additional computation of flow-sensitive data flow analysis information *may be preserved*, *live on entry* and *live on exit*. The algorithm described by Horwitz et al allows the calculation of data dependencies in the presence of pointers, structures and dynamic memory allocation. The data dependencies calculated using this method are a static approximation



PostDom(STOP) = { }
PostDom(ENTRY) = {STOP}
PostDom(START) = {STOP,1,3,4,7,8,10}
PostDom(1) = {STOP,3,4,7,8,10}
PostDom(2) = {STOP,3,4,7,8,10}
PostDom(3) = {STOP,4,7,8,10}
PostDom(4) = {STOP,7,8,10}
PostDom(5) = {STOP,7,8,10}
PostDom(6) = {STOP,7,8,10}
PostDom(7) = {STOP,8,10}
PostDom(8) = {STOP,10}
PostDom(9) = {STOP,1,3,4,7,8,10}
PostDom(10) = {STOP}

Figure 4.14: Control flow graph and post-dominator sets.

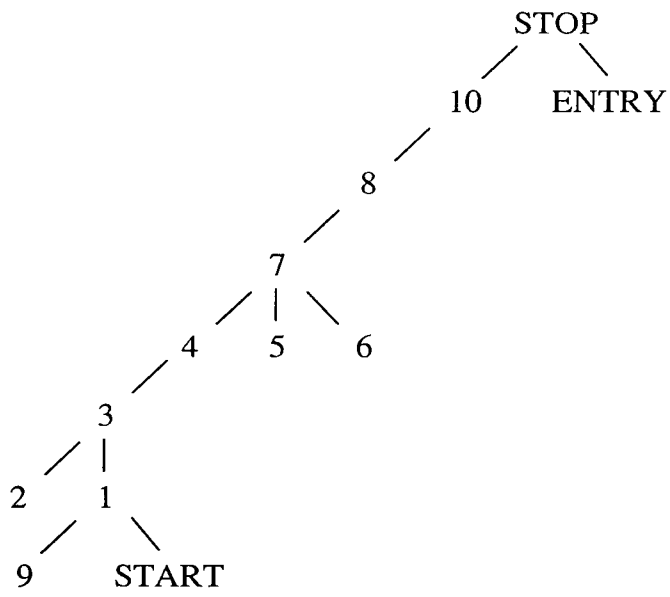


Figure 4.15: Post-dominator tree.

of those dependencies actually arising during program execution.

The algorithm is divided into two phases. The first phase, the 'reaching-stores phase', computes at each program vertex a set of store graphs that approximate the possible memory layouts that could arise during execution. Program variables, together with any dynamic variables allocated during execution, are represented by abstract memory locations. Structure variables are decomposed into the constituent fields and abstract locations allocated for each field. Array variables are represented by a single abstract location. Each abstract memory location is labelled by the CCG vertex which last wrote to that location. The second phase, the 'inference phase', examines the set of stores reaching each program vertex and determines the locations read. A flow dependence $p \rightarrow_f q$ exists if q reads a location labelled p in any store graph reaching q.

Reaching-stores phase

Each store graph at a program vertex is made up of subgraphs representing each function in the current activation stack, together with subgraphs representing external variables, static variables and dynamically allocated variables. Each function subgraph contains abstract locations for each local variable. The 'external' and 'static' variable subgraphs contain abstract locations representing

the program's external or static variables. The 'dynamic' subgraph contains abstract locations representing each of the program's dynamically allocated variables.

An abstract location *loc* is represented by the following relations:

- *id*(*loc*) - a unique identifier.
- *name*(*loc*) - the name of the program variable represented by *loc*. Static and local variables are in the form */function/name/lineno* and external variables in the form *name*. Dynamically allocated variables are not named.
- *last_def*(*loc*) - the program vertex at which *loc* was last defined.
- *points_to*(*loc*) - the set of abstract locations pointed to by *loc*.

An initial store graph is created at the start of the program comprising the external and static variable subgraphs only, together with an initially empty 'dynamic' subgraph. This store graph is iterated throughout the program's control flow/call graph. In the absence of expression-use, return expression-use and lvalue-definition edges the store-graph is updated at each vertex p as follows. Let `a` and `b` each represent expressions.

- p is an entry vertex:
  - Add to the store graph a new subgraph representing the newly active function.
- p is a return vertex:
  - Remove from the store graph the subgraph representing the currently active function.
- p is a non-pointer assignment `a = b`:
  - Determine the object o referred to by expression `a`.
  - Find the abstract location *loc* corresponding to o.
  - $last\_def(loc) = p$.
- p is a pointer assignment `a = b`:
  - Determine the object $o1$ referred to by expression `a`.
  - Find the abstract location $loc1$ corresponding to $o1$.
  - Determine the object $o2$ referred to by expression `b`.

- Find the abstract location $loc2$ corresponding to $o2$.
- $points\_to(loc1) = points\_to(loc2)$.
- $last\_def(loc1) = p$.

- p is a pointer assignment `a = &b`:
  - Determine the object $o1$ referred to by expression `a`.
  - Find the abstract location $loc1$ corresponding to $o1$.
  - Determine the object $o2$ referred to by expression `b`.
  - Find the abstract location $loc2$ corresponding to $o2$.
  - $points\_to(loc1) = loc2$.
  - $last\_def(loc1) = p$.

This algorithm is modified when the CCG contains expression-use, return-expression-use or lvalue-definition edges. These edges each evaluate expressions at the source vertex and use the resulting values at the sink vertex. Annotations are attached to the edges during analysis of the source vertex to enable these values to be 'transmitted'. The annotations are later read during analysis of the corresponding sink vertex to determine $loc1$ and $loc2$ at the sink vertex.

A refined CCG vertex r with an outgoing expression-use, return-expression-use or lvalue-definition edge annotates these outgoing edges as follows.

- Where r represents an expression `a` with outgoing lvalue-definition edge e:
  - Determine the object o referred to by expression `a`.
  - Find the abstract location loc corresponding to o.
  - Annotate e with $id(loc)$.
- Where r represents a pointer expression `a` with outgoing expression-use or return-expression-use edge e:
  - Determine the object o referred to by expression `a`.
  - Find the abstract location loc corresponding to o.
  - Annotate e with $id(loc)$.
- Where r represents an ordinal expression `a` with outgoing expression-use or return-expression-use edge e:

- Annotate e with $NULL$.

A refined CCG vertex r with incoming expression-use, return-expression-use or lvalue-definition edges reads these annotations. Abstract locations $loc1$ and/or $loc2$ are determined from these annotations rather than the sub-expressions at r. $annot(e)$ refers to the annotation associated with edge e.

- Where r represents an expression a with an incoming lvalue-definition edge e:
  - $loc1 = annot(e)$.
- Where r represents an expression a with an incoming expression-use or return-expression-use edge e:
  - $if\ annot(e) \neq NULL,\ loc2 = annot(e)$.

The relations $points\_to$ and $last\_def$ are then updated as before.

Inference phase

The inference phase examines the resulting store graphs at each program vertex p. Flow dependencies are constructed as follows:

- For each object o used at p:
  - Find the abstract location loc corresponding to o.
  - Construct flow dependence $last\_def(loc) \rightarrow_f p$.

Flow sensitive data flow analysis

Three flow-sensitive data flow analysis problems, *may be preserved*, *live on entry* and *live on exit* can be solved by extending the reaching stores and inference phases of the data dependence analysis algorithm. Each abstract location of a store graph at each CCG vertex has additionally two further relations $entry(loc)$ and $exit(loc)$. The former contains tuples of the form (f, v), where:

- f is a function name.
- v is a name referring to a variable visible within f after formal/actual parameter binding.

The latter contains tuples of the form (f, v), where:

- f is a function name.
- v is a name referring to a variable visible within f at a return vertex of f.

The *entry* set for an abstract location in a store graph at a CCG vertex p represents those functions which the location has entered and not yet exited on the execution path to p, after the location was last defined.

The *exit* set for an abstract location in a store graph at a CCG vertex p represents those functions which the location has exited on the execution path to p, after the location was last defined.

The *initial* CCG vertex of a function f is the vertex which is the immediate successor of the final *formal* parameter vertex in the control flow graph.

The *entry* set for each abstract location *loc* is constructed during the reaching-stores phase in the following way.

- $entry(loc) = \{\}$
- At the *initial* CCG vertex i within function f, with *loc* visible, add the tuple (f, v), where v is the name referring to *loc* at i.
- If *loc* is defined at a CCG vertex, $entry(loc) = \{\}$
- When exiting function f, remove all tuples of the form (f, v) from *entry*.

The *exit* set for each location *loc* is constructed during the reaching-stores phase in the following way.

- $exit(loc) = \{\}$
- Following a return vertex r within function f, with *loc* visible, add the tuple (f, v), where v is the name referring to *loc* at the r.
- If *loc* is defined at a CCG vertex, $exit(loc) = \{\}$

Solutions to the data flow problems *may be preserved*, *live on entry* and *live on exit* may now be computed during the inference phase as follows.

- *may be preserved* - a variable v may be preserved across a function f if the *entry* set of its corresponding location *loc* at a return vertex of f contains a tuple (f, v). This indicates that v has not been defined on this path through f and hence has been preserved. The name of variable v when first defined is obtained from the $name(loc)$ relation.

```
int main()
{
  int a, b;

  b = a = 0;
  while (b < 5)
    a = sq(&b);
}
```

```
1.  enter main()
2.  a = 0
3.  b =
4.  while (b < 5)
5.  &b
6.  call sq()
7.  a =
```

```
int sq(int *f)
{
  (*f)++;
  return *f * *f;
}
```

```
8.  enter sq()
9.  f
10. (*f)++
11. return *f * *f
```

Table 4.3: Example C program and corresponding CCG vertices.

- *live on entry* - when a variable v is used at a CCG vertex p within function f, for each tuple $(g, v) \in entry$ at the corresponding abstract location loc at p, add v to $live\_on\_entry(g)$. The name of variable v when first defined is obtained from the $name(loc)$ relation.

- *live on exit* - when a variable v is used at a CCG vertex p within function f, for each tuple $(g, v) \in exit$ at the corresponding abstract location loc at p, add v to $live\_on\_exit(g)$. The name of variable v when first defined is obtained from the $name(loc)$ relation.

Example

Table 4.3 contains an example C program and the corresponding CCG vertices, numbered 1 to 11. Figures 4.16, 4.17 and 4.18 show diagrammatic representations of the store graphs created during the reaching stores phase for this example program. The program contains only local variable vertices and hence the 'external', 'static' and 'dynamic' subgraphs are empty. On entering `main` a subgraph for this function, containing abstract locations representing local variables `a` and `b` is added to the initially empty store graph. These abstract locations are each updated as program variables are defined at vertices 2 and 3.

On entering function `sq`, an additional subgraph is added, containing the local variable `f` of `sq`.

The variable f is defined at CCG vertex 9 and hence the corresponding location $l2$ is updated. The relation *points_to*($l2$) is also updated to reflect the effects of the formal-actual parameter binding.

Using this relation, vertex 10 updates the location $l1$ representing the variable *f. On leaving function sq the subgraph representing sq is removed from the current store graph. The iteration continues until a fixed-point solution is attained, in this case during the second execution of sq.

The inference phase examines each store graph at each CCG vertex. For each variable used, the corresponding abstract location *loc* is determined and flow dependencies constructed from *last_def*(*loc*). For example, vertex 11 uses both f and *f, which are locations $l2$ and $l1$ respectively. *last_def*($l2$) is 9, giving a flow dependence:

(f)$\rightarrow_f$(return *f * *f)

last_def($l1$) is 10, giving a flow dependence:

((*f)++)$\rightarrow_f$(return *f * *f)

At vertex 4, variable b (location $l1$) is used. Examining the store graphs at 4 shows *last_def*($l1$) to be 2 or 10, giving intraprocedural flow dependence:

(b=)$\rightarrow_f$(while (b<5))

and interprocedural flow dependence:

((*f)++)$\rightarrow_f$(while (b<5)).

Other flow dependencies created are:

(b=)$\rightarrow_f$(while (b<5))

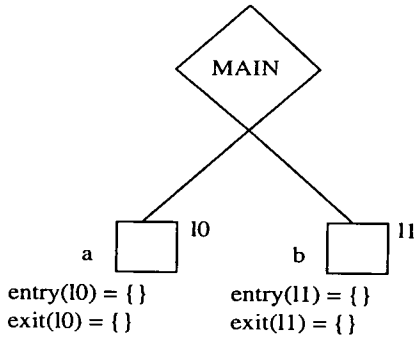
(f)$\rightarrow_f$((*f)++)

Entry and *exit* sets are attached to the abstract locations of the store graphs. During the reaching stores phase, at vertex 10, the tuple (sq, *f) is added to the *entry* set for the abstract location $l1$ representing b. At this vertex *f is defined and *entry*($l1$) becomes {} at the following vertex 11.

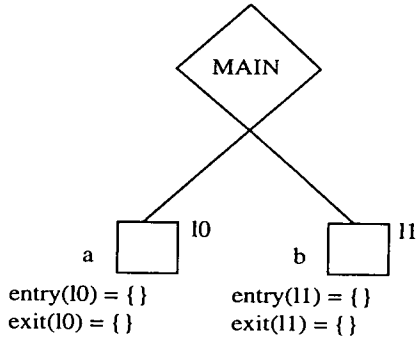
At the return vertex of sq, the variable *f is again visible and is added to *exit*($l1$), which becomes (sq, *f). On the second iteration, *exit*($l1$) is set to {} at vertex 10 when *f is defined.

Examining the *entry* and *exit* sets during the inference phase gives, *may_be_preserved*(sq) = {}, since each entry set is empty at the return vertex of sq. *live_on_entry*(sq) is found to be {(*f, /main/b/1)} due to the use of *f at 10 with *entry*(sq) = {(sq, *f)}. *live_on_exit*(sq) is found to be {(*f, /main/b/1)} due to the use of *f at 10 with *exit*(sq) = {(sq, *f)}.

1. `main()`



2. `b = 0`



3. `a =`

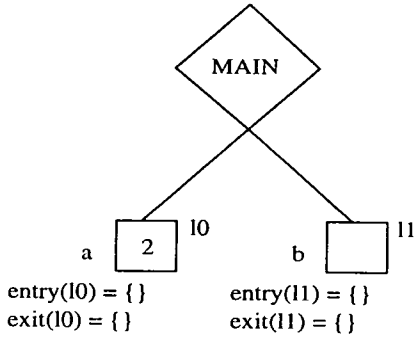
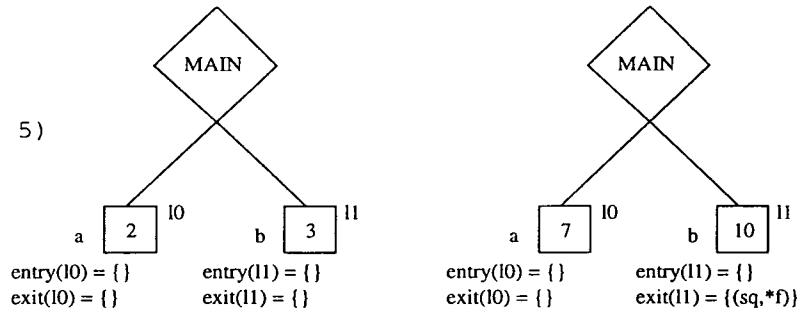
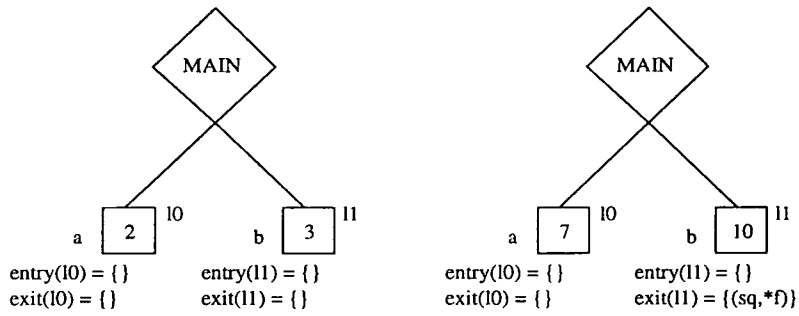


Figure 4.16: Data dependence analysis for example program.

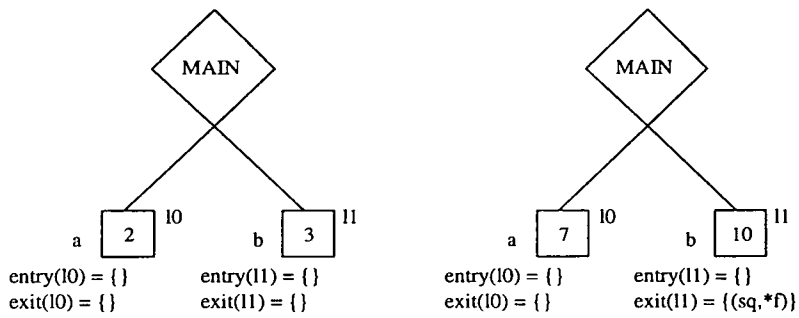
4. `while (b < 5)`



5. `&b`



6. `call(sq)`



7. `a =`

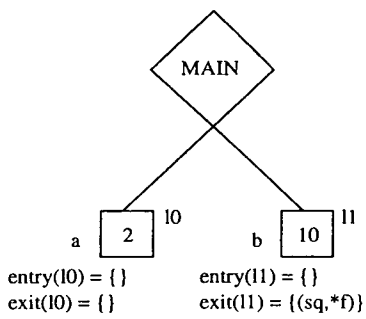
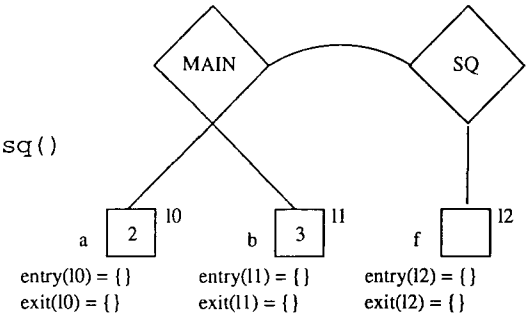
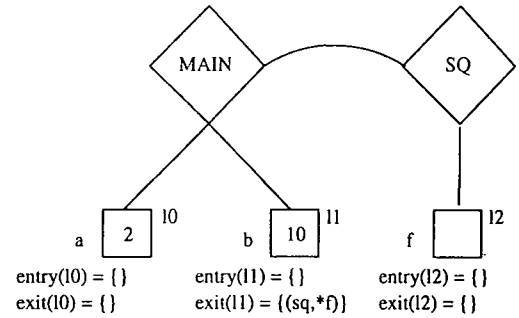


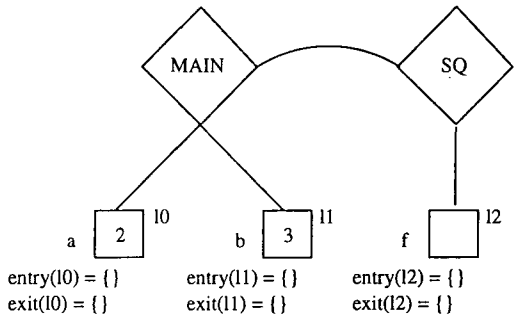
Figure 4.17: Data dependence analysis for example program cont.

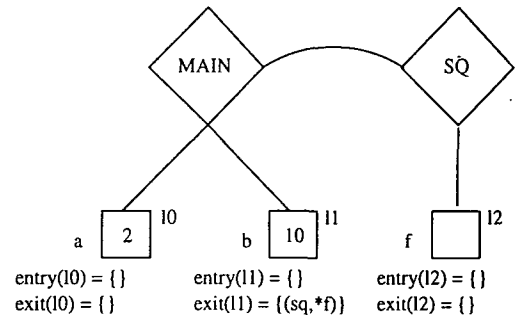
8. `enter sq()`



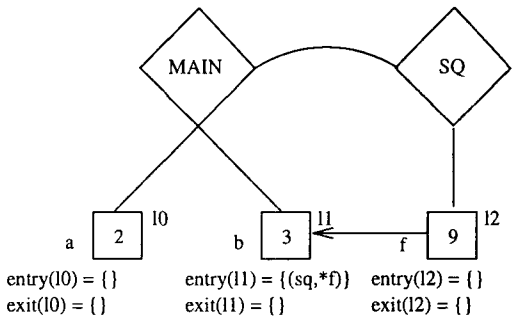


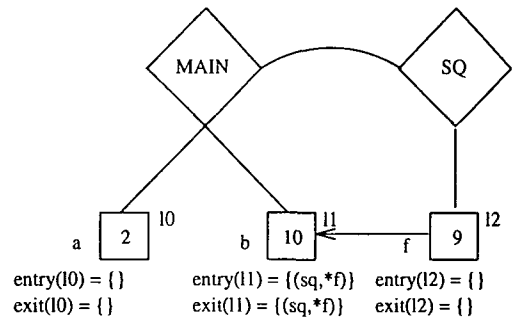
9. `f`





10. `(*f)++`





11. `return *f * *f`

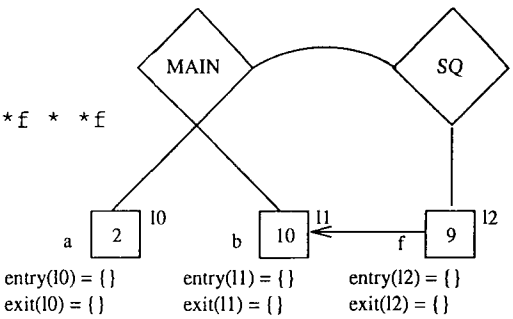


Figure 4.18: Data dependence analysis for example program cont.

```
1.  void main()
2.  {
3.    int sum;
4.    int i;
5.    i = sum = 0;
6.    while (i<20) {
7.      sum = CalcSum(sum,&i);
8.    }
9.    i = i;
10.   sum = sum;
11. }
```

```
12. int CalcSum(int s,int *j)
13. {
14.   Inc(j);
15.   if (s<100) {
16.     s = s + *j;
17.   }
18.   return s;
19. }
```

```
20. void Inc(int *x)
21. {
22.   *x = *x + 1;
23. }
```

Table 4.4: Example C program.

4.6 Example

A CCG for the program shown in table 4.4 is contained in figure 4.19. The program contains value and pointer parameters, value-returning functions, pointer variables and statements with embedded side effects. Three FCCGs are created, for `main`, `CalcSum` and `isq` respectively.

main

- Vertices

  `enter main()`

  `sum = 0`

  `i =`

  `while (i < 20)`

  `call CalcSum()`

  sum (actual parameter)

  `&i` (actual parameter)

  `sum =`

  `i = i`

  `sum = sum`

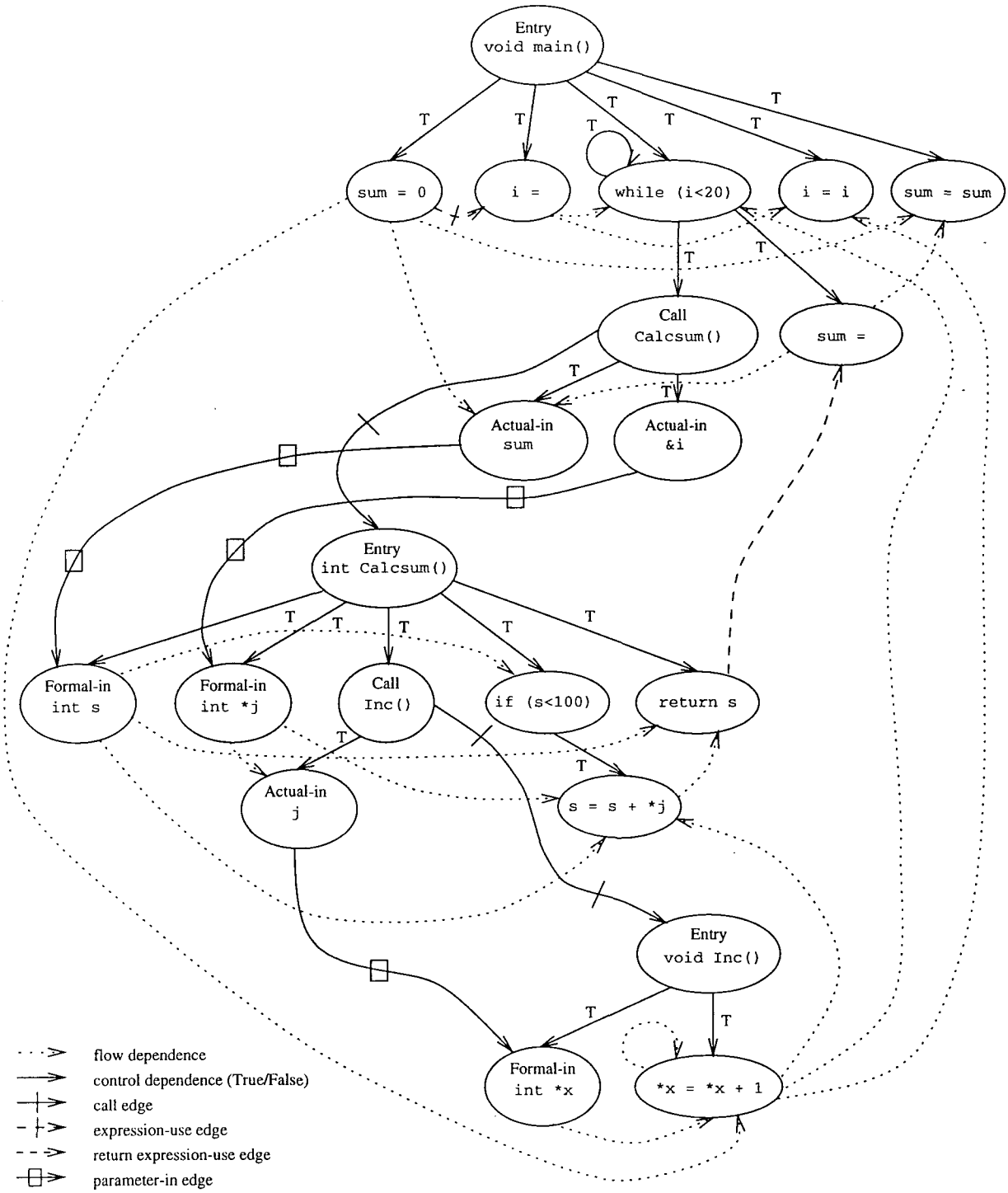


Figure 4.19: Example CCG.

- Flow dependencies

  (sum = 0)$\rightarrow_f$(sum)

  (sum = 0)$\rightarrow_f$(sum = sum)

  (i =)$\rightarrow_f$(while (i < 20))

  (i =)$\rightarrow_f$(i = i)

  (sum =)$\rightarrow_f$(sum)

  (sum =)$\rightarrow_f$(sum = sum)

- Control dependencies

  (enter main())$\rightarrow_c^T$(sum = 0)

  (enter main())$\rightarrow_c^T$(i =)

  (enter main())$\rightarrow_c^T$(while (i < 20))

  (enter main())$\rightarrow_c^T$(i = i))

  (enter main())$\rightarrow_c^T$(sum = sum)

  (while (i < 20))$\rightarrow_c^T$(call CalcSum())

  (while (i < 20))$\rightarrow_c^T$(sum =)

  (call CalcSum())$\rightarrow_c^T$(sum)

  (call CalcSum())$\rightarrow_c^T$(&i)

- Expression-use edge

  (sum = 0)$\rightarrow_{eu}$(i =)

CalcSum

- Vertices

  enter CalcSum()

  s (formal parameter)

  j (formal parameter)

  call Inc()

  j (actual parameter)

  if (s < 100)

  s = s + *j

`return s`

- Flow dependencies

  `(s)`$\rightarrow_f$`(if (s < 100))`

  `(s)`$\rightarrow_f$`(s = s + *j)`

  `(s)`$\rightarrow_f$`(return s)`

  `(j)` (formal)$\rightarrow_f$`(j)`

  `(j)` (formal)$\rightarrow_f$`(s = s + *j)`

  `(s = s + *j)`$\rightarrow_f$`(return s)`

- Control dependencies

  `(enter CalcSum())`$\rightarrow_c^T$`(s)`

  `(enter CalcSum())`$\rightarrow_c^T$`(j)` (formal)

  `(enter CalcSum())`$\rightarrow_c^T$`(call Inc())`

  `(enter CalcSum())`$\rightarrow_c^T$`(if (s < 100))`

  `(enter CalcSum())`$\rightarrow_c^T$`(return s)`

  `(call Inc())`$\rightarrow_c^T$`(j)` (actual)

  `(if (s < 100))`$\rightarrow_c^T$`(s = s + *j)`

- Annotations

  $preserved(CalcSum)$ = {}

  $live\_on\_entry(CalcSum)$ = {`(*j, i/main/2)`}

  $live\_on\_exit(CalcSum)$ = {`(*j, i/main/2)`}

Inc

- Vertices

  `enter Inc()`

  `x` (formal parameter)

  `*x = *x + 1`

- Flow dependencies

  `(x)`$\rightarrow_f$`(*x = *x + 1)`

  `(*x = *x + 1)`$\rightarrow_f$`(*x = *x + 1)`

- Control dependencies

  `(enter Inc())`$\rightarrow_c^T$`(x)`

  `(enter Inc())`$\rightarrow_c^T$`(*x = *x + 1)`

- Annotations

  $preserved(Inc)$ = {}

  $live\_on\_entry(Inc)$ = {`(*x, i/main/2)`}

  $live\_on\_exit(Inc)$ = {`(*x, i/main/2)`}

Interprocedural edges

- Call interface edges

  `(call CalcSum())`$\rightarrow_{call}$`(enter CalcSum())`

  `(sum)`$\rightarrow_{bind}$`(s)`

  `(&i)`$\rightarrow_{bind}$`(j)`

  `(return s)`$\rightarrow_{ret}$`(sum =)`

  `(call Inc())`$\rightarrow_{call}$`(enter Inc())`

  `(j)`$\rightarrow_{bind}$`(x)`

- Interprocedural flow dependencies

  `(i =)`$\rightarrow_f$`(*x = *x + 1)`

  `(*x = *x + 1)`$\rightarrow_f$`(s = s + *j)`

  `(*x = *x + 1)`$\rightarrow_f$`(while (i < 20))`

  `(*x = *x + 1)`$\rightarrow_f$`(i = i)`

4.7 Summary

This chapter has described a new dependence-based IPR, the Combined C Graph (CCG), which extends earlier IPRs to model many features of the C language. The CCG is a fine-grained representation allowing the modelling of expressions with embedded side effects and control flows. Pointer variables, pointer parameters, value-returning functions, structures, arrays and the control constructs `break`, `continue`, `switch` and `goto` are introduced. This chapter has discussed the approaches and techniques employed to represent each of these features.

A variety of programming views may be constructed from the CCG. Simple program slices and ripple analyses can be constructed with accuracy enhanced as a result of the 'fine-grained' approach employed in the CCG. Definition-use pairs, call graphs, control dependence and flow-sensitive data flow information may be easily constructed.

A formal definition of the vertices and edges comprising the CCG has been outlined, followed by algorithms to permit the construction of the representation. Finally an example of the CCG for a small C program has been described.

Chapter 5

Implementation

This chapter describes a prototype implementation of the CCG representation. This prototype system allows a software maintainer to construct a CCG for a C program, to construct 'view-forming' queries on this representation and to view graphically the relationships represented by the CCG.

The subject C source code is translated into Prolog facts, forming the *CCG fact base*. Meta programs are then written to analyse this fact base and to form views of the CCG. A further translation step allows the CCG itself to be viewed using a graphical display tool.

Section 5.1 gives an architectural overview of the CCG system. Section 5.2 outlines the Prolog facts comprising the CCG fact base. Section 5.3 describes meta programs which may be constructed to form views of the subject system and to support maintenance activities. Finally section 5.4 shows the graphical representation of the CCG fact base.

5.1 System Architecture

The components and flow of information within the CCG prototype system are shown in figure 5.1.

The first stage in the construction of the CCG representation is the translation of the subject set of files, the *C sources* into an equivalent Prolog representation, the *partial CCG fact base*. The 'rules' and 'facts' of the Prolog system make it ideal for representing the vertices and edges of the CCG in a relational form. The C sources may be either ANSI[5] or K&R[50] C but must have been preprocessed by the C Preprocessor *cpp*. The translation program *ccg_trans* is written using the *yacc* compiler-compiler, using the C grammar contained in [51]. This translator is based on the PERPLEX C analysis tool described by Bünter[14]. The PERPLEX system forms a generic Prolog fact base to allow the easy development of program analysis and other software engineering

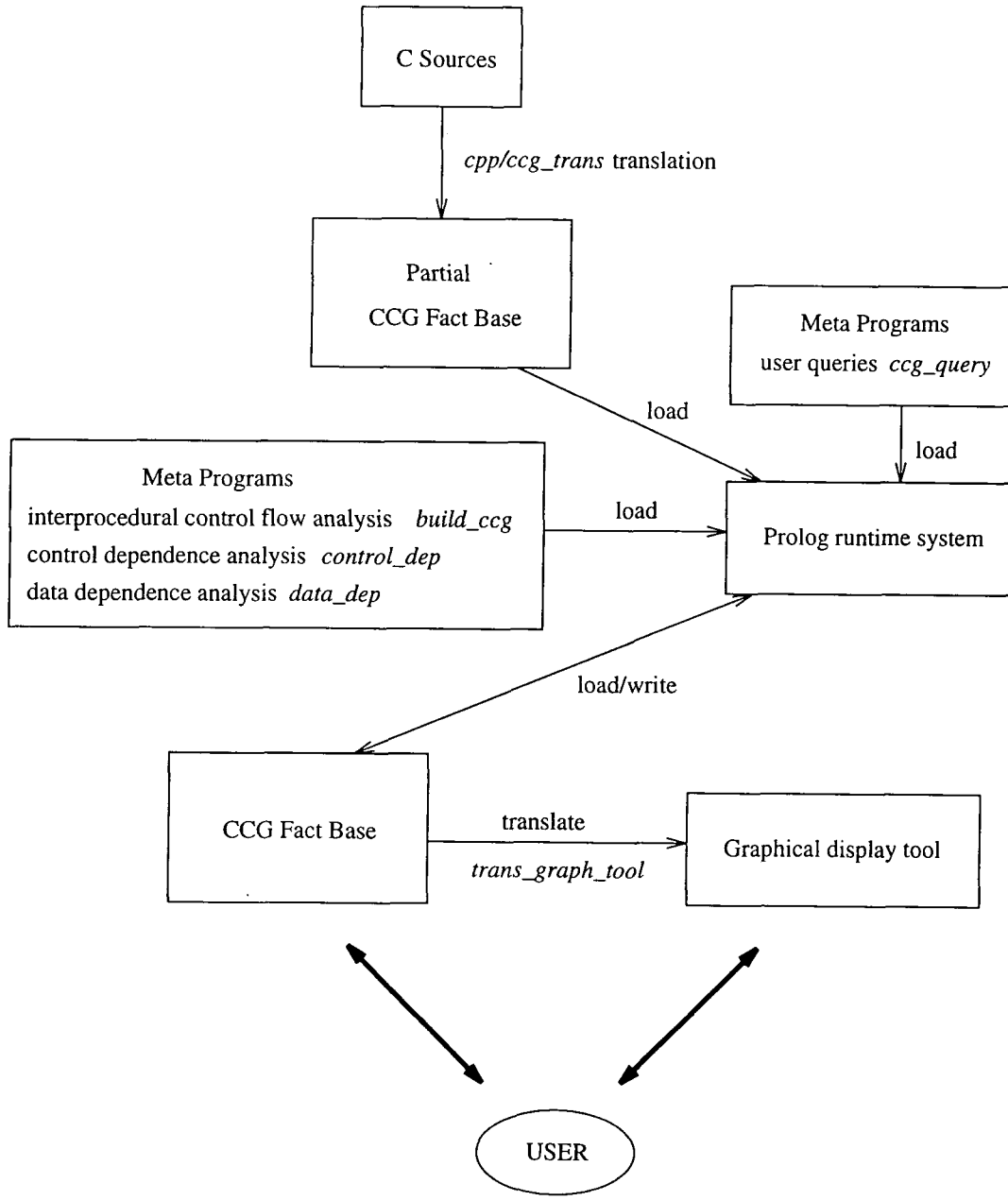


Figure 5.1: CCG prototype system architecture.

programs. The CCG translator *ccg_trans* modifies the PERPLEX system to generate the partial CCG fact base. The C sources are translated file by file and then linked together, allowing incremental update of the partial CCG fact base when changes are confined to a single file. The partial CCG fact base comprises Prolog facts representing the vertices of the CCG, the object and types of the subject program and the control flow within each C function. The interprocedural control edges, control dependence information and data dependence information are not contained in this partial fact base.

The second stage of the CCG construction is to augment the partial CCG fact base to give the complete CCG representation, the *CCG fact base.* This process is achieved using the Prolog runtime system which loads the partial CCG fact base together with Prolog meta programs implementing algorithms described in section 4.5. *Build_ccg* first constructs additional Prolog facts representing the program's call, parameter binding and return relationships. Control and data dependence analyses are then performed on the current fact base to produce the complete CCG fact base. The control dependence algorithm, making extensive use of the program flow graph, post-dominator sets and relations can be easily implemented as a Prolog meta-program. The semantic effects of each CCG vertex are represented in the Prolog fact base and hence the data dependence algorithm can also be implemented in Prolog, making use of Prolog lists to represent the abstract memory states. The control and data dependence algorithms are known as *control_dep* and *data_dep* respectively.

A software maintainer is able to use the CCG fact base in two ways. The Prolog runtime system may load further meta programs *ccg_query* to enable the maintainer to construct a variety of queries via the Prolog command shell, forming views of the subject system. The meta-programming capabilities of the Prolog language make it well suited for this purpose. The CCG fact base can also be translated into a form which may then be loaded into a graphical display tool. The Prolog meta program *trans_graph_tool* implements this translation step. The maintainer can then view directly the program relationships represented in the CCG.

5.2 CCG Fact Base

The CCG fact base is a Prolog representation of the vertices and edges comprising the CCG. The fact base is designed to provide an accurate representation of the subject C system and to allow the construction of a comprehensive set of analysis meta programs. The prototype system covers the C language with the exception of information on constants, casts, case-labels, operators in expressions, initialisation of objects within declarations and pointers to functions. The flow-sensitive data flow

information of the CCG is also not implemented in the prototype system.

The CCG fact base comprises thirteen different fact types. Eight of these facts represent the different vertices and edges of the CCG. These facts are:

- *node fact* - represents the type and semantic effects of a CCG vertex.
- *flow fact* - represents an intraprocedural or interprocedural flow dependence between two CCG vertices.
- *control fact* - represents a control dependence between two CCG vertices.
- *expuse fact* - represents an expression-use edge between two CCG vertices.
- *lvaldef fact* - represents an lvalue-definition edge between two CCG vertices.
- *return-expuse fact* - represents a return-expression-use edge between two CCG vertices.
- *call fact* - represents a call edge between a CCG call vertex and CCG entry vertex.
- *bind fact* - represents a binding edge between two CCG vertices.

The remaining five facts are not defined as part of the CCG itself but represent the program's control flow graph and additional information on the components of the subject C system.

- *edge fact* - represents the control flow between two CCG vertices.
- *file fact* - represents a file which is a member of 'C sources', the subject C system.
- *type fact* - represents a name defined as a type.
- *tag fact* - represents a tag name given in a `struct`, `union` or `enum` definition.
- *object fact* - represents the declaration or definition of a program component.

A full description of each of the thirteen fact types is contained in appendix A. A complete Prolog fact base produced following the analysis of a small C program is contained in appendix B.

5.3 Meta Programs

The CCG fact base may be interpreted as a set of relations and consequently can be treated as a relational database. Prolog makes use of position to allow access to the particular fields of a fact or 'relation' and hence meta programs can be written to imitate database queries.

The current prototype system implements a variety of analysis meta-programs providing a maintainer with different programming level views of the subject C system. Information can be constructed on program objects and types, function call relationships, parameter information, control dependencies, definition-use associations, program slices and ripple effects. Each meta program query can be executed using the Prolog wild card ('_') in place of any of the arguments to produce all matching solutions.

The implemented meta programs are:

- list(Selection) - lists the files, functions, globally defined objects or types specified by Selection.
- module_call(File1, File2) - produces the module call dependency between File1 and File2, showing calls in File1 to function calls in File2.
- fun_call(Function1, Function2) - lists function pairs where Function1 calls Function2.
- global_call(File, Function, ObjectName) - lists globally defined objects ObjectName which are referenced by Function within File.
- fun_call_chain(Function1, Function2) - lists call chains from Function1 to Function2.
- find_def(ObjectName) - extracts information from the definition of objects named ObjectName.
- formals(File, Function) - lists formal parameters of Function within File.
- list_cont_dep(File, Function, From, To) - lists control dependencies from vertex From to vertex To of Function within File.
- list_nodes(File, Function, Number) - list vertex Number of Function within File.
- list_bind(File1, Function1, Number1, File2, Function2, Node2) - lists binding edges from vertex Number1 of Function1 within File1 to Number2 of Function2 within File2.
- list_bind_name(File, Function, Name) - lists binding edges to formal parameter Name of Function within File.
- cont_chain(File, Function, From, To) - lists control dependence chains from vertex From to vertex To of Function within File.
- def_use(File1, Function1, Number1, File2, Function2, Node2) - list definition-use associations from vertex Number1 of Function1 within File1 to Number2 of Function2 within File2.

- inter_def_use(File1, Function1, Number1, File2, Function2, Node2) - lists interprocedural definition-use associations from vertex Number1 of Function1 within File1 to Number2 of Function2 within File2.
- intra_def_use(File, Function, Number1, Number2) - lists intraprocedural definition-use associations from vertex Number1 to vertex Number2 of Function within File.
- slice(File, Function, Number) - constructs a program slice at vertex Number of Function within File.
- ripple(File, Function, Number) - constructs a ripple analysis or 'forward slice' from vertex Number of Function within File.

5.4 Graphical Display Tool

The CCG fact base may be translated via a simple Prolog meta program to a form which allows the CCG, or part of the CCG, to be displayed using a graphical display tool. The tool used was developed by Bodhuin[10]. An example display is shown in figure 5.2.

5.5 Summary

This chapter has described the prototype implementation of the CCG system. The input C programs are translated to a Prolog fact base representation which is then enhanced via control and data dependence algorithms implemented as Prolog meta programs. The resulting fact base can then be queried using further Prolog meta programs to construct programming level views. These queries are both flexible and simple to construct and provide a variety of views including definition-use pairs, control dependencies, call relationships, parameter information, program slices, ripple analysis and program component and type information.

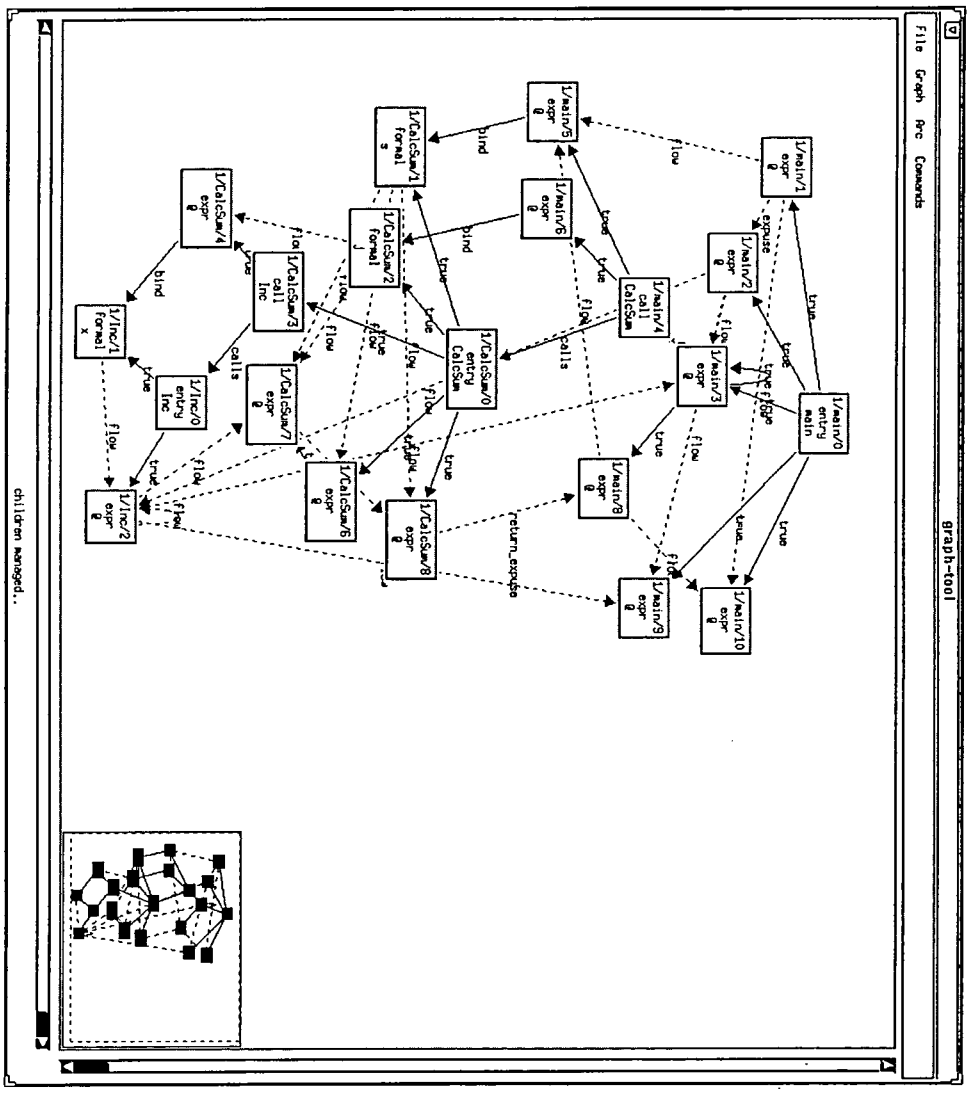


Figure 5.2: Graphical display of CCG.

Chapter 6

Application

This chapter describes the results achieved using the prototype CCG system outlined in chapter 5. A number of C programs of different sizes and involving different features of the C language were analysed using the system. Section 6.1 discusses the analysis of four small programs of up to 121 lines of code in length. Listings of these four programs are contained in appendix C. Section 6.2 describes the application of the CCG system to two larger programs of up to one thousand lines of code. Section 6.3 gives empirical results on the space requirements of the CCGs created in the previous sections and the time taken in the construction of these representations. Section 6.4 provides examples of the program views and information made available to maintainers by the CCG system. Finally an outline of the use of the system in various program understanding and maintenance tasks is given in section 6.5.

6.1 Analysis of Small C programs Using the CCG System

The prototype CCG system has been used to analyse a variety of C programs of different sizes and involving different C language features. This section describes the analysis of four small C programs of up to 121 lines of code. The programs are derived from a number of different sources and are chosen to illustrate the capabilities of the CCG in terms of the language coverage offered by the representation.

Table 6.1 describes the relative sizes of the programs analysed, ordered to reflect the number of lines of code and the relative 'complexity' of the programs in terms of the language features used. The columns represent the number of lines of code of the program, the number of functions making up the program, the number of standard library routines called, the total number of function calls, the total number of external, static and local variables and the number of assignments, both in

Program	Lines of Code	Number of Functions	Number of Std Library Routines	Number of Calls	Number of Variables	Number of Assigns	
						all	pointer
trityp	30	1	2	14	4	12	0
sum	25	3	0	2	5	10	1
linked_list	60	3	2	8	8	16	3
lines	121	7	4	15	26	38	7

Table 6.1: Sizes of subject programs.

total and through pointer dereferences. This final figure represents assignments such as `*p = 10` and `q->x = 20`, where the location defined is determined through a pointer value.

The C features of each subject program are described in table 6.2. The features listed are some of those provided by the CCG representation to extend earlier IPRs; value-returning functions, embedded side effects, embedded control flows, external variables, pointer variables, structure variables, array variables, value parameters, pointer parameters, dynamic memory allocation and standard library function calls.

A discussion of the results achieved with each of the four subject programs follows.

6.1.1 Trityp

This program implements 'Ramamoorthy's triangle', a simple algorithm to determine from the lengths of the three sides of a triangle, whether the triangle is isosceles, equilateral, right-angled, acute or obtuse. The lengths of the sides must be input in descending order. The program is thirty lines long and is a single function program calling only two standard library routines. Only external integer variables are involved.

The CCG for **trityp** contained refined vertices to model the embedded control flows caused by the short circuit evaluation of the logical 'and' and 'or' operators. Vertices were created for each sub-expression of the logical operator. For example, the statement:

```
(a >= b && b >= c)
```

produced two CCG vertices `a >= b` and `b >= c`. Control dependencies were created to reflect the control structure arising from the short circuit evaluation.

Features	trityp	sum	linked_list	lines
Value-returning functions		•	•	•
Embedded side effects		•	•	•
Embedded control flows	•			•
External variables	•		•	•
Pointer variables		•	•	•
Structure variables			•	
Array variables				•
Value parameters	•	•	•	•
Pointer parameters		•	•	
Dynamic allocation/ Recursive structures			•	
Standard library calls	•		•	•

Table 6.2: C features of subject programs.

'Stub' routines were created for the two standard library routines `scanf` and `printf` reflecting the external interface of the routines. Call and binding edges were created connecting the FCCGs for the two functions to the CCG for **trityp**.

6.1.2 Sum

This program is the example program shown in section 4.6 and equivalent to that in section 3.2.7. The program is similar to those used as examples by Horwitz et al[45], Harrold and Malloy[38][39] and Livadas and Croll[62] to illustrate other IPRs. Each of these authors includes a short program involving pass by reference parameters to increment a 'counter' variable. **Sum** calculates the sum of the series of the positive integers from 1 up to a maximum total of 100. The program conforms to ANSI C, is 25 lines long and comprises three functions. **Sum** makes use of value returning functions, pointer variables, pointer parameters and side effects within expressions.

The `CalcSum` function returns an integer value which is assigned to the `sum` variable within the `main` function. The CCG contained a return-expression-use edge to reflect this relationship. An expression-use edge was also produced connecting refined vertices `sum = 0` and `i =` created from the statement `i = sum = 0`.

Data dependence analysis of pointer variables was achieved successfully with variables `*j` and `*x` being decomposed to the pointer and referenced objects. Pointer parameters were also handled

successfully with formal parameter vertices created for the value parameters `j` and `x` and these variables updated to point to the objects referenced by the corresponding actual parameters `&i` and `j`. A number of interprocedural flow dependencies were created, reflecting the effects of the use of pointer parameters in the program.

6.1.3 Linked_list

This program is an example program taken from an ANSI C programming guide published by the Cambridge University Engineering Department[64]. The program creates a singly linked list structure, maintaining pointers to the head and tail of the list. New items are added to the tail of the list and the list traversed to print each item. The program comprises sixty lines of code and is made up of three functions, involving pointer variables, pointer parameters, structure variables, recursive structures, dynamic memory allocation, side effects within expressions and standard library calls.

The **linked_list** program contains many of the C features which may be represented using the CCG. 'Stub' routines were created for the standard library routines `printf` and `malloc`. The FCCGs for these routines were correctly connected to the CCG for **linked_list**.

Side effects within expressions occur in the following statements:

```
tail = add_list_item(tail, 5);
tail = add_list_item(tail, 7);
tail = add_list_item(tail, 2);
new_list_item = (list_item*) malloc(sizeof(list_item));
```

Call vertices were created to model the calls of `add_list_item` and `malloc`. Refined vertices were created for the assignments `tail =` and `new_list_item =`. Return-expression-use edges connected the return vertices of `add_list_item` and `malloc` to the corresponding vertices.

Pointer parameters were modelled successfully with formal parameter vertices created for the value parameters `argv[]` and `entry`. Data dependence analysis of **linked_list** was complicated by the presence of external pointer variables, structure variables and a dynamic allocation statement. The program's pointer variables were correctly decomposed to the pointer and referenced object parts, whilst the program's structure variables of type `list_item`:

```
typedef struct _list_item {
  int val;
  struct _list_item *next;
} list_item;
```

were broken down and analysed in terms of the fields `val` and `next`. The program's dynamic allocation statement:

```
new_list_item = (list_item*) malloc(sizeof(list_item));
```

presented greater difficulties as user interaction was required to indicate the 'shape' of the allocated structure. The user must input the fields making up the structure as the prototype system is unable to determine this information from an allocation statement. However, data dependence analysis proceeded successfully and produced a correct representation of the program's flow dependencies, both intraprocedural and interprocedural.

6.1.4 Lines

This program is taken from Kernighan and Ritchie's standard text on C, *The C Programming Language*[50]. The program is not contained explicitly in the book but is given as a series of examples to illustrate the use of pointers and functions. An input routine reads lines of character input, allocating space from a character array. A pointer array accesses each individual line. A quicksort routine then sorts the pointer array alphabetically and the lines are finally printed in order. The program is 120 lines of K&R C and is made up of seven functions. Simple arrays, arrays of pointers, array parameters, pointer variables, side effects within expressions, control flow within expressions and standard library calls are each used.

Lines contains a number of side effects and control flows embedded within expressions. The CCG system successfully detected nine embedded side effects and in each case created refined vertices and connecting expression-use edges. The side effects found were both embedded assignments and postfix increments, for example `s[i++]`. The use of logical 'and' and 'or' operators produces a number of embedded control flows. These were analysed successfully and control dependencies created accordingly.

Lines makes use of an external character array `allocbuf` and an external array of pointers `lineptr`. Data dependence analysis proceeded successfully in the presence of these array variables, correctly analysing the use of the arrays as actual parameters of both functions of **lines** and standard library routines. However, the accuracy of the resulting flow dependencies was limited as arrays are treated as 'aggregate' variables and no attempt is made to analyse specifically the individual elements of the array. For example, the statement:

```
lineptr[nlines++] = p;
```

is considered to define the entire `lineptr` array rather than the individual element concerned. The aggregate information is still of use in presenting a maintainer with a more general view of the use of an array throughout the subject program.

Lines contains a number of standard library calls and 'stub' routines were created for each of `getchar`, `printf`, `strcmp` and `strcpy`. Appropriate call and binding edges were created connecting these FCCGs to the CCG for **lines**.

6.1.5 Summary

For each of the small C programs analysed, **trityp**, **sum**, **linked_list** and **lines**, correct CCG representations were constructed. The test programs exhibited many C language features, namely value-returning functions, embedded side effects, embedded control flows, external variables, pointer variables, structure variables, array variables, value parameters, pointer parameters, dynamic memory allocation and standard library calls. Each of these features was analysed correctly by the prototype system. However, user interaction is required to produce stub routines for standard library functions and to describe the 'shape' of any dynamically allocated variables.

6.2 Analysis of Larger C Programs Using the CCG System

Any practical software maintenance IPR must allow large programs to be represented. This section describes the analysis of two larger C programs of up to one thousand lines of code. The analysis attempts to demonstrate the applicability of the CCG representation to large industrial-sized systems.

Table 6.3 shows the relative size of the two programs, the number of functions in each program, the number of standard library routines, the total number of function calls and the total number of external, static and local variables.

A discussion of the two programs and the results achieved with each follows.

6.2.1 Knap

The **knap** program solves the classic knapsack problem, as described by Aho et al[2]. Given a collection of positive integers representing the weights of items, is there a selection of weights which totals a given target t? The weights may also be given utility values and the selection chosen to maximise the utility of the items carried, subject to a weight constraint. The program contains twelve functions and over five hundred lines of ANSI C code. The program uses 21 standard library

Program	Lines of Code	Number of Functions	Number of Std Library Routines	Number of Calls	Number of Variables	Number of Conditionals
knap	562	12	21	74	74	70
migrate	1006	15	16	294	231	172

Table 6.3: Sizes of subject programs.

routines from the `stdlib` and `string` modules. **Knap** makes extensive use of pointers, arrays and recursive structures, with four `calloc` memory allocation calls.

The CCG system analysed the **knap** program statements successfully, creating 692 refined vertices. Expressions with embedded side effects were detected correctly, producing 43 expression-use edges. Interprocedural control analysis produced call, parameter binding and return-expression-use edges correctly modelling the 74 function calls. Control dependencies were constructed successfully for each function of the **knap** program. This task involved the analysis of seventy conditionals, including `switch` statements and embedded control flows produced by twelve logical 'and' and 'or' operators. Stub routines were created for each standard library function called.

Disappointing results were obtained from the data dependence analysis of the **knap** program. The Prolog data dependence program was unable to complete the analysis, producing local stack space errors. This result is due to the simple implementation of the data dependence construction algorithm. The implementation attempts to analyse the data flow effects on all paths through the subject program, halting the analysis on each path only when a store graph is repeated at the current CCG vertex, i.e. the current abstract memory layout exactly matches a previous memory layout at the same program vertex. Whilst this simple approach was successful when analysing the small programs described in section 6.1, the number of function calls and conditionals of the **knap** program make the number of possible paths large. The presence of dynamic memory allocation statements further complicates the problem as store graphs may differ in both 'shape' as well as 'content' as new dynamic variables are added to the store graph. The fixed point solution is therefore more difficult to achieve.

The degree of user interaction required also caused some problems. The user must describe the fields of the allocated data structure whenever the data dependence analysis reaches a dynamic memory allocation statement. The user may alternatively specify that no further variables are to

be allocated on the current path, ensuring that the depth of a recursive data structure is bounded. Since an allocation statement may be reached on many paths through the program, the user is frequently required to provide input. The current path being analysed is also not clear and hence it becomes difficult for the user to determine at which point to cease the allocation of new dynamic structures.

6.2.2 Migrate

The **migrate** program is taken from a biology research project, and simulates the spread of of an invading species through an environment. The program has 1006 lines of K&R C code, and is made up of fifteen functions calling a further sixteen standard library routines from the `stdio` and `math` modules. Whilst the **migrate** program is approximately twice as long as the **knap** program, the code involved is simpler. Pointer usage is limited and the code contains no dynamic memory allocation statements. However, the program makes extensive use of multi-dimensional arrays and external variables.

Over 1800 CCG vertices were created from the source statements with 68 expression-use edges to connect the refined vertices modelling embedded side effects. Again interprocedural control analysis was achieved successfully producing call, parameter binding and return-expression-use edges to represent the 294 function calls. The **migrate** program contains 172 conditional statements, predominantly `if` and `for` structures, and fifteen logical 'and' and 'or' operators giving embedded control flows. Control dependence edges were constructed for each function of the program, correctly analysing these conditionals and boolean operators. Stub routines were created for each standard library routine.

Like the **knap** program, problems were found only during the data dependence analysis stage. Again the Prolog data dependence program produced local stack space errors. The number of conditionals in the program gives rise to a large number of possible paths through the program, although the absence of dynamic memory allocation statements simplifies the achievement of a fixed point solution.

6.2.3 Summary

Both **knap** and **migrate** produced similar results when analysed with the prototype CCG system. In each case the analysis produced correctly the CCG vertices, expression-use, call, binding and return-expression-use edges. Control dependence edges were constructed successfully, despite the large numbers of conditional statements and boolean operators in both programs.

However, data dependence analysis was in both cases unsuccessful, producing local stack space errors from the Prolog system. This result is due to the simplistic prototype implementation which requires analysis of the subject program on all paths to produce successful results. The number of conditional statements in both programs, combined with the further complication of dynamic memory allocation in the case of the **knap** program, produced too many possible paths and lead to the run time error observed. The user interaction required to analyse dynamic memory allocation was also unsatisfactory.

These results demonstrate the feasibility of the CCG in representing large programs. CCG vertex, control dependence and interprocedural control information computation each scaled up successfully from the small programs analysed in the previous section. However, a more sophisticated implementation is required for the data dependence analysis step. Automation of the analysis of dynamic allocation statements is also required.

6.3 Empirical Results

The prototype system comprises four programs which together produce a CCG for a subject C program. These are:

- The CCG translator *ccg_trans.*
- Interprocedural control flow analysis programs *build_ccg.*
- Control dependence analysis program *control_dep.*
- Data dependence analysis program *data_dep.*

A fifth program *trans_graph_tool* converts a complete CCG into the input form required by the graphical display tool.

Each of the subject programs **trityp, sum, linked_list, lines, knap** and **migrate** was analysed and empirical results obtained for the time taken for each step and the space requirements of the resulting CCG. In each case these results were achieved running on a SUN 670.

6.3.1 CCG Construction Time

Empirical results for the construction of the CCG are shown in table 6.4 The *ccg_trans* figure represents the elapsed system time in seconds given by the UNIX *time* command. All other figures represent cpu time in seconds determined using the Prolog *time* command.

Program	*ccg_trans* (s)	*build_ccg* (s)	*control_dep* (s)	*data_dep* (s)	*trans_graph_tool* (s)	Total (s)
trityp	0.90	0.15	4.90	1.00	0.53	7.48
sum	0.60	0.02	0.80	0.62	0.15	2.19
linked_list	0.60	0.17	4.87	5.57	0.62	11.83
lines	1.00	0.75	16.28	25.15	2.90	46.08
knap	1.50	12.77	708.30	-	35.12*	-
migrate	2.60	106.42	4748.20	-	243.53*	-

* CCGs for knap and migrate do not include *flow* dependencies.

Table 6.4: CCG construction times.

The CCG translation step *ccg_trans*, producing the 'partial' CCG, was completed in all cases within three seconds of system time, and increased only slowly as the size of the subject program increased. The two largest C programs, **knap** and **migrate** were analysed on average at a rate of 12kb of C source per second by the *ccg_trans* program. The time taken by *ccg_trans* was not affected seriously by the C features used in the subject code. These results are encouraging for the analysis of large programs.

The interprocedural control flow dependence analysis step *build_ccg* took up to 106 seconds of cpu time for the **migrate** program. The time taken on this step increased approximately linearly as the number of functions, function calls, parameters and value-returning functions of the subject program increased.

The time taken in the control dependence analysis step *control_dep* increased up to around 4750 seconds of cpu time for the **migrate** program. The control dependence algorithm is executed on the control flow graph of every function and on the control flow subgraph of each actual parameter list of every function call. The time spent in control dependence analysis increased with the number of functions and as the length of each function and the number of conditional statements increased. The number of function calls and the number and 'complexity' of the subject program's actual parameters also affected the time taken in this step. The results obtained from control dependence analysis were not encouraging but can be explained. The first-step of the control dependence calculation is to determine the program control flow graph's post-dominators. This is achieved using a simple but comparatively inefficient algorithm. More efficient methods are available and should considerably reduce the time taken.

Program	C file(s)		CCG representation	
	lines of code	space (kb)	no. of facts	space (kb)
trityp	30	0.8	156	6.1
sum	25	0.3	81	3.4
linked_list	60	1.1	179	8.9
lines	121	1.9	462	19.1
knap	562	13.9	1516*	73.5*
migrate	1006	37.8	4068*	234.7*

* CCGs for knap and migrate do not include *flow* dependencies.

Table 6.5: CCG space requirements.

The time required by the data dependence analysis step *data_dep* similarly increased with program size, and for the smaller subject programs up to around 25 seconds for the **lines** program. No results were achieved for the larger programs **knap** and **migrate**. The increase in analysis time was related to the number of lines of code of the subject program, the number of variables in the program, the number of assignments in the program and the number of paths through the program. As described in section 6.2, the data dependence implementation employed was a simplistic one and a more efficient technique will be required to analyse larger programs.

The *trans_graph_tool* program required time related to the size of the CCG representation. On average around 20 facts per second are processed for the larger **knap** and **migrate** programs.

The total real time taken to construct the CCG ranges from around two seconds for the **sum** program to 46 cpu seconds for the **lines** program. For each program analysed the main contributors to this figure are the *control_dep* and *data_dep* steps, which together contribute up to 75% of the total time. Complete CCG construction was not achieved for the larger **knap** and **migrate** programs.

6.3.2 CCG Space Requirements

Empirical results for the space requirements of the CCG representation are shown in table 6.5. This table describes the number of lines of code and the space requirements of the subject programs, together with the number of facts and the space requirements of the corresponding CCG representations. Figures for the CCGs representing **knap** and **migrate** do not include flow dependencies.

On average the number of facts making up the complete CCG for each subject program was around three to four times the number of lines of code of the program. This figure was dependent

on the actual statements making up the subject program. For example, expressions with embedded side effects or control flows produce additional refined CCG vertices and corresponding expression-use, lvalue-definition and control dependence edges.

The space requirements of the CCG were up to an order of magnitude larger than the subject program. Whilst this was not particularly encouraging, the space requirements of the CCG can be reduced by producing less verbose Prolog facts.

6.4 Program Views

This section presents the program views and information that are made available to a maintainer using the prototype CCG system. The examples given are each taken from the programs **trityp**, **sum**, **linked_list** and **lines** listed in appendix C and described in section 6.1. The views are divided into five types - call graphs views, control dependence views, definition-use views, program slice and ripple analysis views and finally program components. Each view-forming program is implemented as a Prolog meta program. The output produced by the prototype system is in text form but in many cases may be translated and viewed using a graphical display tool.

6.4.1 Call Graph Views

A maintainer can construct views of the program call graph using the fun_call command. The following example uses wildcards to list all the call relations of the **linked_list** program.

```
?- fun_call(_,_).

Function Call Graph

  fc(add_list_item, malloc)
  fc(add_list_item, printf)
  fc(main, print_list_items)
  fc(main, add_list_item)
  fc(print_list_items, printf)
```

The former function of each pair will contain a call to the latter. This call graph may be viewed using a graphical display tool, producing the output shown in figure 6.1.

The fun_call_chain command can also be used to give specific call chains. The following example lists call chains between the functions `main` and `Inc` of the **sums** program.

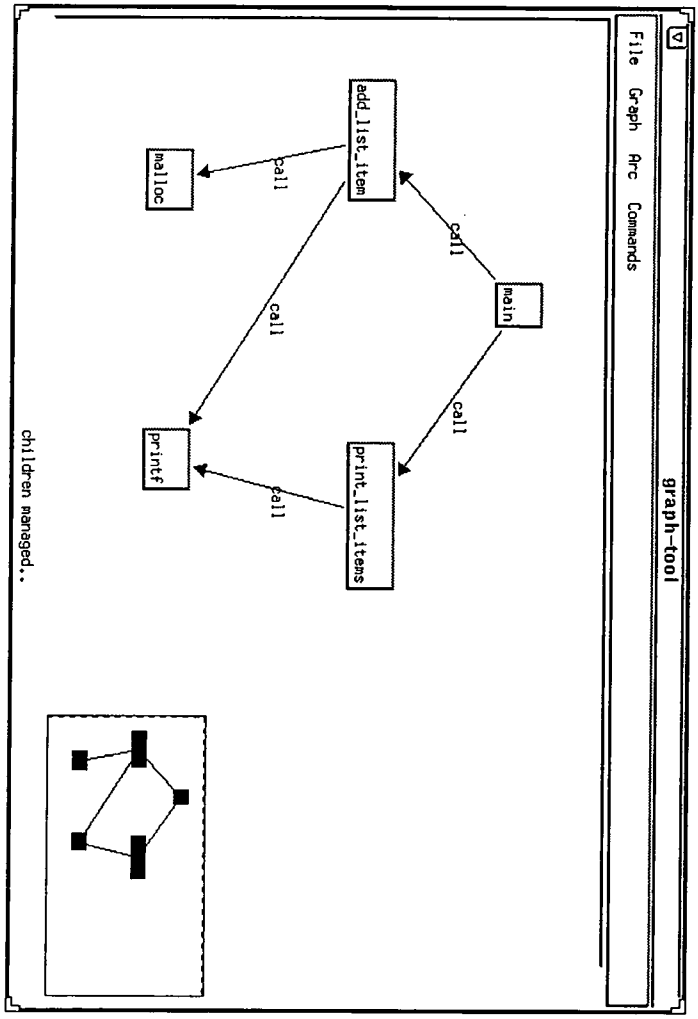


Figure 6.1: Call graph for **linked_list**.

```
?- fun_call_chain(main,'Inc').

The Chain List

  From main to Inc: [main->CalcSum->Inc]

1 chains have been found
```

6.4.2 Control Dependence Views

Using the list_cont_dep command a maintainer can view the control dependence subgraph of a function of the subject program or can construct a more specific view showing the control subgraph below a given statement. The following query constructs a control dependence subgraph for the main function of **linked_list**.

```
?- list_cont_dep(1,main,_,_).

Control Dependencies :-

  control(1, main, 0, 20, true)
  control(1, main, 0, 19, true)
  control(1, main, 0, 15, true)
  control(1, main, 0, 14, true)
  control(1, main, 0, 10, true)
  control(1, main, 0, 9, true)
  control(1, main, 0, 5, true)
  control(1, main, 0, 4, true)
  control(1, main, 0, 3, true)
  control(1, main, 0, 2, true)
  control(1, main, 0, 1, true)
  control(1, main, 10, 12, true)
  control(1, main, 10, 11, true)
  control(1, main, 15, 17, true)
  control(1, main, 15, 16, true)
  control(1, main, 5, 7, true)
  control(1, main, 5, 6, true)
```

This subgraph may be viewed as shown in figure 6.2.

The user can also determine call dependence chains between two statements using the cont_chain command. The following shows the call dependence chain between vertex 7 (a >= b) and vertex 40 ("Acute") of main within the **trityp** program.

```
?- cont_chain(1,main,7,40).
```

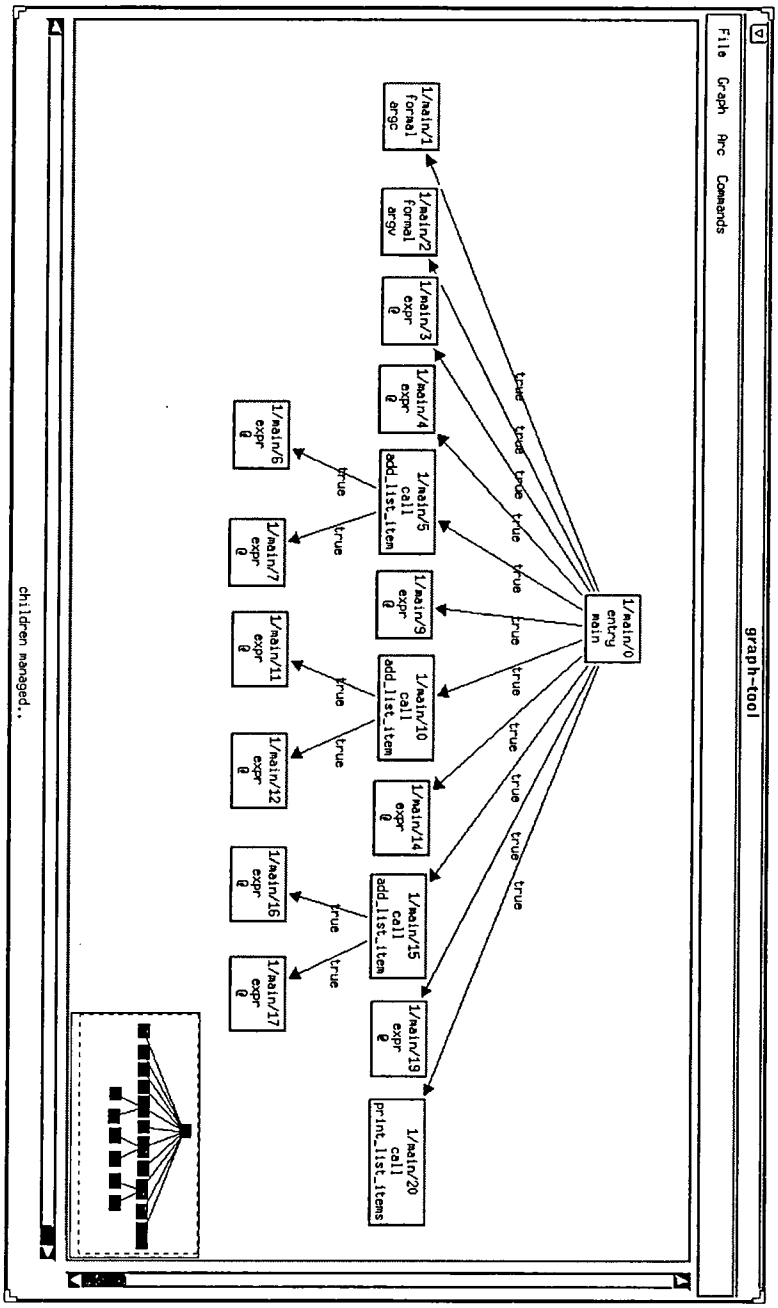


Figure 6.2: Control dependence subgraph for `main` in **linked_list**.

```
Control dependence chains

  file(1, main): from node 7 to 40: [7-true->8-false->39-true->40]
  file(1, main): from node 7 to 40: [7-false->39-true->40]

2 chains have been found
```

These chains represent the control dependencies arising from the short circuit evaluation of the boolean expression ((a >= b) && (b >= c)).

6.4.3 Definition-use Views

A maintainer is able to view definition-use information, making use of wildcards to provide more or less specific information. The def_use command lists definition-use pairs between the specified CCG vertices. For example, the following query lists definition-use pairs from vertex 6 (new_list_item =) of add_list_item within **linked_list**.

```
?- def_use(1, 'add_list_item', 6, _, _, _).

Def-use pairs :-

  flow(1, add_list_item, 6, 1, add_list_item, 12)
  flow(1, add_list_item, 6, 1, add_list_item, 18)
  flow(1, add_list_item, 6, 1, add_list_item, 19)
  flow(1, add_list_item, 6, 1, add_list_item, 20)
  flow(1, add_list_item, 6, 1, add_list_item, 8)
```

The result shows definition-use pairs to vertices 12, 18, 19, 20 and 8, which represent the uses:

```
entry->next = new_list_item;
new_list_item->val = value;
new_list_item->next = NULL;
return new_list_item;
head = new_list_item;
```

Variables used before defined can be found be specifying only a use site. The result will list all the corresponding definition-use associations. An empty solution indicates a previously undefined variable.

The command inter_def_use shows only interprocedural definition-use pairs. The following query shows all interprocedural definition-use associations within the **lines** program.

```
?- inter_def_use(_,_,_,_,_,_).

Interprocedural def-use pairs :-

  flow(1, main, 1, 1, alloc, 2)
  flow(1, main, 1, 1, alloc, 3)
  flow(1, main, 1, 1, alloc, 4)
  flow(1, swap, 6, 1, qsort, 15)
  flow(1, swap, 6, 1, qsort, 16)
  flow(1, readlines, 24, 1, swap, 4)
  flow(1, readlines, 24, 1, swap, 5)
  flow(1, readlines, 24, 1, writelines, 7)
  flow(1, readlines, 18, 2, strcpy, 3)
```

These results represent the dependencies:

```
(allocp = allocbuf) →f(allocbuf + ALLOCSIZE - allocp >= n)
(allocp = allocbuf)→f(allocp += n)
(allocp = allocbuf)→f(return allocp - n)
(temp = v[i])→f(v[i])
(temp = v[i])→f(v[left])
(lineptr[] = p)→f(temp = v[i])
(lineptr[] = p)→f(v[i] = v[j])
(lineptr[] = p)→f(*lineptr++)
(line[len - 1] = '∅')→f(strcpy)
```

The corresponding command intra_def_use lists only intraprocedural definition-use associations. The following command lists only intraprocedural definition-use associations within the main function of the **sum** program.

```
?- intra_def_use(1,main,_,_).

Intraprocedural def-use pairs :-

  flow(1, main, 8, 10)
  flow(1, main, 1, 10)
  flow(1, main, 2, 3)
  flow(1, main, 1, 5)
  flow(1, main, 8, 5)
  flow(1, main, 2, 9)
```

These results represent the dependencies:

```
(sum =) →f(sum = sum)
(sum = 0) →f(sum = sum)
(i =) →f(while (i < 20))
(sum = 0) →f(sum)
(sum =) →f(sum)
(i =) →f(i = i)
```

6.4.4 Program Slices and Ripple Analysis

A program slice at a given CCG vertex can be created using the slice command. A slice can only be taken at a CCG vertex rather than more specifically on a given variable at a given program statement. The resulting slice comprises slices on each variable used at the selected CCG vertex. The slice indicates those statements potentially affecting the value of each variable used. The program slice is given in terms of the CCG vertices involved; the present CCG system is unable to produce output in terms of the actual source code. The examples below are manually translated to give the actual source code of the resulting slice.

A slice on formal parameter `x` of `Inc` within the program **sum** is achieved using the following query.

```
?- slice(1,'Inc',1).

Slice :-

  node(1, CalcSum, 0)
  node(1, CalcSum, 3)
  node(1, Inc, 0)
  node(1, Inc, 2)
  node(1, main, 1)
  node(1, main, 0)
  node(1, main, 2)
  node(1, main, 3)
  node(1, main, 4)
  node(1, main, 6)
  node(1, CalcSum, 2)
  node(1, CalcSum, 4)
  node(1, Inc, 1)

13 nodes in the slice
```

The resulting slice is listed in table 6.6 and shown in figure 6.3. A second slicing example, showing a slice on `ptr_to_list_item = head` of function `print_list_items` in **linked_list** produces the result contained in table 6.7 and figure 6.4.

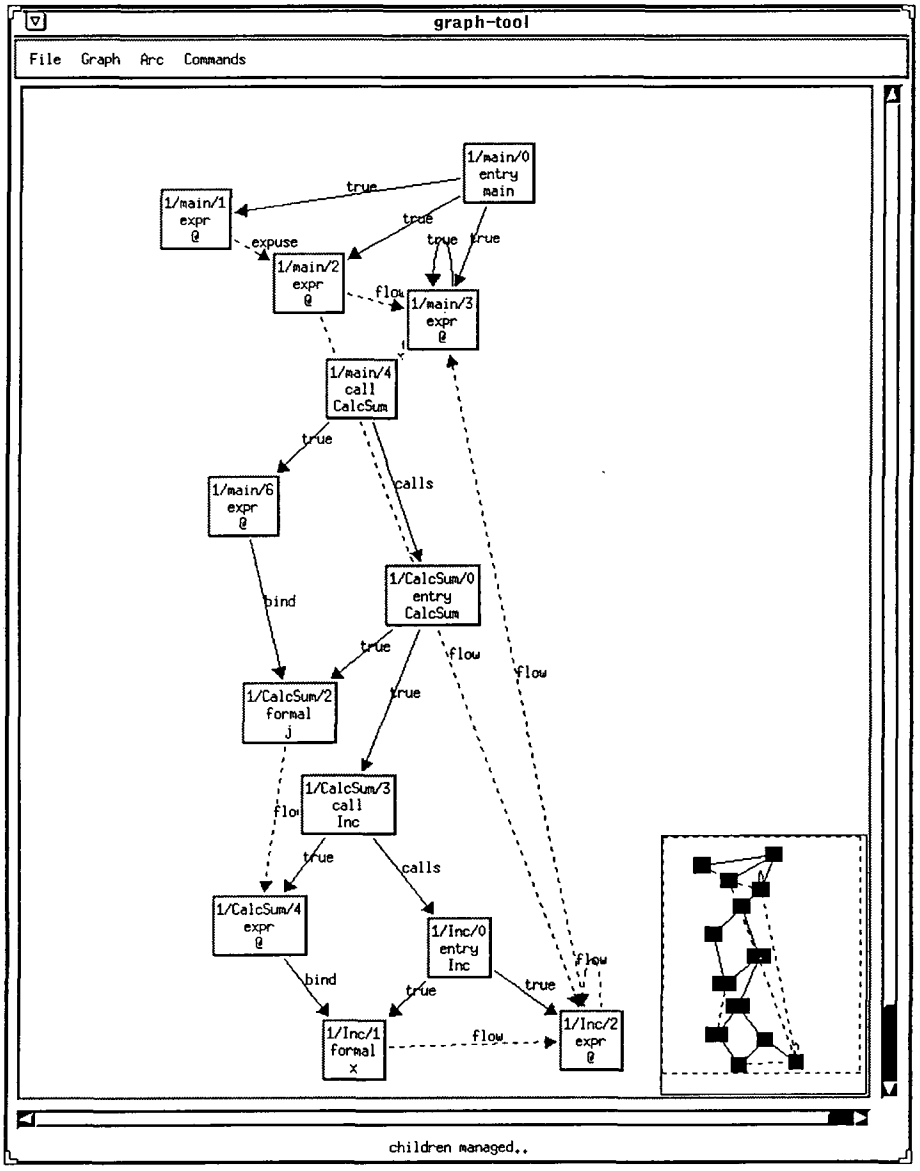


Figure 6.3: Program slice on formal parameter x of Inc function.

```
void main()
  int sum;
  int i;
{
  i = sum = 0;
  while (1 < 20) {
    CalcSum(&i);
  }
}
```

```
int CalcSum(int *j)
{
  Inc(j);
}
```

```
void Inc(int *x)
{
  *x = *x + 1;
}
```

Table 6.6: Program slice on formal parameter x of Inc function.

A slice on `ptr_to_list_item = head` of function `print_list_item` in **linked_list** produces the result shown in table 6.7 and figure 6.4.

A ripple analysis view can be created using the **ripple** command. Like the slice command, the maintainer must specify a CCG vertex. Again the ripple output is translated manually to show the actual source code affected.

The following query constructs a ripple analysis for the `s = s + *j` statement in function `CalcSum` of the **sum** program.

```
?- ripple(1,'CalcSum',7).

Ripple (forward slice) :-

  node(1, CalcSum, 6)
  node(1, CalcSum, 1)
  node(1, main, 5)
  node(1, main, 10)
  node(1, main, 8)
  node(1, CalcSum, 8)
  node(1, CalcSum, 7)

7 nodes in the forward slice
```

The impacted statements are shown in table 6.8 and figure 6.5.

```
main()
{
  tail = NULL;
  tail = add_list_item(tail);
  tail = add_list_item(tail);
  tail = add_list_item(tail);
  print_list_items();
}
```

```
void *print_list_items()
{
  list_item *ptr_to_list_item;

  ptr_to_list_item = head;
}
```

```
list_item *add_list_item(list_item *entry)
{
  list_item *new_list_item;

  new_list_item = (list_item*) malloc(sizeof(list_item));

  if (entry == NULL) {
    head = new_list_item
  }
  return new_list_item;
}
```

Table 6.7: Program slice on `ptr_to_list_item = head` of `print_list_item` function.

```
void main()
{
  int sum;

  sum = CalcSum(sum);
  sum = sum;
}
```

```
int CalcSum(int s)
{
  if (s < 100) {
     s = s + *j;
  {
  return s;
}
```

Table 6.8: Ripple analysis on `s = s + *j` of `CalcSum` function.

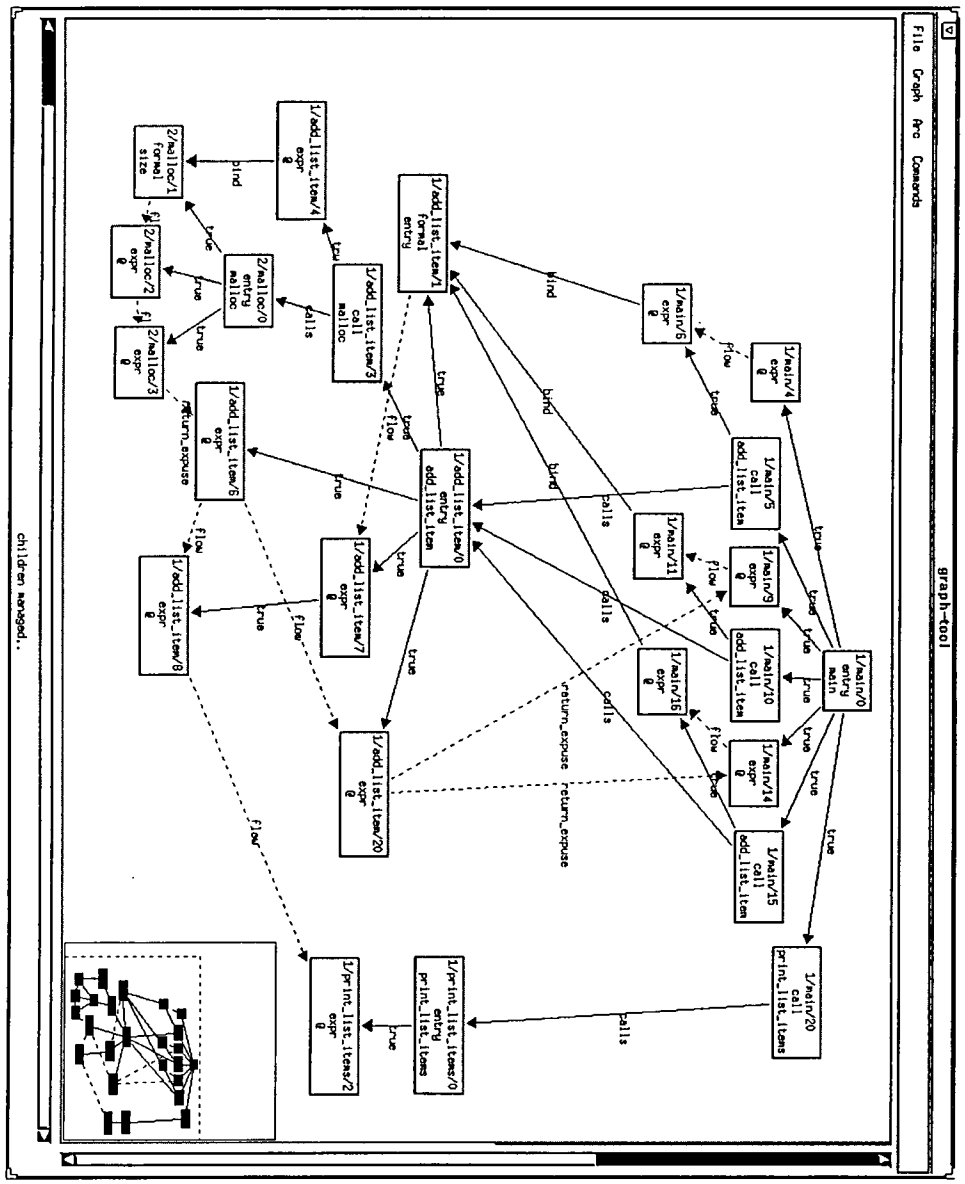


Figure 6.4: Program slice on `ptr_to_list_item = head` of `print_list_item` function.

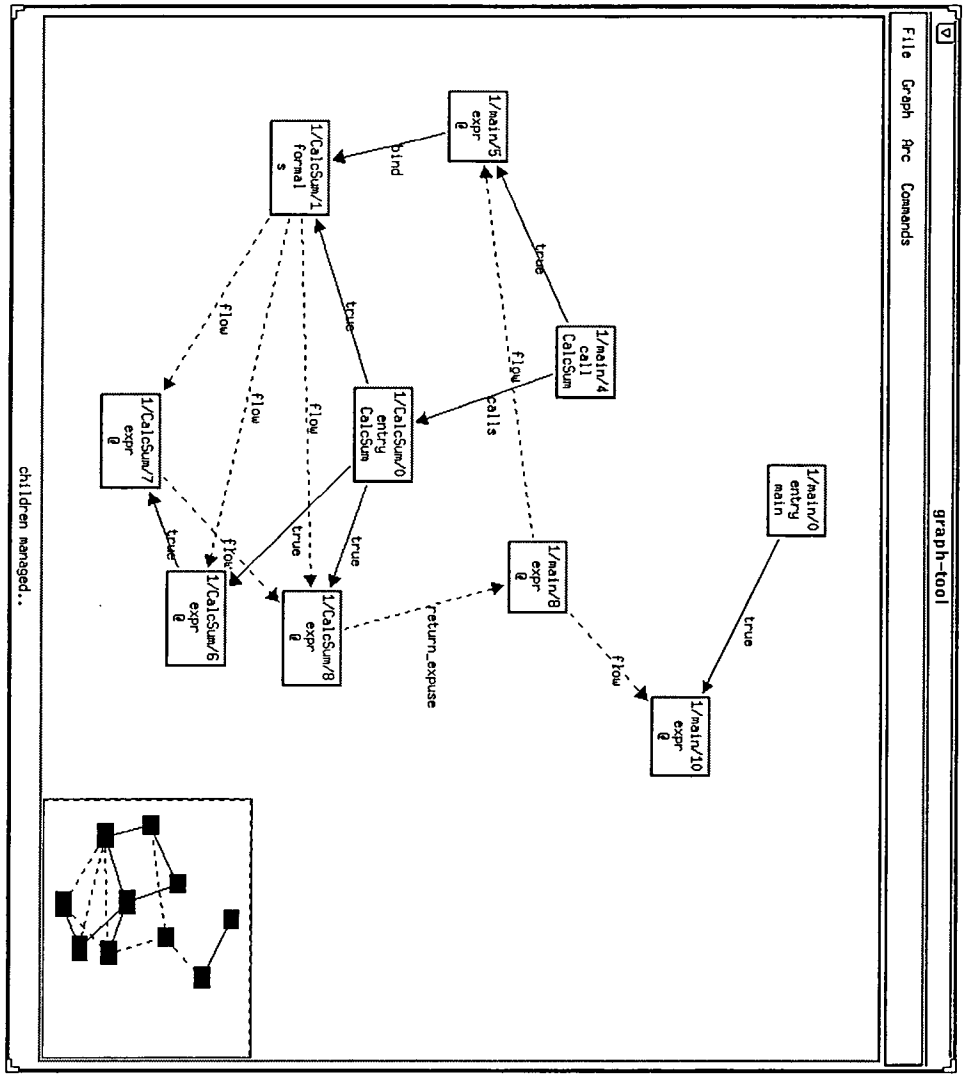


Figure 6.5: Ripple analysis on `s = s + *j` of `CalcSum`.

6.4.5 Program Components

Although not an integral part of the CCG representation, the prototype tool allows a maintainer to construct views of the functions and components comprising the subject system.

The user can list the files making up the subject system using the list command. The following example lists all C files of the subject system, in this case **sum**.

```
?- list(_).

List of Files:

  file(1, ../examples/sum.c)
```

Information on the component definitions within the subject program is constructed using the find_def command. The types and scopes of functions and variables are listed, as shown for the program **sum** below.

```
?- find_def(_).

  CalcSum : fun ret @int (fun: @external; Block: []; file: ../examples/sum.c)
  Inc : fun ret @void (fun: @external; Block: []; file: ../examples/sum.c)
  main : fun ret @void (fun: @external; Block: []; file: ../examples/sum.c)
  j : ptr to int (fun: CalcSum; Block: []; file: ../examples/sum.c)
  s : int (fun: CalcSum; Block: []; file: ../examples/sum.c)
  x : ptr to int (fun: Inc; Block: []; file: ../examples/sum.c)
  i : int (fun: main; Block: [1]; file: ../examples/sum.c)
  sum : int (fun: main; Block: [1]; file: ../examples/sum.c)
```

Information on a specific component can be gathered by using a more precise query. The following example will list definitions of the variable allocp.

```
?- find_def(allocp).

  allocp : ptr to char (fun: @external; Block: []; (static);
           file: ../examples/lines.c)
```

Type information is constructed using the list(types) query.

```
?- list(types).

Type Definitions in: ../examples/linked_list.c
  list_item : [struct,_list_item]
```

The list of functions contained in the subject system is created with the list(functions) command. The functions of the **linked_list** program are shown below.

```
?- list(functions).

List of Functions:

  fun(1, add_list_item)
  fun(1, main)
  fun(1, print_list_items)
  fun(2, malloc)
  fun(2, printf)
```

Information on the subject program's external variables may be similarly constructed using the list(globals) command.

```
?- list(globals).

Globals in: ../examples/linked_list.c
  head : ptr to list_item ( file: ../examples/linked_list.c)
  tail : ptr to list_item ( file: ../examples/linked_list.c)
```

References to external variables may be listed using the global_call query. The following command lists all functions of **lines** which reference the allocp external variable.

```
?- global_call(_,_,allocp).

Global Calls (gc(FileNr, Fun, Global):

  gc(1, alloc, allocp)
  gc(1, main, allocp)
```

The types and names of the formal parameters of a subject program can be found with the formals command. The user can provide a more general or specific query to generate information on all or only given functions. The following command lists the names, types and scopes of all formal parameters of the **sum** program.

```
?- formals(_,_).

Formal parameters :
```

```
s : int (fun: CalcSum; Block: []; file: ../examples/sum.c)
j : ptr to int (fun: CalcSum; Block: []; file: ../examples/sum.c)
x : ptr to int (fun: Inc; Block: []; file: ../examples/sum.c)
```

A maintainer can view parameter binding information in two ways. A formal parameter vertex can be specified using the list_bind command or a formal parameter name using the list_bind_name command. In either case a list is produced of CCG binding edges incident on the given formal parameter.

The following examples demonstrate binding information for the parameter x at vertex 1 of Inc within **sum**. The results of each query show that vertex 4 of CalcSum is an actual parameter bound to x.

```
?- list_bind(_,_,_,1,'Inc',_).

Binding edges :-

  bind(1, CalcSum, 4, 1, Inc, 1)


?- list_bind_name(1, 'Inc', x).

Binding edges :-

  bind(1, CalcSum, 4, 1, Inc, 1)
```

6.5 Software Maintenance Scenarios

Whilst the CCG has been shown to provide a software maintainer with many different views of a subject system, it is important that this information can be used to help the maintainer perform maintenance tasks. A maintainer may wish to gain an understanding of the code to correct an error in the software - corrective maintenance. Another task could be to modify the software in some way to improve its functionality - perfective maintenance.

This section outlines two scenarios showing the use of the CCG system in each of these activities. The examples indicate the aim of the maintenance task and describe the views which may be constructed by the maintainer to help achieve the given task.

In both case the **linked_list** program is used as an example. The **linked_list** is a small program but contains a number of 'difficult' features, for example recursive structures, dynamic allocation, external variables and pointer parameters.

6.5.1 Corrective Maintenance

Many maintenance tasks involve correcting problems with the software or 'bug fixing'. For example, initialising `ptr_to_list_item` to `tail` rather than `head` in the function `print_list_items` gives the following.

```
void print_list_items(void)
{
  list_item *ptr_to_list_item;

  for (ptr_to_list_item = tail; ptr_to_list_item != NULL;
       ptr_to_list_item = ptr_to_list_item->next) {
     printf("Value is %d \n", ptr_to_list_item->val);
  }
}
```

This function now outputs only the final item of the linked list structure rather than each item. The following scenario describes how a maintainer may detect this problem.

A maintainer may first wish to determine the function in which the problem is occurring. The list(functions) command will list each function of the program.

```
?- list(functions).

List of Functions:

  fun(1, add_list_item)
  fun(1, main)
  fun(1, print_list_items)
  fun(2, malloc)
  fun(2, printf)
```

Producing a call graph view will indicate how these functions are used. The fun_call command outputs the program's call graph.

```
?- fun_call(_,_).

Function Call Graph

  fc(add_list_item, malloc)
```

```
fc(add_list_item, printf)
fc(main, print_list_items)
fc(main, add_list_item)
fc(print_list_items, printf)
```

Given that the problem surfaces as the linked list is printed, the maintainer may conjecture two possible sources.

1. There is a problem within `print_list_items`. The list is not printed correctly.

2. There is a problem within `add_list_items`. Items are not added to the list correctly.

Investigating the former, the maintainer will find the print statement:

```
printf("Value is %d \n", ptr_to_list_item->val);
```

A useful technique when searching for the cause of an error is to construct a program slice on the statement at which the error is observed. A slice on the `ptr_to_list_item->val` actual parameter of the `printf` call:

```
?- slice(1, print_list_items, 6)
```

will produce a reduced program containing any statement potentially affecting this variable. This slice should contain the statement causing the problem. The resulting slice is shown in table 6.9. This view may be useful in locating the source of the error but the maintainer may still require further information. A definition-use view on `ptr_to_list_item->val` created using the def-use command produces the output shown below.

```
?- def_use(_,_,_,1,print_list_items,6).

Def-use pairs :-

  flow(1, print_list_items, 2, 1, print_list_items, 6)
  flow(1, add_list_item, 18, 1, print_list_items, 6)
  flow(1, print_list_items, 8, 1, print_list_items, 6)
```

This represents the definitions:

```
ptr_to_list_item = tail;
tail = add_list_item(tail, 2);
ptr_to_list_item = ptr_to_list_item->next;
```

```
main()
{
  add_list_item(5);
  add_list_item(7);
  tail = add_list_item(2);

  print_list_items();
}


void *print_list_items()
{
  list_item *ptr_to_list_item;

  for (ptr_to_list_item = tail; ptr_to_list_item != NULL
       ptr_to_list_item = ptr_to_list_item->next) {
      printf(ptr_to_list_item->val);
  }
}


list_item *add_list_item(int value)
{
  list_item *new_list_item;

  new_list_item = (list_item*) malloc(sizeof(list_item));
  new_list_item->val = value;
  new_list_item->next = NULL;
  return new_list_item;
}
```

Table 6.9: Program slice on `ptr_to_list_item->val` of modified `print_list_items` function.

Inspecting the program slice and definition-use view, the maintainer should observe that the temporary pointer ptr_to_list_item is initialised to tail, the end of the list and that as a result the list is never traversed.

6.5.2 Perfective Maintenance

The most common category of software maintenance is perfective maintenance where changes are made to improve the functionality of the software. The following example describes a maintenance scenario in which the linked list of items is modified to become a list of character pointers rather than a list of integers.

The initial site for this modification will be the type definition of list_item. The val field must be altered to char* to reflect the new list contents. The maintainer can use the CCG system to determine the possible effects of the change and other code that may require modifications.

The commands list(globals), global_call and find_def can be used to find functions or variables which may also require modification. External variables are found using the list(globals) query.

```
?- list(globals).

Globals in: ../examples/linked_list.c
  head : ptr to list_item ( file: ../examples/linked_list.c)
  tail : ptr to list_item ( file: ../examples/linked_list.c)
```

Two external variables head and tail are listed, each of type ptr_to_list_item. Functions accessing these globals are found using global_call.

```
?- global_call(_,_,_).

Global Calls (gc(FileNr, Fun, Global):

  gc(1, add_list_item, head)
  gc(1, main, head)
  gc(1, main, tail)
  gc(1, print_list_items, head)
```

Further information on the functions accessing head and tail is found using the find_def query.

```
?- find_def(add_list_item).
```

```
  add_list_item : fun ret ptr to list_item (fun: @external; Block: [];
                  file: ../examples/linked_list.c)

?- find_def(print_list_items).
  print_list_items : fun ret @void (fun: @external; Block: [];
                     file: ../examples/linked_list.c)
```

The external variables head and tail are accessed in the functions main, add_list_item and print_list_item which may require modifications. The function add_list_item also returns a pointer to list_item and is therefore a strong candidate for further inspection.

The formal parameters of add_list_item can be inspected using the formals command.

```
?- formals(1, add_list_item).

Formal parameters :
  entry : ptr to list_item (fun: add_list_item; Block: [];
          file: ../examples/linked_list.c)
  value : int (fun: add_list_item; Block: [];
          file: ../examples/linked_list.c)
```

The parameter value is of interest since this represents the modified field of the linked list structure. The parameter should be changed to type char*.

A ripple analysis on this parameter will show the potential effects of this modification. The ripple analysis query:

```
?- ripple(1, add_list_item, 2)
```

produces the following output:

```
  list_item *add_list_item(int value)
  {
    printf(value, entry->val);
    new_list_item->val = value;
  }


  void print_list_items()
  {
    printf(ptr_to_list_item->val);
  }
```

Each of these statements will require modification to reflect the changed type of the val field.

A program slice on the formal parameter value:

```
?- slice(1, add_list_item, 2)
```

may provide the maintainer with further areas of the code to investigate. The resulting slice is:

```
main()
{
  add_list_item(5);
  add_list_item(7);
  add_list_item(2);
}

list_item *add_list_item(int value)
{
}
```

The statements in this slice are each calls to `add_list_item`. The actual parameters of these calls must be altered to the new type.

6.6 Summary

This chapter has described the results achieved using the prototype CCG system. Firstly, a group of four small C programs of up to 121 lines of code were analysed. Correct CCG representations were constructed for each of these subject programs. Secondly, two larger programs of up to one thousand lines of code were analysed. CCG vertices, control dependencies and interprocedural control information were each computed successfully for these two larger programs. However, the implemented data dependence analysis algorithm was found to be inadequate for programs of this size.

Empirical results of the time taken in construction and space requirements of each of the CCGs were outlined. Most encouraging was the time spent during the initial CCG translation step, producing the 'partial' CCG. Less encouraging was the inefficiencies of the control and data dependence analysis algorithms. Space analysis of the CCGs showed that on average the ratio of CCG facts to lines of code was approximately three or four to one. Space requirements for the CCG approached an order of magnitude greater than the subject C code.

The views and information made available to a maintainer were demonstrated. These views include call graphs, control dependencies, definition-use information, program slices and ripple analyses.

Finally scenarios describing the use of the CCG system in two maintenance tasks were outlined.

The examples showed how a maintainer can use the CCG system to produce program views to aid program comprehension and help with maintenance activities.

Chapter 7

Evaluation of the Combined C Graph

This chapter presents an evaluation of the Combined C Graph (CCG). The representation is first evaluated in terms of the C language coverage provided both by the theoretical CCG and by the prototype implementation. The program views made available to the maintainer are then discussed. The algorithms used to build the CCG are evaluated in terms of the success in constructing the representation and their theoretical complexity. Finally the space requirements of the CCG are analysed.

7.1 C Language Coverage

This section presents an evaluation of the C language coverage provided by the CCG program representation. The theoretical CCG defined in chapter 4 is first discussed and this is followed by an analysis of the coverage given by the current prototype system.

7.1.1 Theoretical CCG Representation

This section assesses the C features that may be represented by the CCG. The features provided by the language are divided into two categories. The first of these categories of features are those represented by the CCG vertices and analysed to determine their data and control flow effects, which if any are modelled by the CCG's edges. Other C features are described which are permitted in a subject program but do not create any dependencies and consequently have no effect within the CCG. The second category of C features are those which cannot be represented by the vertices of the CCG and have data and control flow effects which cannot be modelled by the CCG.

Permitted C features

Assignment expressions are a common feature of most C programs. An assignment expression without embedded side effects is represented simply by a CCG vertex. More complex assignment expressions with embedded side effects are analysed and represented by refined CCG vertices connected by expression-use or lvalue-definition edges. This gives a more intuitive program representation and improved program slicing, discussed later in section 7.2. The data flow effects of assignment expressions are modelled successfully by determining the variables defined and used within the expression to construct flow dependencies.

The operators involved within the sub-expressions of an assignment expression are modelled by the CCG but the specific effects of the operators are not distinguished. For example, *arithmetic operators*, *increment and decrement operators*, *relational operators* and *bitwise operators* are all analysed in the same way; the variables and referenced objects operated on are considered to be used within the expression. This approach is adequate for a dependence-based representation since it is the actual use or definition of a variable within an expression that is important and not the actual values involved. For the same reason, the values of any constants in the code, or the specific types of any ordinal variables involved in an expression are not important.

Operators with control flow effects such as *logical operators* or *conditional operators* are modelled by producing refined CCG vertices. The control effects of the operators are represented by constructing control dependencies between the refined vertices. Again the advantage of a more intuitive program representation results.

The CCG is able to represent each of the *control structures* provided by the C language. The control effects of `if..then..else`, `while`, `for`, `do..while`, `break`, `continue` and `goto` are each modelled by the control dependence edges of the CCG. The `switch` statement is modelled accurately using both control dependencies and the new switch dependence edge.

The CCG represents the individual *functions* of a subject C program by creating CCG subgraphs, FCCGs, for each function. Call and entry vertices connected by call edges successfully model the call relationships of the C programs. The *value parameters* of the C language are represented by formal and actual parameter vertices, and parameter binding effects modelled by the CCG's binding edges. Function calls of the form `f(g())` where the actual parameter is not explicitly known but is a value returned from another function call are represented by introducing 'dummy' CCG vertices. This new vertex provides consistency in the CCG representation by giving a common 'shape' for the function call interface.

The use of *pointer parameters* is allowed by the CCG but the objects referenced by the pointer do not appear as vertices in the call interface. The actual pointer parameter and corresponding formal pointer parameter, like other value parameters, are represented by actual and formal vertices. The data flow effects of the referenced objects within the called function are represented by explicit interprocedural data dependencies. This means that the CCG does not present an encapsulated interface between functions, but instead contains interprocedural edges connecting arbitrary CCG vertices. The CCG representation is more explicit in its characterisation of data dependence and removes the need for interprocedural propagation algorithms such as those used by the IFG/UIG representations to calculate the interprocedural data dependencies. The disadvantage of this approach is an absence of calling context information and consequently less accurate program slices. This issue is discussed in detail in section 7.2.

The approach taken is selected because of the difficulties of representing the referenced objects adequately as part of the call interface. A referenced object such as a recursive data structure may be unbounded in size or may vary on different paths to the call site or at different calls of the same function. Representing the referenced object by explicit actual and formal parameter vertices would therefore require an approximation of the structures that could be referenced by the actual parameter. The task of selecting an approximation providing a conservative and useful solution is a difficult one.

The *return value* of a function is represented by the new return-expression-use edge. Again the specific value returned is not required by a dependence based representation but the relationship between the return vertex and the vertex referencing the returned value is important.

The pointer variables, structure variables and recursive structures of the C language are each represented in the CCG by decomposing them into their component objects and analysing these objects. The data flow effects of a pointer variable are represented in terms of the pointer and any referenced objects. The data flow effects of a structure variable are represented in terms of the component fields. This solution improves the accuracy of the data dependence information contained in the CCG and consequently presented to the maintainer.

The representation of array variables is less accurate. Array variables are treated as 'aggregates', or a single structure, and unlike pointers or structure variables are not decomposed further into the individual component elements. This approach is a common one in the field of static analysis because of the difficulty of evaluating subscript values given by a variable. For example, the specific array element referred to by the expression `a[i]` cannot be determined from static analysis of the source code.

The rationalisation for the use of 'aggregates' is that individual elements of an array are often treated in the same way in a program and consequently generate the same data dependence effects. For example a loop structure may be used to initialise each element of the array, using a counter to access subsequent array elements on successive iterations. A second loop structure may then print out each individual element. Each element is defined at the same CCG vertex and used at the same CCG vertex and will produce the same data dependencies. Analysis of pointer arithmetic is also simplified as an increment or decrement of an array pointer maintains a pointer to the same 'aggregate' structure. However, the data dependence information contained in a CCG representation of programs involving arrays will inevitably become less precise as array usage increases. This is particularly important for more complex data structures involving arrays where the accuracy of analysis of the entire structure can be limited by the imprecise array analysis as information on the shape of the structure is lost.

Standard library routines are represented in the CCG using 'stub' routines representing only the interprocedural control and data flow effects. Since the standard library is provided by any environment supporting C, only these interprocedural effects should be of interest to a maintainer. The approach provides a complete CCG representation for a given subject program and enables a maintainer to construct complete program slices and views of the program call graph.

The CCG representation does not contain any explicit information on the component declarations of the subject C program or the scoping rules of the variables involved. The block structure of the C language is represented implicitly in the CCG only through the data flow effects of the local and external variables. Like pointer parameters, *external variables* may give rise to interprocedural data dependencies. *Static variables* are similarly distinguished from local variables only through their data flow effects.

The CCG could be enhanced to contain 'declaration vertices' describing the external, static and local variable declarations, type definitions and *typedefs*. Scoping rules may be represented by locating these vertices at the entry point of the relevant C block. Thus a local declaration could become control dependent on the entry vertex of an FCCG or on the expression controlling the entry to a new compound statement. For example, the declaration of `i` after the `if` expression:

```
if (n > 0) {
  int i;

}
```

would produce a declaration vertex control dependent on the expression `n > 0`.

No mechanism currently exists to represent initialisation of variables as they are defined. The introduction of declaration vertices would also allow for the representation of these initialisations.

Other C language features which may occur in a subject C program but which do not create dependencies in the CCG are *register variables*, *type conversions* and the *sizeof* operator. The `register` declaration informs the C compiler to place the specified variables in machine registers. This request can be ignored and is not an issue for a program comprehension environment. The `sizeof` operator computes the size of an object but has no data or control flow effects. Type conversions or *casts* convert a value of one type to a value of another type. Since specific expression values are not used in the CCG, the type conversion of a value has no effect and is not part of the representation.

Unrepresented C features

The most important feature of the C language not represented by the CCG is *pointers to functions.* It is possible within C to define pointers to functions which can then be used in the same way as other variables. For example, pointers to functions may be assigned to, placed in arrays, passed to functions as parameters or returned by functions.

The use of pointers to functions as parameters is equivalent to the use of procedure parameters in other programming languages. Procedure parameters complicate the construction of the program call graph since a reference to a formal parameter may represent invocations of distinct procedures. Ryder[80] reports an algorithm for the construction of a call graph for programs with procedure parameters, which could be applied to the pointers to functions of C. Using this algorithm it may be possible to approximate the set of functions that could be invoked at any call site which is a dereference of a pointer to a function.

Union variables are single variables which may hold at different times objects of different sizes and types. The C compiler provides enough space for the 'largest' object. For example, given the following union declaration:

```
union u_tag {
  int ival;
  float fval;
  char *cval;
} u;
```

the variable u may hold either an integer value, a float value or a character pointer.

The problem presented by a union variable is in representing the data flow effects of the variable. The approach of analysing the components used in structure and pointer analysis cannot be applied since the components of the union variable effectively change as the type of the stored value changes. A conservative approach would be to treat the entire union variable as an 'aggregate' and not to attempt any more detailed analysis.

The final component of the C language which is not represented by the CCG is the C preprocessor. The preprocessor provides features such as conditional compilation, macro substitution and file inclusion. The C preprocessor is difficult to analyse for a number of reasons. Conditional compilation produces multiple versions of the same program. A program representation would ideally model each of the possible versions of the code. Macro substitution presents two main problems. The first of these is that a macro definition is not necessarily a complete syntactic unit. Consequently the code surrounding the macro use will not be syntactically legal C, making analysis a difficult task. The second problem is that a maintainer would benefit from information on both the unexpanded macro form and the expanded code form. An IPR must therefore represent each form. This problem is related to that of file inclusion, particularly standard library headers. In addition to function declarations, these files include many macro definitions. Substitution of these macros can complicate the source code, for example the easily understood `stdin` is replaced by the less intuitive `(&_iob[0])`.

Summary of Theoretical C Coverage

The CCG covers a large part of the C language. In addition to the language features modelled by representations such as the UIG, the CCG permits expressions with embedded side effects, embedded control flows, value returning functions, value parameters, pointer variables, pointer parameters, structure variables, recursive structures, C control structures and standard library routines. Approximate information is contained to represent the use of array variables. Component declaration and scoping information is not specifically represented in the CCG but could be introduced by the addition of declaration vertices. C features which cannot currently be represented by the CCG are pointers to functions, union variables and the C preprocessor.

7.1.2 Prototype CCG Representation

This section discusses the C language coverage provided by the prototype system described in chapter 5. The prototype system constructs an accurate CCG for subject C programs involving the language features permitted by the theoretical CCG representation with only two further re-

strictions. Each restriction is a limitation of the prototype coding, rather than a theoretically more difficult problem.

- Where an array variable such as `a[i]` is represented using the pointer notation `*(a + i)`, it is assumed that the left hand side operand of the + operator represents the pointer variable, and that the right hand side operand represents the array offset. Therefore an expression written as `*(i + a)` would be misinterpreted.

- Side effects are not permitted within a 'pointer notation' array reference. For example, an expression such as `*(a++ + i)` which includes a side effect whereby the pointer variable a is incremented to point to the next array element, is not permitted.

Each of these features causes problems during the C translation stage *ccg_trans*.

Ccg_trans is able to analyse a larger class of C programs than may be modelled by the CCG representation. The analyser covers the complete C language but does not provide information on the following features.

- Constant values are represented only with a generic *constant* or *string* value.

- Operators, with the exception of side effect or control flow inducing operators, are represented with a generic *operator*.

- Case labels are represented with a generic *case-label*.

- Initialisation of variables within declarations is ignored.

- Type casts are not analysed.

However, with the exception of initialisation of variables, these features do not create any control or data flow effects and more detailed information is not required to construct the CCG. Initialisation of variables must be achieved by modifying the source code to contain explicit initialisation statements.

The prototype translator could be used as the basis for an enhanced CCG with declaration vertices, since the information required on both component types and scopes is already generated by the system. Union variables and pointers to functions which must be added to produce a complete C representation are also analysed by the translator.

The remaining phases of the prototype CCG system each adequately deal with the language features permitted by the theoretical CCG representation. The interprocedural control analysis

routines *build_ccg* and control dependence analysis routines *control_dep* can each be applied to any C code not involving the use of pointers to functions.

To summarise, the prototype CCG system is able to construct the CCG with the exception of only two restrictions, each concerning pointer/array relationships. The current CCG translator in fact offers a more complete language coverage than the CCG itself and may be of use in the construction of an enhanced CCG.

7.2 Program Views

This section discusses the achievements of the CCG in providing a variety of program views to a maintainer. This issue is described in two sections, firstly the information made available by the theoretical CCG representation and secondly the views provided by the prototype system.

7.2.1 Theoretical CCG Views

The CCG representation makes a number of programming level views available to a maintainer. The majority of the views are integrated directly into the CCG and the construction algorithms necessary to make the view available are trivial. This is the case for *call graph*, *control dependence* and *definition-use* views. In each case the view is constructed simply by extracting a CCG subgraph comprising the appropriate graph edges. The presence of explicit interprocedural flow edges in the CCG trivialises the construction of the definition-use information. Other IPRs require a more complex propagation algorithm to create this view. The CCG presents a more useful control dependence view due to the refined analysis of expressions with embedded side effects.

Flow sensitive data flow views created during data dependence analysis are provided by the CCG and are represented as graph annotations attached to each function. These views provide useful information and are simple to provide to a maintainer. However the information stored in the representation is somewhat at odds with the other components of the CCG. The annotations are in effect 'tagged on' to the CCG rather than an integral part of the representation.

The CCG can be used as a basis for the construction of program slices which are computed using a simple backwards traversal of the edges of the representation, starting from the selected CCG vertex. A slice is therefore constructed in linear time.

The refined vertices of the CCG were found in certain cases to produce more accurate program slices. Refined vertices ensure that only a single program object is defined at any single CCG vertex and that any flow dependencies incident on that vertex involve variables contributing to the

defined object. Traversing any flow dependence does not introduce any spurious vertices into the slice. Expression-use, lvalue-definition and return-expression-use edges ensure that a slice reaching a refined vertex also includes vertices contributing values to the expression at the refined vertex.

However, a limitation of this approach is that a slice can only be performed on a variable defined at a given vertex, or on each variable defined or used at the vertex. For example, given the statement:

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

a slice on variable a will include any vertices which contribute to the values of x, y or z. This slice is correct since each of x, y and z do contribute to a. However a slice on variable z is not possible since the result will include vertices also contributing to a, x and y, none of which contribute to the value z. This problem could be overcome by annotating each flow dependence with a label indicating the program object creating the dependence. A more sophisticated traversal algorithm must then use these annotations to determine whether to traverse a given flow dependence.

The accuracy of program slices constructed using the CCG is also limited by the lack of calling context information in the graph and the presence of interprocedural data dependencies and return-expression-use edges. The need for calling context information during program slicing was recognised by Weiser[89]. A slice descending into a function must return to the calling function from which the function was entered, and not to any other calling function. If this does not occur, an inaccurate overly-conservative slice may result.

Horwitz et al[45] developed a two-phase traversal algorithm to overcome this problem, based on the SDG representation. The SDG restricts interprocedural edges to the call interface, therefore a graph traversal can only pass from one procedure to another via the edges of these encapsulated interprocedural edges. The SDG also contains the *interprocedural transitive flow dependence* representing transitive dependencies between a procedure's reference parameters across a call site. The two-phase graph traversal algorithm makes use of these new edges to 'step across' a call site rather than descend into the called function. Each phase traverses only a subset of the SDG's edges to prevent 'ascending' or 'descending' call sites respectively.

The presence of explicit data dependence edges and return-expression-use edges in the CCG prevents the application of a similar traversal algorithm to counter the lack of calling context information. A program slice may proceed throughout the CCG representation via these interprocedural edges and cannot be restricted by an encapsulated interprocedural interface. The absence of calling context information in the CCG will in some cases lead to excessively pessimistic program slices.

A program slice can propagate along any interprocedural edges from a given function, returning to out of context call sites along these edges.

A simple solution, in the absence of recursive functions, is to introduce separate copies of each FCCG for each individual call site. In this way calling context problems are eliminated completely but the space requirements of multiple FCCG representations are prohibitive. A more acceptable solution would be to eliminate the unconstrained interprocedural edges to give an encapsulated interprocedural interface similar to the SDG. A two-phase traversal algorithm based on that described by Horwitz et al could then be employed. As discussed in section 7.1.1, this solution would require a new representation for pointer parameters to explicitly model the referenced objects by formal and actual parameter vertices.

The CCG also makes available a ripple analysis view. A ripple analysis view is constructed at a specified CCG vertex, traversing the CCG along all edges in the forward direction from that vertex. Like the program slicing view discussed above, the accuracy of the ripple analysis is enhanced by the refined CCG vertices and expression-use, lvalue-definition and return-expression-use edges but inaccuracies will also be introduced by the absence of calling context information.

The CCG provides a wide range of programming level views covering the control dependence, data dependence and interprocedural effects of the subject C program. Program slicing and ripple analysis provide further information for program comprehension and software maintenance. However, other program views not presented by the CCG would further help program comprehension.

A software maintainer can gather useful information from cross reference information. For example, a maintainer may wish to view the type declarations or variable declarations of the subject program and then determine all future references to these components. In its present form the CCG does not allow the maintainer to create this view. Adding cross reference information to the representation in the form of declaration vertices proposed earlier would help a maintainer understand in particular the use of externally defined components which may be referenced throughout the program. Declaration edges could be added to the CCG to relate the declaration vertices to other CCG vertices referencing the defined component. In this way a cross reference view is integrated into the CCG representation.

In addition to the existing control dependence view, a maintainer may also benefit from a control flow view. Control flow information is required to compute the control dependencies of the CCG and hence is already available. Control flow edges may be introduced into the CCG, connecting each vertex with its control flow successors. A control flow view is then trivial to construct. It is also possible to compute control flow information from the existing control dependencies, thereby

reducing the space requirements of the CCG. However the algorithms to perform this task are non-trivial.

Another program view which could be made available is code quality or metrics information. Information may be derived from the CCG on problems such as unused variables or missing parameters. Unclear C such as order of evaluation dependent code may also be detected. Metrics could be derived using the representation's interprocedural edges to indicate the level of coupling between functions or to describe the complexity of the subject program's control flow.

7.2.2 Prototype System Views

The prototype CCG implementation provides many of the views made available by the theoretical CCG. By constructing Prolog meta programs a maintainer is able to create call graph views, control dependence views, definition-use views, program slices and ripple analyses. These views are each constructed quickly, with response times within a few seconds. The prototype system is unable to provide flow-sensitive data flow analysis views since this information is not collected by the data dependence analysis step.

The CCG is also able to provide information on the components and types of the subject C program. This information is not part of the CCG representation but is gathered during the initial translation step *ccg_trans*. Control flow views also not part of the CCG may additionally be constructed.

The major drawback of the prototype system in its present form is that the views presented to a maintainer are given in terms of the vertices and edges of the CCG and not the original C source code. Whilst for a small program it is simple to interpret the output and to relate the information to the source statements, as program size increases this quickly becomes impossible. A usable CCG system must present the maintainer with information relating to the actual C statements, not the intermediate CCG vertices.

A second problem with the current views is that information constructed using the Prolog meta-programs is presented textually. Graphical representations may only be produced for the complete CCG representation, program slices and ripple analysis views by translating the Prolog output and loading the resulting file into a graphical display tool. The user is able to position and move the vertices and edges of the displayed graph but is unable to generate further views.

Whilst for information on program components a textual interface is sufficient, views such as program slices or definition-use pairs become more useful if presented in a graphical form. An improved front end is therefore required to improve the presentation of the information to a

maintainer. Views may be presented graphically with links to the source code. A maintainer must be able to create views either textually or by clicking on the program component or statement which is of interest. The use of colour may also help comprehension, particularly in the display of program slices or ripple analyses where the results can be highlighted both graphically and within the source text.

7.3 CCG Construction Algorithms

This section evaluates the algorithms used in the construction of the CCG. The theoretical complexity of each algorithm is discussed. The algorithms are also appraised in terms of the results achieved by and the time requirements of the implemented prototype system. Each phase of the CCG construction is addressed in turn.

7.3.1 Partial FCCG Construction

This section discusses the algorithms employed in the construction of the partial CCG. This involves an initial CCG translation step followed by control dependence analysis of each function.

CCG Translation

The first step in the construction of the CCG is to parse the source code and to construct an abstract syntax tree representation. The parse of the source code and construction of the of the syntax tree are each well known algorithms used in compiler construction and are therefore not addressed in detail in this thesis.

The resulting abstract syntax tree is then traversed and CCG vertices created, taking into account any embedded side effects or control flows. Expression-use edges and lvalue-definition edges are created together with a control flow graph for each C function.

The theoretical complexity of the traversal of the abstract syntax tree for a single function is determined as follows. For a graph with E edges and V vertices, where $V < E$, the depth first traversal is achieved in $O(E)$. This result is given by Aho et al[2]. For a function with N statements, the expression subgraph of the abstract syntax tree for each statement must be traversed. Therefore the traversal of the complete abstract syntax tree for a single function is achieved in $O(N.E)$.

The prototype *ccg_trans* program successfully implements the CCG translation algorithm. The *lex* and *yacc* derived parser constructs the abstract syntax tree representation. C routines then perform a depth first traversal of this tree, creating refined CCG vertices, expression-use edges,

lvalue-definition edges and a Prolog representation of each function's control flow graph.

The time requirements of the translation step were found to be encouraging. The prototype implementation running on a SUN 670 analysed on average 12kb of C source code per second and processed programs up to 1000 lines of code in only 2.6 seconds of system time. The algorithms are simple to implement and the time requirements suggest that the current translation process is suitable for the analysis of large programs.

Control Dependence Analysis

The control dependence edges of each FCCG are constructed using an algorithm described in section 4.5.1. The algorithm modifies that of Ferrante et al[25], using a simpler method to construct the flow graph's post-dominator tree. The algorithm employed allows the construction of control dependencies from an arbitrary control flow graph. Programs involving loops or unstructured control flow may be analysed. Consequently, control dependencies for C programs using any of the language's control structures, including `break`, `continue` or `goto` may be constructed. The proof of correctness of the algorithm is given by Ferrante at al.

The theoretical complexity of the control dependence algorithm is as follows. Using bit vectors, set operations *member*, *insert* and *delete* are achieved in complexity $O(1)$. Set *union*, *difference* and *intersection* are achieved in time $O(N)$. For a control flow graph with N vertices, the initial post-dominator set for each vertex N may change at most N times, giving N^2 possible changes. The update following each change requires the computation of a post-dominator set for each vertex, each computation requiring time $O(N^2)$. Total complexity is therefore $O(N^2.(N).N^2)$, i.e. $O(N^5)$. Both computation of the post-dominator tree and the edge set S each require set operation with complexity $O(N)$. Calculation of the control dependencies is finally achieved in time $O(N.E)$, where E is the number of edges in S. The least common parent for each edge in the post-dominator tree is found in time $O(N)$, where N is the worst case height of the tree. Overall complexity for the complete algorithm is therefore:

$$O(N^5) + O(N) + O(N) + O(NE) \approx O(N^5)$$

The most inefficient step in the algorithm occurs in the computation of the reverse control flow graph's post-dominators. A simple algorithm taken from Aho et al[3] is used by the prototype system, replacing the conceptually more difficult algorithm reported by Ferrante et al. However, the theoretical complexity of this simple algorithm is $O(N^5)$, in contrast to the $O(N\alpha(N))$ for the more difficult algorithm.

The control dependence algorithm is simple to implement and correctly computes dependencies

for C programs including any of the language's control structures. However, the Prolog implementation was found to be inefficient, taking up to 4750 seconds of cpu time for the largest program analysed (1000 lines of code). The algorithm is applied on the control flow graph of each of the subject program's FCCGs and also on the control flow subgraph of each actual parameter list. Therefore any inefficiencies in the algorithm are repeated as the number of functions and function calls increases.

A practical CCG system would therefore require implementation of this more efficient algorithm. Another improvement could be achieved by detecting functions which involve only structured control flow statements. In this case, the corresponding control dependencies can be determined simply by examining the syntax of the function. The control dependencies reflect the nesting structure of the function's control flow statements and are trivial to compute.

7.3.2 Interprocedural Control Flow Analysis

Interprocedural control flow analysis is performed to connect the partial FCCG subgraphs. Call edges, parameter binding edges and return expression-use edges are each constructed.

The interprocedural control flow analysis algorithms have theoretical complexity as follows. Each function call requires time $O(P + R)$, where P represents the number of actual parameters and R the number of return statements in the called functions. For a program with C call sites, overall complexity is $O(C.P + R)$.

The algorithms are simple to implement in Prolog and the prototype system successfully connects the FCCGs. Efficiency was adequate with the largest subject program **migrate** with 294 function calls requiring 106 seconds. This suggests that interprocedural control flow analysis of larger programs will not be prohibitively expensive.

7.3.3 Data Dependence Analysis

The data dependence analysis phase uses the algorithm described in section 4.5.3, which is based on that described by Horwitz et al[42]. The algorithm has two phases, the *reaching stores* phase computes at each CCG vertex a set of store graphs representing the possible memory layouts that could arise during execution; the *inference phase* examines the set of store graphs at each CCG vertex to create the program's data dependencies.

The reaching stores phase iterates an initial store graph throughout the program's control flow graph, updating the store graph to represent the semantic effects of each program vertex, until achieving a fixed point result. Each update at a vertex is achieved in time $O(VO.N)$, where VO

represents the number of program objects accessed and N the cost of updating the *last_def* and *points_to* fields to represent the semantic effects. The inference phase requires time $O(TO)$, where TO represents the total number of objects accessed at each vertex of the program. Construction of flow dependencies for each object is achieved in constant time.

The results achieved using the prototype system demonstrate that where the number of paths is small, the data dependencies are constructed successfully. As the number of paths becomes large, for example with the **knap** and **migrate** programs, a fixed-point solution could not be achieved with the current system.

The reasons why a fixed-point was not reached can be explained. For programs with nested loop structures, the number of possible paths through the program quickly becomes large. For example, a nested while loop structure gives rise to seven paths, assuming each loop has zero, one or two iterations. The number of paths is further increased by the need to detect loop carried dependencies. Such dependencies occur whenever a variable is defined on one iteration of a loop and is then used on a subsequent iteration. Detection of these dependencies will require that the loop involved is analysed on sufficient iterations to uncover the dependence. For ordinal variables, two complete iterations will suffice. For programs involving pointers and recursive structures, the number of iterations required may be unbounded as dynamic allocation introduces new program objects on each subsequent iteration.

The prototype implementation makes use of only simple user intervention to limit the analysis required in these situations. The user is prompted on reaching a dynamic allocation statement whether to introduce a new dynamic object. If the user wishes not to introduce a new variable, analysis of the current path ceases. No other techniques are used to help reduce the number of paths to be analysed.

Whilst this approach is acceptable for the small examples, the **knap** and **migrate** programs showed that additional approximations are necessary if an algorithm of this type is to be successful with larger programs. Horwitz et al[42] make use of two techniques, *condensation* and *collapsing*. The condense operation limits the size of a store representing a recursive structure to a depth k. In this way the number of iterations associated with the recursive structure is bounded. The collapse operation limits the size of a set of states at a program vertex by applying an equivalence relation between different store graphs. Use of this equivalence relation means that a fixed point is achieved more quickly in certain cases.

An improved implementation making use of the condense and collapse operation would reduce the number of paths to be analysed. However it is not clear whether this reduction is sufficient to

overcome the problem of the large number of paths arising from the program's control structures. Other techniques may also be required, for example limiting the number of loop iterations to a specified maximum. For example a common approach in program testing is to consider zero, one or two iterations. Another approach could be to combine the static data dependence analysis with dynamic analysis techniques. In this way, run time information can be used to reduce the analysis. Information such as the value of input variables will help to reduce the number of paths as the outcome of some or all of the program's conditionals is known.

7.4 CCG Space Requirements

The space requirements of the CCG representation were found to be up to an order of magnitude larger than that of the original source code. Adding further dependence types such as declaration dependencies, or introducing multiple representations of single functions to help program slicing would further increase the size of the CCG. Nevertheless in its present form, the CCG is not prohibitively expensive in terms of space and could be reduced further by producing less verbose Prolog facts. At present the actual fact names make up a large proportion of the space taken.

The number of facts making up the CCG was found to be around three to four times the number of lines of source code analysed. The fine-grained approach taken in the CCG has not prohibitively increased the size of the representation from a statement based graph such as the UIG, whilst savings are made over parse tree based representations.

7.5 Summary

This chapter has evaluated the CCG. The chapter first addressed the language coverage provided by the CCG. The theoretical CCG permits the representation of expressions having embedded side effects or embedded control flows, value-returning functions, value parameters, pointer variables, array variables, pointer parameters, structure variables, recursive structures, C control structures and standard library routines. The CCG can easily be extended to represent component declaration and scoping information. Only pointers to functions, union variables and C preprocessor constructs are not permitted. The prototype system enables the construction of a CCG with only two main limitations. Extra program information provided by the prototype translator may be of use in the development of an enhanced CCG.

The CCG provides many different program views, covering intra and interprocedural control and data flow information, program slicing and ripple analysis. Each view is simple to construct.

The fine-grained approach taken in the CCG provides more accurate program slices and ripple analyses, however the explicit representation of interprocedural data dependencies and lack of context information can in certain cases lead to over-conservative solutions. Cross reference information can easily be added to the CCG. The prototype system allows construction of many of the possible views but requires further work to develop a better user interface.

The algorithms used in the construction of the CCG were finally evaluated. The CCG translation step produced encouraging results and could be used in the analysis of larger programs. Control dependence analysis was adequate but a time complexity of $O(N^5)$ suggests problems with larger subject systems. However, more efficient algorithms are available. Interprocedural control analysis techniques were also found to be acceptable. Disappointing results were achieved in the data dependence analysis steps. Techniques must be introduced to help reduce the number of program paths to be analysed if a data dependence solution is to be achieved for large programs.

Chapter 8

Conclusions

This chapter first presents a summary of the research described in this thesis. The success of the research is then addressed in relation to the criteria outlined in chapter 1. Finally ideas for extending the research are detailed.

8.1 Summary of Research

Software maintenance environments have been created by grouping together sets of static analysis tools such as control flow analysers, data flow analysers, program slicers and call graph builders. These environments suffer from the fact that each of the constituent tools rely on intermediate program representations. Consequently a maintenance environment will require a number of different program representations, giving repetition of information and inefficient use of storage space.

This problem was identified by Harrold and Malloy[38][39], who proposed the Unified Interprocedural Graph (UIG) dependence-based representation. The UIG incorporates a number of Intermediate Program Representations (IPRs), eliminating redundancies wherever possible and providing the maintainer with a number of different code views. However the UIG is able to represent only a small language.

This research has extended the ideas of the UIG to allow the representation of programs written in the C Programming language. The new representation, the Combined C Graph (CCG), in addition to the language features addressed by the UIG is able to model C programs with expressions with embedded side effects or embedded control flows, value parameters, C control structures, pointer variables, array variables, pointer parameters, external and static variables and standard library routines.

The CCG is a fine-grained dependence-based program representation. Each function of the

C program is represented by a Function CCG (FCCG). The vertices of the FCCG represent the statements and conditionals of the function. Expressions with embedded side effects and control flows are broken down to create 'fine-grained' vertices. The vertices of an FCCG are connected by a number of different edge types. Data and control dependence edges represent the program's data and control relationships. New *expression-use* and *lvalue-definition* edges connect the fine-grained vertices. Each FCCG is connected by a number of interprocedural edges. *Call* and *binding* edges represent the program's call and parameter binding relationships. *Return expression-use* edges represent value-returning functions. The CCG also contains an explicit representation of the program's interprocedural data dependencies. This approach differs from earlier IPRs such as the SDG[45] and UIG which propagate intraprocedural definitions and uses via an encapsulated interprocedural interface. The use of pointer parameters and recursive structures in C precludes this approach. Explicit representation of each object passed at a call site is not possible without introducing approximations.

The CCG provides the software maintainer with a number of programming level views which are each simple to construct. Call graphs, intraprocedural and interprocedural data and control dependence information are each available as simple CCG subgraphs. Flow-sensitive data flow information is also presented as annotations to the CCG. Program slices and ripple analyses can be constructed by simple graph traversal algorithms. In some cases, the fine-grained approach used in the CCG can provide more accurate results than statement-based IPRs. However, the lack of context information caused by the interprocedural data dependence edges can also produce more conservative solutions.

Construction algorithms for the CCG have been devised. Construction of the representation's vertices is achieved using well known parsing techniques. Control dependence information is also created using well known algorithms. Data dependence analysis for languages involving pointer variables and dynamically allocated memory is an ongoing area of research. This research modifies the algorithm described by Horwitz et al[42] to permit the additional computation of flow-sensitive data flow information.

A prototype implementation was developed to demonstrate the feasibility of the CCG representation. The prototype comprises a *lex/yacc*/C analyser to provide the partial CCG representation in the form of a Prolog fact base. Control dependence and data dependence analysis algorithms implemented as Prolog meta-programs complete the fact base representation. The maintainer is then able to query the CCG fact base, using simple Prolog meta-programs to produce different program views. The CCG representation or a subset of the CCG may also be viewed using a

graphical display tool. The system was evaluated with C programs of up to one thousand lines of code and fifteen functions.

8.2 Success of Research

The criteria for success described in section 1.4 are repeated followed by an evaluation of the success of the research.

8.2.1 Criteria for Success

1. Different views of a C program are to be made available to a maintainer and these views should help the comprehension process. The interface presented to the maintainer should allow quick switching between views and should allow the maintainer to concentrate on areas of the program which are of particular interest. Views available should include call graphs, definition-use, data flow, control dependence and program slices. These views should be created quickly and should provide the maintainer with accurate and useful information.

2. The level of coverage of the C language that is provided by the representation. Of particular importance are features such as pointers, embedded side-effects, embedded control flows and value-returning functions.

3. The accuracy of the representation. Language features with dynamic effects, such as self-referential structures and arrays will require approximations in order to be modelled statically. These approximations must still provide the maintainer with useful information.

4. Practical application of any program comprehension tool requires that the tool be able to deal with large programs, since it is precisely with such systems that the most significant problems in understanding occur. The new intermediate representation must enable large programs to be modelled, both in theory and in any practical implementation. Construction algorithms must not be prohibitively expensive whilst the resulting representation should be space efficient.

8.2.2 Success of Research

1. The CCG presents a variety of programming level views to a maintainer. Control dependence, call graph and intraprocedural and interprocedural data dependence are each formed as simple CCG subgraphs. Flow sensitive data flow information is available as graph annotations whilst

program slices and ripple analyses are constructed by simple traversals of the CCG. Each view is simple to construct.

The scenarios outlined in chapter 6 show how the use of the different views can help a maintainer gather information on areas of code which are of interest and acquire a variety of different information useful for program comprehension. However, the current prototype system presenting only a text-based Prolog interface with a limited graphical facility is insufficient for effective program comprehension. This problem is further compounded as the information is only made available in terms of CCG vertices and not the original C statements. An effective interface should present information both graphically and textually and allow the maintainer to switch easily between different views and levels of information. Other information on program components or at higher levels of abstraction would also be beneficial.

2. The CCG is successful in modelling a large proportion of the features of the C language. Expressions with embedded side effects or embedded control flows are modelled using fine-grained graph vertices. Value-returning functions with value or pointer parameters, pointer variables, structure variables, recursive structures and array variables may each be represented by the CCG. Each of the C control constructs can be modelled, whilst the standard library routines are represented using simple stub routines. Component declarations and scoping information may be introduced by the addition of declaration vertices. C features not represented by the CCG are pointers to functions, union variables and the C preprocessor constructs.

3. Approximations in the construction of the CCG are introduced in the computation of the program's data dependencies. Array variables are represented as aggregates and hence more refined information on the individual elements is not provided. Pointer and structure variables are decomposed to calculate information on their primitive components. However, recursive structures may only be analysed to a depth k. Information beyond this depth is lost. Whilst the data dependence information of the CCG is only approximate, useful information was constructed for each of the test programs analysed.

4. The CCG representation is able to model subject programs of any size. With the exception of the CCG's data dependence information, the prototype implementation was able to demonstrate that the representation is able in practice to deal with large programs. The initial CCG translation step analysed around 12kb of source code per second, whilst the intraprocedural

and interprocedural control dependence analysis, although relatively slower, were still able to analyse successfully the larger test programs.

The space requirements of the CCG representation were demonstrated to be an order of magnitude greater than that of the subject program. This result is not prohibitive and suggests that a CCG representation for large C programs is feasible.

8.3 Further Work

A number of improvements and extensions are possible to the research described in this thesis. Of primary importance is that a representation is developed that is able to model the C language in its entirety. The current CCG omits pointers to functions and union variables. Representations must be found for each of these features. The standard library, currently modelled by simple stub CCGs also requires further analysis to construct a complete and accurate representation.

A method to incorporate the language primitives of the C preprocessor also requires investigation. As described in chapter 7, the preprocessor can lead to multiple versions of the same program and problems in analysing incomplete syntactic units. A maintainer will benefit from information on both un-preprocessed and preprocessed code.

A second area of further work is the development of improved algorithms for the construction of the CCG. The present algorithm used to create the representation's data dependencies was found to be inadequate when applied to larger subject programs, due to the large number of paths which must be analysed. One possible approach to solving this problem would be to additionally incorporate dynamic program information to help constrain the number of paths through the program. By fixing the values of some or all inputs, the number of paths may be reduced significantly. Symbolic execution methods may also be a useful way of reducing paths or of eliminating the need for user intervention at dynamic allocation statements. Another solution is to employ a knowledge-based approach using techniques such as program plans in addition to programmer guidance to constrain the analysis to particular code areas. Improved algorithms are also required to improve the construction of the CCG's control dependencies.

Another area of further research is to extend the range of information provided to the maintenance programmer. At the programming level, declarations of program components may be incorporated into the representation. Improved program slicing algorithms introducing context information will in certain cases provide smaller and hence more useful results. Other research, for example by von Mayrhauser and Vans[86] has shown the benefits of introducing information at

higher levels of abstraction. An extended representation could incorporate design and requirements information with links between the different levels, giving full traceability of information throughout the life-cycle. New techniques for program slicing and impact analysis would traverse these different abstraction levels to provide a maintainer with a full range of domain and programming information.

Presentation of information to a maintainer is also an important area of research. Multiple window environments together with the use of colour and hypertext links to allow switching between between different views and levels of abstraction are commonly used techniques. The CCG is ideally suited to these kind of presentation methods. More advanced research into visualisation has proposed the use of virtual reality techniques[35].

As subject programs increase in size, the costs involved in construction and update will increase. A final area of further work is to investigate the incorporation of incremental approaches to the construction and update of the CCG. For large subject systems, in some cases incremental methods may allow the construction of a CCG representation for the limited area of the system which is of interest. This will reduce costs in both space and time. Incremental updates of the CCG in response to program changes may further reduce the costs involved and enable a maintainer to observe interactively the effects of program changes, even for large subject systems.

8.4 Summary

This chapter has presented a summary of the research described in this thesis in the development of the CCG representation. The criteria for success of the research described in section 1.4 have each been addressed and a number of areas of further research work outlined. These areas include improved language coverage, construction algorithms, additional views, improved information presentation and incremental construction and updates of the CCG.

Appendix A

CCG Fact Base

Program component facts

File fact

Syntax:

file(File-Name, C-File-Id).

Semantics:

A *file-fact* is produced for every translation unit, i.e. input C file. The *file-fact* contains a C-File-Id given by the *ccg_trans* linker and a File-Name, the name of the translated C file.

Example:

file('main.c', 1).

Type fact

Syntax:

type(C-File-Id, Function, sc(Stmt-Block, Storage-Specifier), Type-specifier, Name, Access-List).

Semantics:

A *type-fact* is produced for every name defined as a type. The scope is determined from the C-File-Id, Function, Stmt-Block and the Storage-Specifier. Function is the name of the function in which the type definition is contained. Stmt-Block is the C block in which the type definition appears. Storage-Specifier can have the value `extern`, `static` or `0`, representing 'not defined'.

Type-Specifier and Access-List specify the type represented by Name. Where Type-Specifier is a `struct`, `enum` or `union` definition with a tag name, a *tag fact* is produced and the tag name is used instead of the complete type definition. The Access-List represents the declarator.

Example:

```
typedef struct list LIST, *LIST_PTR;
```

becomes:

type(0, '@external', sc([], '@'), [struct, 'list'], 'LIST', []).

type(0, '@external', sc([], '@'), [struct, 'list'], 'LIST_PTR', ['@pointer']).

Tag fact

Syntax:

tag(C-File-Id, Function, sc(Stmt-Block, Storage-Specifier), Tag-Type, Name, Member-List).

Semantics

A *tag fact* is created for a `struct`, `enum` or `union` definition if a tag name is given. Tag-Type has the value `struct`, `enum` or `union`. Member-List contains mem(Type-Specifier, Name, Access-List) terms as elements.

Examples:

```
struct list {
  char *name;
  LIST_PTR next;
};
```

becomes:

tag(0, '@external', sc([], '@'), struct, 'list', [mem(char, 'name', ['@pointer']), mem('LIST_PTR', 'next', [])]).

and:

```
enum colour {blue, yellow, red, black} brush;
```

becomes:

tag(0, '@external', sc([], '@'), enum, 'colour', ['blue', 'yellow', 'red', 'black']).

Object fact

Syntax:

object(C-File-Id, Function, sc(Stmt-Block, Storage-Specifier), Type-Specifier, Name, Access-List).

Semantics

Each C component definition or declaration, including functions, produces an *object fact.* The enum example above produces:

object(0, '@external', sc([], '@'), [enum, 'colour'], 'brush', []).

The example `char (*(*x())[])()` is translated into:

object(0, '@external', sc([], '@'), char, 'x', ['@fun', '@pointer', '@array', '@pointer', '@fun']).

Node fact

Syntax:

node(C-File-Id, Function, Node-Id, Node-Type, Label, Variable-List).

Semantics

C-File-Id, Function and Node-Id uniquely identify an individual CCG vertex. Node-Type is the kind of statement represented by the vertex.

Node-Type ∈ { expr, formal, call, entry, allocate, end, return, case_label, default, continue, goto, label, break }

Label represents either a label associated with a label vertex or goto vertex, or the name of the callee function at a call vertex. With other vertex types label is set to the null value '@'. Variable-list is a list representation of the semantic effects of an expression node and contains items from the set:

{(ref, @string), (ref, @constant), (ref, name), (ref, .fieldname), (ref, '*'), (ref, '@'), (address, name), (address, '@'), (address, '*'), (rval, '@'), (def, name), (def, .fieldname), (def, '*'), (def, '@'), (predef, '@'), (postdef, '@') }

The former of each pair, either ref, address, def, postdef or predef, indicates that the object given by the latter is either referenced, has its address taken, is defined, is post-defined or is pre-defined. The object involved is determined by applying in sequence the operations given by the latter of each pair. Starting with (_, name) the object involved is that referred to by name. The next tuple may then be (_, '@'), indicating the object is still that given by name, (_, .fieldname) indicating that the new object is given by name.fieldname, or (_, '*'), indicating that the new object is *name. The values '@string' and '@constant' indicate string and other constant values, whilst the tuple (rval, '@') serves as a separator between the expressions on the right and left hand sides of an assignment operator.

Examples:

```
*x = y + z.f;
```

within `main` becomes:

node(0, 'main', 12, expr, '@', [(ref, 'y'), (ref, 'z'), (ref, '.f'), (rval, '@'), (ref, 'x'), (def, '*')]).

where 12 is a unique Node-Number within `main`.

```
sq();
```

within `main` becomes:

node(0, 'main', 13, call, 'sq', []).

where 13 is a unique Node-Number within `main`.

Intraprocedural edge facts

Edge fact

Syntax:

edge(C-File-Id, Function, From-Node-Id, To-Node-Id, Edge-Label).

Semantics:

The *edge fact* represents an intraprocedural control flow edge between two CCG vertices From-Node-Id and To-Node-Id. Edge-Label $\in$ { true, false, uncond }.

Example:

edge(0, 'main', 12, 13, 'uncond').

represents an unconditional edge from vertex 12 to vertex 13 within `main`.

Control fact

Syntax:

control(C-File-Id, Function, From-Node-Id, To-Node-Id, Control-Label).

Semantics:

The *control fact* represents an intraprocedural control dependence or intraprocedural switch dependence from From-Node-Id to To-Node-Id. Control-Label $\in$ { true, false, switch }.

Example:

control(0, 'main', 16, 21, 'true').

represents a control dependence from vertex 16 to vertex 21 within `main`.

Expuse fact

Syntax:

expuse(C-File-Id, Function, From-Node-Id, To-Node-Id).

Semantics:

The *expuse fact* represents an expression-use edge from From-Node-Id to To-Node-Id.

Example:

expuse(0, 'main', 18, 19, 'true').

represents an expression-use edge from vertex 18 to vertex 19 within main.

Lvaldef fact

Syntax:

lvaldef(C-File-Id, Function, From-Node-Id, To-Node-Id).

Semantics:

The *lvaldef fact* represents an lvalue-definition edge from From-Node-Id to To-Node-Id.

Example:

lvaldef(0, 'main', 22, 23, 'true').

represents an lvalue-definition edge from vertex 22 to vertex 23 within main.

Call interface facts

Call fact

Syntax:

call(From-C-File-Id, From-Function, From-Node-Id, To-C-File-Id, To-Function, To-Node-Id).

Semantics:

The *call fact* represents a call edge between a CCG call vertex and a CCG entry vertex.

Example:

call(0, 'main', 13, 0, 'sq', 0)

represents a call edge from call node 13 within main to entry node 0 within sq.

Bind fact

Syntax:

bind(From-C-File-Id, From-Function, From-Node-Id, To-C-File-Id, To-Function, To-Node-Id).

Semantics:

The *bind fact* represents a binding edge between two unique CCG vertices.

Example:

bind(0, 'main', 14, 0, 'sq', 1)

represents a binding edge from vertex 14 within main to vertex 1 within sq.

Return_expuse fact

Syntax:

return_expuse(From-C-File-Id, From-Function, From-Node-Id, To-C-File-Id, To-Function, To-Node-Id).

Semantics:

The *return_expuse fact* represents a return-expression-use edge between two unique CCG vertices.

Example:

return_expuse(0, 'sq', 7, 0, 'main', 24)

represents an return-expression-use edge from vertex 7 within `sq` to vertex 24 within `main`.

Data dependence fact

Flow fact

Syntax:

flow(From-C-File-Id, From-Function, From-Node-Id, To-C-File-Id, To-Function, To-Node-Id).

Semantics:

The *flow fact* represents an intraprocedural or interprocedural flow dependence between two unique CCG vertices.

Example:

flow(0, 'main', 12, 0, 'sq', 5)

represents an interprocedural flow dependence from vertex 12 within `main` to vertex 5 within `sq`.

Appendix B

Example CCG Fact Base

The following is a listing of the Prolog fact base produced by the CCG system following the analysis of the **sum** program.

```
bind(1, 'CalcSum', 4, 1, 'Inc', 1).
bind(1, main, 5, 1, 'CalcSum', 1).
bind(1, main, 6, 1, 'CalcSum', 2).

call(1, 'CalcSum', 3, 1, 'Inc', 0).
call(1, main, 4, 1, 'CalcSum', 0).

control(1, 'CalcSum', 6, 7, true).
control(1, 'CalcSum', 0, 8, true).
control(1, 'CalcSum', 0, 6, true).
control(1, 'CalcSum', 0, 3, true).
control(1, 'CalcSum', 0, 2, true).
control(1, 'CalcSum', 0, 1, true).
control(1, 'Inc', 0, 2, true).
control(1, 'Inc', 0, 1, true).
control(1, main, 3, 3, true).
control(1, main, 3, 8, true).
control(1, main, 3, 4, true).
control(1, main, 0, 10, true).
control(1, main, 0, 9, true).
control(1, main, 0, 3, true).
control(1, main, 0, 2, true).
control(1, main, 0, 1, true).
control(1, 'CalcSum', 3, 4, true).
control(1, main, 4, 6, true).
control(1, main, 4, 5, true).
```

```
edge(1, 'CalcSum', 0, 1, true).
edge(1, 'CalcSum', 0, 9, false).
edge(1, 'CalcSum', 1, 2, uncond).
edge(1, 'CalcSum', 2, 3, uncond).
edge(1, 'CalcSum', 3, 4, true).
edge(1, 'CalcSum', 3, 5, false).
edge(1, 'CalcSum', 3, 6, uncond).
edge(1, 'CalcSum', 4, 5, uncond).
edge(1, 'CalcSum', 6, 7, true).
edge(1, 'CalcSum', 6, 8, false).
edge(1, 'CalcSum', 7, 8, uncond).
edge(1, 'CalcSum', 8, 9, uncond).
edge(1, 'Inc', 0, 1, true).
edge(1, 'Inc', 0, 3, false).
edge(1, 'Inc', 1, 2, uncond).
edge(1, 'Inc', 2, 3, uncond).
edge(1, main, 0, 1, true).
edge(1, main, 0, 11, false).
edge(1, main, 1, 2, uncond).
edge(1, main, 10, 11, uncond).
edge(1, main, 2, 3, uncond).
edge(1, main, 3, 4, true).
edge(1, main, 3, 9, false).
edge(1, main, 4, 5, true).
edge(1, main, 4, 7, false).
edge(1, main, 4, 8, uncond).
edge(1, main, 5, 6, uncond).
edge(1, main, 6, 7, uncond).
edge(1, main, 8, 3, uncond).
edge(1, main, 9, 10, uncond).

expuse(1, main, 1, 2).

file('../examples/thesis_ch3/thesis_ch3.c', 1).

flow(1, 'CalcSum', 2, 1, 'CalcSum', 4).
flow(1, 'CalcSum', 1, 1, 'CalcSum', 6).
flow(1, 'CalcSum', 2, 1, 'CalcSum', 7).
flow(1, 'Inc', 2, 1, 'CalcSum', 7).
flow(1, 'CalcSum', 1, 1, 'CalcSum', 7).
flow(1, 'CalcSum', 7, 1, 'CalcSum', 8).
flow(1, 'CalcSum', 1, 1, 'CalcSum', 8).
flow(1, 'Inc', 1, 1, 'Inc', 2).
flow(1, main, 2, 1, 'Inc', 2).
flow(1, 'Inc', 2, 1, 'Inc', 2).
flow(1, main, 8, 1, main, 10).
flow(1, main, 1, 1, main, 10).
flow(1, main, 2, 1, main, 3).
flow(1, 'Inc', 2, 1, main, 3).
```

```
flow(1, main, 1, 1, main, 5).
flow(1, main, 8, 1, main, 5).
flow(1, 'Inc', 2, 1, main, 9).
flow(1, main, 2, 1, main, 9).


node(1, 'CalcSum', 0, entry, @, []).
node(1, 'CalcSum', 1, formal, @, [ex(def, s)]).
node(1, 'CalcSum', 2, formal, @, [ex(def, j)]).
node(1, 'CalcSum', 3, call, 'Inc', []).
node(1, 'CalcSum', 4, expr, @, [ex(ref, j)]).
node(1, 'CalcSum', 5, end_params, @, []).
node(1, 'CalcSum', 6, expr, @, [ex(ref, '@constant'),ex(ref, s)]).
node(1, 'CalcSum', 7, expr, @, [ex(ref, j),ex(ref, *),ex(ref, s),ex(rval, @),ex(def, s)])
node(1, 'CalcSum', 8, expr, @, [ex(ref, s)]).
node(1, 'CalcSum', 9, end, @, []).
node(1, 'Inc', 0, entry, @, []).
node(1, 'Inc', 1, formal, @, [ex(def, x)]).
node(1, 'Inc', 2, expr, @, [ex(ref, '@constant'),ex(ref, x),ex(ref, *),ex(rval, @),
     ex(ref, x),ex(def, *)]).
node(1, 'Inc', 3, end, @, []).
node(1, main, 0, entry, @, []).
node(1, main, 1, expr, @, [ex(ref, '@constant'),ex(rval, @),ex(def, sum)]).
node(1, main, 10, expr, @, [ex(ref, sum),ex(rval, @),ex(def, sum)]).
node(1, main, 11, end, @, []).
node(1, main, 2, expr, @, [ex(rval, @),ex(def, i)]).
node(1, main, 3, expr, @, [ex(ref, '@constant'),ex(ref, i)]).
node(1, main, 4, call, 'CalcSum', []).
node(1, main, 5, expr, @, [ex(ref, sum)]).
node(1, main, 6, expr, @, [ex(address, i)]).
node(1, main, 7, end_params, @, []).
node(1, main, 8, expr, @, [ex(ref, 'CalcSum'),ex(rval, @),ex(def, sum)]).
node(1, main, 9, expr, @, [ex(ref, i),ex(rval, @),ex(def, i)]).

object(1, '@external', sc([], @), int, 'CalcSum', ['@fun']).
object(1, '@external', sc([], @), int, 'CalcSum', ['@fun']).
object(1, '@external', sc([], @), void, 'Inc', ['@fun']).
object(1, '@external', sc([], @), void, 'Inc', ['@fun']).
object(1, '@external', sc([], @), void, main, ['@fun']).
object(1, 'CalcSum', sc([], @), int, j, ['@pointer']).
object(1, 'CalcSum', sc([], @), int, s, []).
object(1, 'Inc', sc([], @), int, x, ['@pointer']).
object(1, main, sc([1], @), int, i, []).
object(1, main, sc([1], @), int, sum, []).

return_expuse(1, 'CalcSum', 8, 1, main, 8).
```

Appendix C

Test Program Listings

The following is a listing of the four smaller programs analysed using the CCG system.

C.1 Trityp.c

```
/*************************************************************/
/* Ramamoorthy's triangle Trityp                             */
/* This program reads in 3 sides of a triangle and outputs   */
/* the type of the triangle.                                 */
/*                                                           */
/*************************************************************/

#include <stdio.h>

int a,b,c,d;


main()
{
 scanf("%d%d%d", &a, &b, &c);
 if (a >= b && b >= c)
    {
     if (a == b || b == c)
        {
         if (a == b && b == c)
             printf("%s\n", "Equilateral");
         else
             printf("%s\n", "Isosceles");
        }
     else
         { a = a * a;
           b = b * b;
           c = c * c;
           d = b + c;
           if (a != d)
              {
               if (a < d)
                  printf("%s\n", "Acute");
               else
                  printf("%s\n", "Obtuse");
              }
           else
               printf("%s\n", "Right Angled Triangle");
         }
     }
 else
    printf("%s\n","Triangle Sides not in order");
}
```

C.2 Sum.c

```
/*************************************************************/
/* lines                                                     */
/* This program sums integers up to maximum total of 100.    */
/*                                                           */
/*************************************************************/

#include <stdio.h>

void Inc(int *x);
int CalcSum(int s, int *j);


void main()
{
 int sum;
 int i;

 i = sum = 0;
 while (i < 20) {
   sum = CalcSum(sum, &i);
 }
 i = i;
 sum = sum;
}


int CalcSum(int s, int *j)
{
 Inc(j);
 if (s < 100) {
    s = s + *j;
 }
 return s;
}


void Inc(int *x)
{
 *x = *x + 1;
}
```

C.3 Linked_list.c

```
/*************************************************************/
/* linked_list                                               */
/* T.P. Love - 'ANSI C for Programmers on UNIX Systems'      */
/* This program constructs a linked list of items and        */
/* prints out the list.                                      */
/*                                                           */
/*************************************************************/

#include <stdio.h>

typedef struct _list_item {
  int val;
  struct _list_item *next;
} list_item;


/* prototypes */
list_item *add_list_item(list_item *entry, int value);
void print_list_items(void);

list_item *head;
list_item *tail;


main(int argc, char *argv[])
{
 head = NULL;
 tail = NULL;

 tail = add_list_item(tail, 5);
 tail = add_list_item(tail, 7);
 tail = add_list_item(tail, 2);

 print_list_items();
}


list_item *add_list_item(list_item *entry, int value)
{
 list_item *new_list_item;

 new_list_item = (list_item*) malloc(sizeof(list_item));

 if (entry == NULL) {
    head = new_list_item;
    printf("First list_item in list\n");
```

```
 }
 else {
    entry->next = new_list_item;
    printf("Adding %d to list. Last value was %d \n", value, entry->val);
 }
 new_list_item->val = value;
 new_list_item->next = NULL;
 return new_list_item;
}


void print_list_items(void)
{
 list_item *ptr_to_list_item;

 for (ptr_to_list_item = head; ptr_to_list_item != NULL;
      ptr_to_list_item = ptr_to_list_item->next) {
     printf("Value is %d \n", ptr_to_list_item->val);
 }
}
```

C.4 Lines.c

```
/*************************************************************/
/* Kernighan & Ritchie  - 'The C Programming Language'       */
/* 2nd edition Pages 108-110                                 */
/* This program reads in lines and outputs them in sorted    */
/* order.                                                    */
/*                                                           */
/*************************************************************/

#include <stdio.h>
#include <string.h>

#define MAXLINES 10     /* max #lines to be sorted */
#define MAXLEN   30     /* length of input line */
#define ALLOCSIZE 100   /* available space */

static char allocbuf[ALLOCSIZE];
static char *allocp;

char *lineptr[MAXLINES];


char *alloc(n)
int n;
{
 if (allocbuf + ALLOCSIZE - allocp >= n) {
     allocp += n;
     return allocp - n;
 } else
     return 0;
 }


int getline (s, lim)
char s[];
int lim;
{
 int c, i;

 i = 0;
 while (--lim > 0 && (c = getchar()) != EOF && c != '\n')
    s[i++] = c;
 if (c == '\n')
    s[i++] = c;
 s[i] = '\0';
 return i;
}
```

```
int readlines(lineptr, maxlines)
char *lineptr[];
int maxlines;
{
 int len, nlines;
 char *p, line[MAXLEN];

 nlines = 0;
 while ((len = getline(line, MAXLEN)) > 0) {
   if (nlines >= maxlines)
      return -1;
   if ((p = alloc(len)) == NULL)
      return -1;
   line[len-1] = '\0';
   strcpy(p, line);
   lineptr[nlines++] = p;
 }
 return nlines;
}

writelines(lineptr, nlines)
char *lineptr[];
int nlines;
{
 while (nlines-- > 0)
    printf("%s\n", *lineptr++);
}

swap(v, i, j)
char *v[];
int i,j;
{
 char *temp;
 temp = v[i];
 v[i] = v[j];
 v[j] = temp;
}


qsort(v, left, right)
char *v[];
int left, right;
{
 int i, last;
 if (left >= right)
    return;
 swap(v, left, (left + right)/2);
```

```
 last = left;
 for (i = left + 1; i <= right; i++)
     if (strcmp(v[i], v[left]) < 0)
        swap ( v, ++last, i);
 swap(v, left, last);
 qsort(v, left, last-1);
 qsort(v, last+1, right);
}


main()
{
int nlines;

 allocp = allocbuf;
 if ((nlines = readlines(lineptr, MAXLINES)) >= 0) {
    qsort(lineptr, 0, nlines-1);
    writelines(lineptr, nlines);
    return 0;
 } else {
    printf("error : input too big to sort\m");
    return 1;
 }
}
```

Bibliography

[1] H. Agrawal and J.R. Horgan. Dynamic program slicing. *SIGPLAN Notices*, 25(6):246–256, June 1990.

[2] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, Reading, Massachusetts, 1983.

[3] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers : Principles, Techniques and Tools.* Addison-Wesley, Reading, Massachusetts, 1986.

[4] J. Ambras and V. O'Day. Microscope : a program analysis system. In *Proceedings of the 20th Hawaii International Conference on System Sciences*, pages 71–81, January 1987.

[5] ANSI. Ansi x3.159-1989 American National Standard for Information Systems - programming language C, 1989.

[6] P. Antonini, P. Benedusi, G. Cantone, and A. Cimitile. Low-level design documents production and improvement. In *Proceedings of the Conference on Software Maintenance, Austin, Texas*, pages 91–100, October 1987.

[7] G. Avellis, A. Iacobbe, D. Palmisano, G. Semeraro, and C. Tinelli. An analysis of incremental assistant capabilities of a software evolution expert system. In *Proceedings of the Conference on Software Maintenance, Sorrento, Italy*, pages 220–227, October 1991.

[8] J.P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Conference Record of the 6th Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 29–41, January 1979.

[9] J.M. Bieman, A.L. Baker, P.N. Clites, D.A. Gustafson, and A.C. Melton. A standard representation of imperative language programs for data collection and software measures specification. *Journal of Systems and Software*, 18(1):13–37, January 1988.

[10] T.M. Bodhuin. An interaction paradigm for impact analysis. Master's thesis, Department of Computer Science, University of Durham, Durham, UK, 1994.

[11] B.W. Boehm. The high cost of software. In E. Horowitz, editor, *Practical Strategies For Developing Large Software Systems.* Addison-Wesley, Reading, Massachusetts, 1975.

[12] B.W. Boehm. Software engineering. *IEEE Transactions on Computers*, 25(12):1226–1242, December 1976.

[13] R. Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18(6):543–554, June 1983.

[14] T.A. Bünter. PERPLEX : An extensible tool architecture for C source code. Technical Report Computer Science 6/93, School of Engineering and Computer Science, University of Durham, Durham, UK, 1993.

[15] D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, Atlanta, Georgia*, pages 47–56, June 1988.

[16] F.W. Calliss and B.J. Cornelius. Dynamic data flow analysis of C programs. In *Proceedings of the 21st Hawaii International Conference on System Sciences*, pages 518–523, January 1988.

[17] D.R. Chase, M. Wegman, and F.K. Zadeck. Analysis of pointers and structures. *SIGPLAN Notices*, 25(6):296–310, June 1990.

[18] Y.F. Chen, M.Y. Nishimoto, and C.V. Ramamoorthy. The C information abstraction system. *IEEE Transactions on Software Engineering*, SE-16(3):325–334, March 1990.

[19] L. Cleveland. An environment for understanding programs. In *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, pages 500–509, January 1988.

[20] L. Cleveland. A user interface for an environment to support program understanding. In *Proceedings of the Conference on Software Maintenance, Phoenix, Arizona*, pages 86–91, October 1988.

[21] K. Cooper. Analysing aliases of reference formal parameters. In *Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages, New Orleans, Louisiana*, pages 281–290, January 1985.

[22] N.C. Debnath and J.M. Bieman. A representation and analysis of interprocedural structure. In *Proceedings of the 20th Hawaii International Conference on System Sciences*, pages 92–100, January 1987.

[23] L.E. Deimel and J.F. Naveda. Reading computer programs: Instructor's guide and exercises. Educational Materials CMU/SEI-90-EM-3, Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1990.

[24] R.E. Fairley. *Software Engineering Concepts.* McGraw-Hill, 1985.

[25] J. Ferrante, K.J. Ottenstein, and J.D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[26] N.T. Fletton. Redocumentation for software maintenance and the redocumentation of existing systems. Master's thesis, School of Engineering and Applied Science, University of Durham, Durham, UK, 1988.

[27] N.T. Fletton and M. Munro. Redocumentation of systems using hypertext technology. In *Proceedings of the Conference on Software Maintenance, Phoenix, Arizona*, pages 54–59, October 1988.

[28] L.D. Fosdick and L.J. Osterweil. Data flow analysis in software reliability. *ACM Computing Surveys*, 8(3):305–330, September 1976.

[29] J.R. Foster. *Cost factors on software maintenance.* PhD thesis, School of Engineering and Computer Science, University of Durham, Durham, UK, 1993.

[30] J.R. Foster, A.E.P. Jolly, and M.T. Norris. An overview of software maintenance. *British Telecom Journal*, 7(4):37–48, October 1989.

[31] J.R. Foster and M. Munro. Documentation method based on cross referencing. In *Proceedings of the Conference on Software Maintenance, Austin, Texas*, pages 181–185, September 1987.

[32] K.B. Gallagher and J.R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, SE-17(8):751–761, August 1991.

[33] P.K. Garg and W. Scacchi. A hypertext system to manage software life cycle documents. In *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, pages 337–346, January 1988.

[34] P.K. Garg and W. Scacchi. A hypertext system to manage software life cycle documents. *IEEE Software*, 7(3):90–98, May 1990.

[35] R.D. Gregson. Virtual reality and program comprehension: application using spreadsheet visualisation. Master's thesis, Department of Computer Science, University of Durham, Durham, UK, 1995.

[36] W.G. Griswold and D. Notkin. Automated assistance for program restructuring. Technical Report CS92-221, Department of Computer Science and Engineering, University of California, San Diego, 1992.

[37] M.H. Halstead. *Elements of Software Science.* Elsevier North-Holland, New York, 1977.

[38] M.J. Harrold and B. Malloy. A unified interprocedural program representation for a maintenance environment. In *Proceedings of the Conference on Software Maintenance, Sorrento, Italy*, pages 138–147, October 1991.

[39] M.J. Harrold and B. Malloy. A unified interprocedural program representation for a maintenance environment. *IEEE Transactions on Software Engineering*, SE-19(6):584–593, June 1993.

[40] M.J. Harrold and M.L. Soffa. Computation of interprocedural definition and use dependencies. In *Proceedings of the IEEE Computer Society International Conference on Computer Languages, New Orleans, Louisiana*, pages 297–306, March 1990.

[41] E. Horowitz and R.C. Williamson. SODOS: a software documentation support environment - its definition. *IEEE Transactions on Software Engineering*, SE-12(8):849–859, August 1986.

[42] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. *SIGPLAN Notices*, 24(7):28–40, 1989.

[43] S. Horwitz, J. Prins, and T. Reps. On the adequacy of program dependence graphs for representing programs. In *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages, San Diego, California*, pages 146–157, January 1988.

[44] S. Horwitz, J. Prins, and T. Reps. Integrating noninterfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345–387, July 1989.

[45] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[46] J.C. Huang. Detection of data flow anomaly through program instrumentation. *IEEE Transactions on Software Engineering*, SE-5(3):226–236, May 1979.

[47] J. Jiang, X. Zhai, and D.J. Robson. Program slicing in C - the problems in implementation. In *Proceedings of the Conference on Software Maintenance, Sorrento, Italy*, pages 182–190, October 1991.

[48] W.L. Johnson and E. Soloway. PROUST: knowledge-based program understanding. *IEEE Transactions on Software Engineering*, SE-11(3):267–275, March 1985.

[49] D. Kafura and G.R. Reddy. The use of complexity metrics in software maintenance. *IEEE Transactions on Software Engineering*, SE-13(3):335–343, March 1987.

[50] B.W. Kernighan and D.M. Ritchie. *The C Programming Language.* McGraw-Hill, first edition, 1978.

[51] B.W. Kernighan and D.M. Ritchie. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, New Jersey, second edition, 1988.

[52] D.A. Kinloch and M. Munro. A combined representation for the maintenance of C programs. In *Proceedings of the IEEE 2nd Workshop on Program Comprehension, Capri, Italy*, pages 119–127, July 1993.

[53] 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*, pages 172–180, September 1994.

[54] W. Landi and B.G. Ryder. Pointer induced aliasing: A problem classification. In *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida*, pages 93–103, January 1991.

[55] W. Landi and B.G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. *SIGPLAN Notices*, 27(7):235–248, July 1992.

[56] S. Letovsky. Cognitive processes in program comprehension. In E. Soloway and S. Iyengar, editors, *Empirical Studies of Programmers*, pages 58–79. Ablex, Norwood, New Jersey, 1986.

[57] S. Letovsky and E. Soloway. Delocalized plans and program comprehension. *IEEE Software*, pages 41–49, May 1986.

[58] H.K.N. Leung and H.K. Reghbati. Comments on program slicing. *IEEE Transactions on Software Engineering*, SE-13(12):1370–1371, December 1987.

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

[60] P. Linos, P. Aubet, L. Dumas, Y. Helleboid, P. Lejeune, and P. Tulula. Facilitating the comprehension of C programs : an experimental study. In *Proceedings of the IEEE 2nd Workshop on Program Comprehension, Capri, Italy*, pages 55–63, July 1993.

[61] P.E. Livadas and S.D. Alden. A toolset for program understanding. In *Proceedings of the IEEE 2nd Workshop on Program Comprehension, Capri, Italy*, pages 110–118, July 1993.

[62] P.E. Livadas and S. Croll. Program slicing. Technical Report SERC-TR-61F, Computer and Information Sciences Department, University of Florida, Gainsville, Florida, October 1992.

[63] P.E. Livadas and P. Roy. Program dependence analysis. In *Proceedings of the Conference on Software Maintenance, Orlando, Florida*, pages 356–365, November 1992.

[64] T. Love. ANSI C programmers on UNIX systems. Technical report, Cambridge University Engineering Department, Cambridge, UK, December 1993.

[65] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, December 1976.

[66] S. McGowan. Fortune - an ipse documentation tool. Technical Report Alvey Project ALV/PRJ/SE/050, CAP(UK) Limited, 1987.

[67] K.J. Ottenstein and L.M. Ottenstein. The program dependence graph in a software development environment. *SIGPLAN Notices*, 9(3):177–184, May 1984.

[68] 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*, SE-20(5):385–402, May 1994.

[69] G. Parikh and N. Zvegintzov. *Tutorial On Software Maintenance.* IEEE Computer Society Press, Silver Spring, Maryland, 1983.

[70] B.H. Patkau. *A foundation for software maintenance.* PhD thesis, Department of Computer Science, University of Toronto, Toronto, Ontario, Canada, 1983.

[71] N. Pennington. Comprehension strategies in programming. In G.M. Olsen, S. Sheppard, and E. Soloway, editors, *Empirical Studies of Programmers: Second Workshop*, pages 100–113. Ablex, Norwood, New Jersey, 1987.

[72] M. Platoff and M. Wagner. An integrated program representation and toolkit for the maintenance of C programs. In *Proceedings of the Conference on Software Maintenance, Sorrento, Italy*, pages 129–137, October 1991.

[73] J-P Queille, J. Voidrot, N. Wilde, and M. Munro. The impact analysis task in software maintenance: a model and a case study. In *Proceedings of the International Conference on Software Maintenance, Victoria, British Columbia, Canada*, pages 234–242, September 1994.

[74] V. Rajlich, N. Damaskinos, P. Linos, and J. Silva. Visual support for programming in the large. In *Proceedings of the Conference on Software Maintenance, Phoenix, Arizona*, pages 92–99, October 1988.

[75] S.P. Reiss. PECAN: Program development systems that support multiple views. In *Proceedings of the ACM 7th International Conference on Software Engineering, Orlando, Florida*, pages 324–333, March 1984.

[76] C. Rich. *Inspection Methods in Programming.* PhD thesis, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Cambridge, Massachusetts, June 1980. Technical Report, AI-TR-604.

[77] C. Rich. A formal representation for plans in the Programmer's Apprentice. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence, Vancouver, British Columbia, Canada*, pages 1044–1052, August 1981.

[78] C. Rich and L.M. Wills. Recognizing a program's design: A graph-parsing approach. *IEEE Software*, pages 82–89, January 1986.

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

[80] B.G. Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, SE-5(3):216–225, May 1979.

[81] B. Schneiderman, P. Schafer, R. Simon, and L. Weldon. Display strategies for program browsing. *IEEE Software*, pages 7–15, May 1986.

[82] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, SE-10(5):595–609, September 1984.

[83] E. Soloway, J. Pinto, S. Letovsky, D. Littman, and R. Lampert. Designing documentation to compensate for delocalized plans. *Communications of the ACM*, 31(11):1259–1267, November 1988.

[84] I. Sommerville. *Software Engineering, 3rd edition.* Addison-Wesley, Reading, Massachusetts, 1989.

[85] T.A. Standish. An essay on software reuse. *IEEE Transactions on Software Engineering*, SE-10(5):494–497, September 1984.

[86] A. von Mayrhauser and A.M. Vans. From program comprehension to tool requirements for an industrial environment. In *Proceedings of the IEEE 2nd Workshop on Program Comprehension, Capri, Italy*, pages 78–86, July 1993.

[87] M. Ward, F.W. Calliss, and M. Munro. The maintainer's assistant. In *Proceedings of the Conference on Software Maintenance, Miami, Florida*, pages 307–305, October 1989.

[88] M. Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, July 1982.

[89] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[90] C. Wild and K. Maly. Towards a software maintenance support environment. In *Proceedings of the Conference on Software Maintenance, Phoenix, Arizona*, pages 80–85, October 1988.

[91] N. Wilde and R. Huitt. A reusable toolset for software dependency analysis. *Journal of Systems and Software*, 14(2):97–102, February 1991.

[92] L.M. Wills. Automated program recognition: A feasibility demonstration. *Artificial Intelligence*, 45(1-2):113–171, 1990.

[93] C. Wilson and L.J. Osterweil. Omega - a data flow analysis tool for the C programming language. *IEEE Transactions on Software Engineering*, SE-11(9):832–838, September 1985.

[94] S.S. Yau and J.S. Collofello. Some stability measures for software maintenance. *IEEE Transactions on Software Engineering*, SE-6(6):545–552, 1980.

[95] E. Yourdon. *Modern Structured Analysis.* Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[95] E. Yourdon. *Modern Structured Analysis.* Prentice-Hall, Englewood Cliffs, New Jersey, 1989.