Inter-module code analysis techniques for software maintenance

Calliss, Frank William

University of Durham

School of Engineering and Applied Science
(Computer Science)

Inter-Module Code Analysis Techniques for Software Maintenance

Frank William Calliss

Ph.D.

1989

Abstract

The research described in this thesis addresses itself to the problem of maintaining large, undocumented systems written in languages that contain a module construct. Emphasis is placed on developing techniques for analysing the code of these systems, thereby helping a maintenance programmer to understand a system. Techniques for improving the structure of a system are presented. These techniques help make the code of a system easier to understand.

All the code analysis techniques described in this thesis involve reasoning with, and manipulating, graphical representations of a system. To help with these graph manipulations, a set of graph operations are developed that allow a maintenance programmer to combine graphs to create a bigger graph, and to extract subgraphs from a given graph that satisfy specified constraints.

A relational database schema is developed to represent the information needed for inter-module code analysis. Pointers are given as to how this database can be used for inter-module code analysis.

Acknowledgements

This thesis is dedicated to my mother and sister. I would like to thank them for the support and encouragement that they have given me.

I wish to thank Mr. B.J. Cornelius for agreeing to be my supervisor, and for the invaluable advice and encouragement that he has given me.

I wish to thank Chuck Bilbe and Sun Microsystems for allowing me to refer in this thesis to parts of the code for the `m2dep` program.

This Ph.D. thesis has been produced using the LaTeX and BibTeX text formatting system. The typesetting of the VDM specifications is done using the `vdm` style file written by Mario Wolczko of the University of Manchester.

This work was supported by a studentship from the Science and Engineering Research Council.

Copyright

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Purpose of the Research

Analysis techniques are described that can be applied to programs consisting of interconnected **modules***. These analysis techniques are intended to help maintenance programmers understand the code of large, undocumented systems. Some of the techniques concentrate on analysing the **architectural structure** of a system, while others concentrate on analysing the module interfaces. The proposed techniques lead to methods for analysing the code of a system. These methods, together with their supporting software tools, can be used by maintenance programmers in order to understand a large system.

*In this thesis, words in bold typeface are defined in the glossary given in Appendix A

1.2 Motivation

In his Turing award address Dijkstra said,

> "As long as there were no machines, programming was no problem at all; when we had weak computers, programming became a mild problem and now that we have gigantic computers, programming has become an equally gigantic problem." [50]

This comment is referring to the problem of designing and implementing large complex systems that are beyond the intellectual capacity of the programmers working on them. Dijkstra's comment is equally valid when it comes to maintaining large complex programs.

As the power of the available machines has grown, so ever more complex programs have been developed that make use of the new power of the hardware. Many of these programs were written without the aid of any design method and as a result have a very complex structure. This complexity of structure seriously affects the productivity of maintenance programmers who have to spend over 50% ([33]) of their time on **code analysis** activities.

Software maintenance is the most costly stage of the software lifecycle, accounting for between 50% and 80% of software expenditure [90, 108]. An important part of any software maintenance task is code analysis. Code analysis is performed several times for any given maintenance task. Before any maintenance operation can be performed, a maintenance programmer has to have a general understanding of how a program works, together with knowledge about which sections of the code are important for the maintenance operation. This information can be obtained

by analysing the program code and determining the dependencies that exist between the different entities. The information gained from this analysis can be supplemented by information obtained from the documentation. After a proposed modification has been designed, another code analysis process is performed in order to ensure that the change will not have any unforeseen side effects. It is therefore important that techniques are developed that can help programmers in their code analysis work. An important part of the process of understanding programs written in module languages is determining the role that a module plays within a system, and determining the nature of the connections between modules. It is the production of techniques in this area that is the goal of the research described in this thesis.

1.3 Objectives of the Research

1.3.1 Assumptions

The work of this thesis is concerned with the analysis of the dependencies between the modules that comprise a software system. As such the work is aimed at languages that support a module construct as part of the language. In some languages modules have to be simulated (e.g., in C [86] a `.h` file can represent the specification part of a module and a `.c` file can represent the implementation part). The special requirements needed to consider such languages will not be addressed in this thesis.

Lisp-based languages like Flavours [107] and Common Lisp [13] determine many

of their inter-module dependencies dynamically. These languages will not be served by this thesis. Languages like Ada[†] [1], Modula-2 [169], Simula [5] and Eiffel[‡] [104, 105] determine their inter-module dependencies statically, and it is to this form of module language that the thesis will confine itself.

1.3.2 Goals

The research described in this thesis is aimed at the maintenance phase of the software lifecycle. The primary objective of the research is to find techniques to help a programmer understand an undocumented system. To this end, ways to analyse the architectural structure of a system are developed which give a clue as to what design strategy, if any, has been used on the system. This involves establishing the different module hierarchies within a system that one expects from the different design methods.

At a more detailed level, the role played by individual modules within a system is established. This helps a maintenance programmer to detect modules that are playing a particular role within a system. As a result of this, modules can be located that require closer examination because they belong to a category of modules that has been identified as being overly complex.

In order to be able to understand the role of a module within a system properly, it is necessary to derive and classify the nature of the connections between modules. To do this, techniques are developed to analyse the connections between modules, and thereby classify the connections according to some of the existing

[†]Ada is a registered trademark of the U.S. Government, Ada Joint Program Office
[‡]Eiffel is a trademark of Interactive Software Engineering Inc.

classifications.

Techniques are developed for taking a module that is found to be overly complex, because the module is found to provide several disparate services, and for breaking it into smaller and simpler modules. By breaking up complex modules into smaller modules that contain logically related entities, the structure of a system is then improved because each of the modules represents a single service.

When dealing with large systems, it is impractical to regenerate the information on the module interconnections for every code analysis operation. Therefore a relational database schema is designed that can record the necessary information, and the thesis provides indicators as to the way in which this database is best interrogated.

1.3.3 Anticipated Benefits

Time constraints make it impossible to study all the applications of inter-module code analysis for this thesis. Some of the anticipated benefits of this work are briefly described below. It is hoped to pursue some of these in future work.

Cohesion and Coupling

Cohesion is the functional strength of a **routine** and coupling is a measure of the dependence between routines. Some cohesion and coupling measures have been given in the literature for routines [116, 175]. With the module construct which provides clustering facilities these measures need to be reassessed.

For example, Yourdon and Constantine [175] classify two or more routines that use the same global variable as being *commonly coupled.* This is regarded as being a bad form of dependence between routines. This form of connection is used within a module to implement an abstract-state machine or an abstract data type. Both of these type of modules are classed as being "good". This means that with the existing measures, bad routine dependencies are needed to create some good modules. This is an undesirable situation so the existing cohesion and coupling definitions ought to be modified to take into account the module construct.

Software Metrics

Software metrics are used to determine how good a piece of software is according to a given criterion. The work of this thesis is concerned with trying to help maintenance programmers understand modules and module interconnections. If this work is combined with the proposed work on redefining the cohesion and coupling definitions, a solid base on which to create a set of software metrics on the inter-module connections will be established.

1.4 Thesis Structure

Chapters 2–3 introduce the subject matter of this thesis. Chapter 2 discusses some of the code analysis techniques from which ideas are borrowed in order to perform inter-module code analysis. Chapter 3 discusses the module construct and some of the system decomposition techniques that exist.

Chapters 4–8 discusses the graphs and graph operations that are used for inter-

module code analysis. In Chapter 4, the general properties of a graph showing the dependencies between entities are discussed. In Chapter 5, the notation used to manipulate graphs is explained. Chapters 6–8 describe three different forms of interconnection graphs that show dependencies between the entities. Chapter 9 discusses the use of the interconnection graphs in analysing a system and breaking up a module into smaller modules.

The relational database that is to be used for inter-module code analysis is described in Chapters 10–11. Chapter 10 describes the design of the relational database and Chapter 11 describes how this database can be used in inter-module code analysis.

Chapter 2

Code Analysis Techniques — An Overview

2.1 What is Code Analysis?

Code analysis is a generic term used to denote those programmer activities where the primary emphasis is on *examining* a piece of program code. Code analysis activities take on many different forms, such as: the program derivation and program proving work of Hoare [71], Dijkstra [52, 53], and Gries [64, 65]; the error and anomaly detection work of Fosdick and Osterweil [58, 112], Hennell [70], Howden [75], and Hartmann and Robson [66, 67]; and the program understanding work of Green [63], Shneiderman [140, 141], Johnson and Soloway [81, 82], Letovsky and Soloway [91, 93, 94].

Two important aspects of code analysis are: determining the dependencies between different entities and analysing the usage of entities. Some of the work in these areas that has influenced the research direction of this thesis will be described in the following sections. Section 2.2 describes data flow analysis, where the usage of variables is analysed in order to determine anomalous usage. Section 2.3 describes program slicing, where statements are extracted from a program segment because they affect the value of a designated variable. Section 2.4 describes the use of call graphs which gives a high level representation of a program's structure. Finally, section 2.5 describes program transformation systems, a means by which the structure of a system is improved.

2.2 Data Flow Analysis

Data flow analysis techniques examine a piece of program code in order to determine if there are any anomalous uses of variables within that code. Osterweil and Fosdick [112] give two rules on variable usage in terms of the actions that can be performed on variables. It is possible for three actions to be performed on a variable. These are:

`defined` — a value is stored in the variable,

`referenced` — the value stored in the variable is used,

`undefined` — it is impossible to state the value stored in the variable.

Consider for example the Pascal [80] program given in Figure 2.1. (The line numbers in the leftmost column are to aid the discussion of the program and do not form part of the actual program.)

```
1     PROGRAM ExampleProgram1 (Output);
2
3       VAR int1, int2: INTEGER;
4
5     BEGIN
6       int1:= 10;
7       int2:= int1 + 20;
8       WriteLn(int2: 5)
9     END. (* ExampleProgram1 *)
```

Figure 2.1: The Program used for Data Flow Analysis

Prior to the execution of the statement on line 6, the variables `int1` and `int2` are said to be **undefined** as no values have yet been given to them. When the statement on line 6 has been executed, the variable `int1` is now said to be **defined** as it has been assigned the value `10` by the assignment statement. The variable `int2` is still **undefined**. The statement on line 7 performs actions on two variables. Firstly the variable `int1` is **referenced**, as its value is needed in order to evaluate the expression,

```
int1 + 20
```

The result of this expression is then assigned to the variable `int2`. Hence the statement on line 7 causes `int1` to be **referenced** and `int2` to be **defined**. During the statement on line 8, `int2` is **referenced** because its value is used by the output routine `WriteLn`. Line 9 marks the end of the program so `int1` and `int2` become **undefined** as with the termination of the program's execution they lose their values.

The following rules on variable usage are given by Osterweil and Fosdick:

Rule 1: a `reference` must be preceded by a `define` without an intervening `undefine`.

Rule 2: a `define` must be followed by a `reference` without an intervening `define` or `undefine`.

Violation of either rule constitutes an anomalous variable usage.

Using the alphabet {`D`, `R`, `U`} to represent the actions `defined`, `referenced` and `undefined` respectively, the path expressions `PURQ`, `PDDQ` and `PDUQ`, where `P` and `Q` are arbitrary path expressions, represent the three anomalous paths that Fosdick and Osterweil [58] identified as being the violations of the above rules on variable usage.

`PURQ` — *Undefined Reference* — The value of a variable is used before the variable is given a value. This violates rule 1.

`PDDQ` — *Double Definition* — The value of a variable is changed without the old value being used. This violates rule 2.

`PDUQ` — *Lost Definition* — The value of a variable is undefined without the old value being used. This violates rule 2.

Much work has appeared in the computing literature on how to perform data flow analysis. Initially data flow analysis was performed statically, but it proved necessary to develop dynamic data flow analysis techniques because there are classes of data flow anomalies that can only be detected at run time. Some of the different data flow analysis techniques are described below.

2.2.1 Static Data Flow Analysis

Fosdick and Osterweil developed algorithms for performing static data flow analysis [57, 58]. These algorithms make use of some of the work on global flow optimisation, in particular, live-variable analysis and available expression analysis. (An explanation of live-variable analysis and available expression analysis is given by Aho et. al. [7].) Fosdick and Osterweil's data flow analysis technique involves the analysis of the control flow graph of a program where a node represents a program statement, and an edge represents a possible execution sequence. For example, the edge (n_α, n_β) denotes that there is a path through the program code such that the statement associated with the node n_β is executed after the statement associated with the node n_α. DAVE [112], was developed by Fosdick and Osterweil to analyse programs written in Fortran 66 [3]. Subsequent work by Jachner and Agarwal [79] corrected some of Fosdick and Osterweil's algorithms as these algorithms sometimes incorrectly detected data flow anomalies. Jachner and Agarwal also provide some new data flow analysis algorithms which have half the storage requirement and twice the execution speed of the Fosdick and Osterweil algorithms.

Initially the work on data flow analysis was aimed at Fortran. As Fortran does not require that variables be explicitly declared before they are used, it is possible for typographical errors in a program to go undetected. Data flow analysis can help detect this form of error. Since then data flow analysis has been usefully applied to other languages like C and PL/1 [4], and is used in test data generators.

Wilson and Osterweil [164] have developed Omega, a static data flow analysis tool for C [86] programs. C, unlike Fortran, allows dynamic variables. A dynamic variable is a variable whose associated storage is allocated and deallocated as the program is being executed. This dynamic property hinders the ability to use static

data flow analysis. In order to perform static data flow analysis on C programs, Wilson and Osterweil had to restrict the possible pointer operations to a small subset of those that C allows. Despite this restriction however, Wilson and Osterweil's data flow analysis technique has been shown to be of value in regression testing C programs. TRICS [131] (Testing by Regression and Integration of C Software) is a program testing tool developed by Raither and Osterweil that helps prepare test data for C programs. Sarraga developed PROBE [137], a static data flow analysis tool for PL/1 programs. As with Omega, PROBE can only work with programs written in a subset of the intended language. In order to overcome these restrictions, it is necessary to perform dynamic data flow analysis.

2.2.2 Dynamic Data Flow Analysis

The objectives of dynamic data flow analysis are the same as those of static data flow analysis, namely the detection and reporting of any data flow anomalies within a program. The difference is that with dynamic data flow analysis the anomalies are detected as the program is being executed.

Huang [76] introduced the fundamental aspects of dynamic data flow analysis. Every variable in a program must have associated with it a status variable, that records either the last action performed (`referenced`, `defined` or `undefined`) or the fact that the last action was anomalous. The value of a status variable is updated by the invocation of a status transfer function.

Huang used dynamic data flow analysis to tackle the problem of performing data flow analysis on arrays. Static data flow analysis cannot handle arrays properly as the appropriate array index is often only determinable at run time, e.g., with,

```
ReadInt(i);
Temp:= IntArray[i];
```

Huang's dynamic data flow analysis technique have been extended by several authors. Calliss and Cornelius [21] show how dynamic data flow analysis can be used with C programs. Status variables have been developed that can record status information on compound structures like `struct` and `union`, as well as on pointer variables. In order to do this, new status transfer functions were developed that cope with pointer and structure operations. Although Huang's techniques only finds one anomaly per variable, Calliss and Cornelius' data flow analysis technique allow more than one data flow anomaly to be found. Similar work by Chen and Poole [28] has also produced dynamic data flow analysis techniques for C programs.

2.2.3 Other Data Flow Analysis Work

With the static data flow analysis techniques mentioned in subsection 2.2.1, the result of performing data flow analysis on a routine must be known *before* the result of a call to that routine can be assessed. With recursive procedures this is not possible. However, Fairfield and Hennell [56] have developed static data flow analysis techniques that can cope with recursion.

Data flow analysis techniques have been developed for concurrent software by Taylor and Osterweil [149] and Osterweil et. al. [113]. At the heart of static data flow analysis are algorithms which operate on an annotated graphical representation of a program. With sequential programs, this graphical representation is the control flow graph (or flowgraph) whereas for concurrent software it is the *process*

augmented flowgraph [148]. A process augmented graph is formed by connecting the flowgraphs representing the individual processes. Data flow anomalies in concurrent software are divided into those anomalies that *must* occur and those that *may* occur depending on the state of one of the processes.

An alternative technique for performing data flow anomaly detection is given by Bergeretti and Carré [12] and Carré [27]. This technique is based on information-flow analysis. More information is gathered on variable usage than with data flow analysis and the number of anomalies than can be detected statically is increased. Information-flow analysis allows tests for ineffectiveness of statements and variables and `for` loop stability which usefully extend the class of anomalies that can be detected statically. The information-flow analysis techniques of Bergeretti and Carré have been incorporated in SPADE (the Southampton Program Analysis and Development Environment).

Data flow analysis can be used for both intra-procedural and inter-procedural code analysis. With static data flow analysis, the speed suffers greatly as the number of routines that have to be analysed increases. With dynamic data flow analysis tools the amount of output generated by a program run is potentially very large. These points indicate that, with interactive code analysis tools, data flow analysis is best suited to intra-procedural code analysis or to inter-procedural code analysis of small sections of a program.

2.3 Program Slicing

Program slicing is a form of program decomposition based on control flow and data flow analysis. The concept of the program slice was introduced by Weiser in [158, 159, 160]. A program slice, S, from a program, P, is a sequence of statements where the order of the statements in S is the same as in P. A program slice S, from a program P is obtained by *projecting* the statements from P, that conform to some slicing criterion. This will be denoted by,

$$P\overrightarrow{(C)}S$$

where C is the slicing criterion being employed. A slicing criterion is an ordered tuple of the form,

statement-range, vars-of-interest

For a particular slicing criterion, the value for *statement-range* is the range of statements over which a program is to be sliced; the value for *vars-of-interest* is some subset of variable identifiers that are visible in the given statement range. When a program is sliced in this way, the statements in the given range that do not *affect* the value of one of the chosen variables of interest are deleted to produce the program slice which contains all the statements in the chosen range that affect the chosen variables of interest.

Consider for example, the segment of a Pascal program given in Figure 2.2. Figure 2.3 gives three different slices for this program segment. Let P be the program segment in Figure 2.2. The program slice in Figure 2.3(a) represents the

```
 1      READ(X, Y);
 2      TOTAL:= 0.0;
 3      SUM:= 0.0;
 4      IF X <= 1
 5         THEN SUM:= Y
 6         ELSE BEGIN
 7                 READ(Z);
 8                 TOTAL:= X * Y
 9              END;
10      WRITE(TOTAL, SUM);
```

Figure 2.2: A Segment of a Pascal Program

```
READ(X, Y);
IF X <= 1
   THEN
   ELSE READ(Z);
```

(a)

⟨(1,10),{Z}⟩

```
READ(X, Y);
```

(b)

⟨(1,10),{X}⟩

```
READ(X, Y);
TOTAL:= 0.0;
IF X <= 1
   THEN
   ELSE TOTAL:= X * Y;
```

(c)

⟨(1,10),{TOTAL}⟩

Figure 2.3: Three Example of Program Slices

slicing operation,

$$\overrightarrow{P\langle(1, 10), \{Z\}\rangle}S_1$$

The input statement READ(X, Y) is included in the slice because the value of X affects whether the second input statement is executed or not, and it is this statement that affects the value of Z. The program slice in Figure 2.3(b) represents the slicing operation,

$$\overrightarrow{P\langle(1, 10), \{X\}\rangle}S_2$$

This time the variable of interest is X. The condition test on line 4 and the assignment statement on line 8 of the program segment P are not included in the program slice because although they *use* the value of X they do not *affect* its value. The program slice in Figure 2.3(c) represents the slicing operation,

$$\overrightarrow{P\langle(1, 10), \{TOTAL\}\rangle}S_3$$

It is possible for a slicing criterion to contain more than one variable of interest. Consider for example the slicing operation,

$$\overrightarrow{P\langle(1, 10), \{Z, TOTAL\}\rangle}S_4$$

This slicing is regarded as an amalgam of the slicing with respect to each of the variables. Thus,

$$S_4 \equiv S_1 \uplus S_3$$

where $\uplus$ denotes an amalgamation operator. This slice is shown in Figure 2.4. This program slice is an amalgam of the program slices given in Figure 2.3(a) and (c).

```
READ(X, Y);
TOTAL:= 0.0;
IF X <= 1
   THEN
   ELSE BEGIN
          READ(Z);
          TOTAL:= X * Y
        END
```

Figure 2.4: Program Slice Formed from the Criterion ⟨(1,10),{Z,TOTAL}⟩

Weiser implemented a program slicer for programs written in Algol-W. The implementation of this program slicer is described in [160]. It was used in experiments to determine if programmers use slices when debugging programs [158]. The experiments showed that there is evidence that this in fact true. This conforms to an analogous experiment conducted by Soloway and Erhlich [145] where they determined that programmers used program plans when analysing programs. Letovsky provides the following definition of program plans in [92]

> "...the cliches of programming, the familiar idioms and algorithms that make up a programmer's expertise".

Work on program slicers has been undertaken by other authors but this work has mainly been confined to making the slicing algorithms faster (for example Leung and Reghbati [95]). Ottenstein and Ottenstein [114] describe a graph structure called the program dependence graph, which allows programs to be sliced in linear time and the redundant statements on multi-statement lines are stripped out. (Weiser's program slicer makes use of statement line numbers so with multi-statement lines it is possible for redundant statements to be included in the program

slice.)

Lyle and Gallagher [100] show how program slicing can be of value to the modification and testing phases of software maintenance, by slicing out the portions of code that needs to be modified. The effect of a proposed modification can be determined by performing appropriate slices. Lyle and Gallagher also include output statements in a program slice, because although an output statement does not affect the value of a variable, it was found that programmers found the inclusion of output statements in a program slice helpful when trying to understand a program.

Ambras and O'Day [8] describe how in the MicroScope system, program slices are used in conjunction with execution histories (a log of control flow and data structure changes), in order to understand why certain events happened when a program was executed. For example, why was a variable set to a particular value, or why was a particular branch taken in the control flow.

Program slicing is strictly an intra-procedural code analysis technique. If a program slice is to be performed over a routine call then only the call statement is included in the program slice. In order to obtain the statement sequence from the called routine that affects the selected variables of interest, a separate program slicing operation must be performed on the routine. The intra-procedural nature of program slicing means that its main value is when a programmer wants to obtain a detailed picture of how a section of code works, and it is of limited value when global variables are being considered.

2.4 Call Graphs

The above code analysis techniques make use of control flow graphs in order to perform the desired analysis. With modern high level languages, there is another form of graph that can convey meaningful information about a program — the **call graph**. A call graph is a directed graph that represents the dynamic relations between routines. A node of a call graph denotes a routine and an edge denotes the calling of one routine by another. The direction of the edge indicates which routine is the caller and which routine is being called. Consider for example, an edge (n_λ, n_μ). This is an edge going from n_λ to n_μ and represents the calling of the routine denoted by n_μ by the routine denoted by n_λ. Techniques for constructing the call graph of programs have been described by Ryder [134] for Fortran 66 and Cooper [36] for Pascal.

The call graph forms the backbone of many code analysis techniques where inter-procedural analysis is to be performed. When trying to understand a program, it is important to be able to view a program from different levels of abstraction. The call graph provides a means of examining a program's structure at a higher level than the statement level, which is depicted by the control flow graph.

Shneiderman et. al. [142, 143] use call graph information in their program browser. This program browser is a hierarchical browser that has the objective of making the program's structure more visible to programmers. The programmer is presented with a piece of program text together with a list of callable routines. As the programmer moves through the program text the list of callable routines is automatically updated so that the list of callable routines is always in synchronisation with the program text being examined. Other uses of call graphs that have appeared in the computer literature include: Sengler's description of how call graph informa-

tion is used by programmers to divide a program into manageable segments [139], Khun and Holliss' automatic documentation system for PL/1 programs [89] and Ryder's incremental updates on software [135]. Ryder and Carroll [136] describe some incremental algorithms for analysing a call graph.

Some specialised forms of call graphs have been devised. Debnath and Bieman [45] use the generalised program graph as a model for the analysis of the interprocedural structure of a program. The generalised program graph is a form of call graph where information on parameter passing is also recorded.

2.5 Program Transformation Systems

Program transformation systems are systems that transform a program into a structurally different but logically equivalent program. These systems fall into two categories.

- Restructurers

  These systems manipulate the control flow graph representation of the program.

- Formal Transformations

  These systems manipulate a series of assertions that depict what the program is doing.

These two categories will be considered below.

2.5.1 Restructurers

Böhm and Jacopini [14] showed that it is possible to transform the control flow of a program so that the control flow is a composition of sequence, `repeat ... until`, and `if ... then ... else` structures. The idea of transforming the control flow of a program gained further momentum after Dijkstra's paper, "GOTO Statement Considered Harmful" [48], was published.

A clearer understanding of unstructuredness was obtained when Williams [161] examined the nature of unstructuredness in programs and identified the five basic structures that result in unstructured programs. These structures are:

- abnormal selection path
- loop with multiple exit points
- loop with multiple entry points
- overlapping loops
- parallel loops

Williams and Ossher [163] show how to detect these unstructured structures and to transform them into a structured form.

Several authors have investigated the problem of restructuring unstructured programs, e.g., Ashcroft and Manna [9], Baker [10], Cowell et. al. [40], Oulsnam [115], Prather [130] and Williams [162].

Some problems with automatic restructurers have been identified [20]. For example, the amount of code produced by a restructurer is usually greater than

the original program; and many of the restructuring algorithms make use of state variables which the restructurer adds to the program. These examples illustrate the main problem with restructurers however: they deal with the symptom of bad code and not the cause. Restructurers tidy up spaghetti code but do not tidy up the dubious logic that resulted in the writing of the bad code.

2.5.2 Formal Transformations

Formal transformation techniques approach the subject of program transformation differently. Instead of searching the control flow graph for a structure that is deemed bad and transforming it, formal transformation techniques first establish what the program is doing and then find an alternative and more acceptable coding.

Formal transformation techniques to derive programs have been known for several years, for example, Burstall and Darlington [19], Manna and Waldinger [101], Reddy [132] and Scherlis [138]. More recently some authors like Sneed and Jandrasic [144] and Ward [153] have used formal transformations to derive specifications from code. Ward has developed transformation techniques that are based on Dijkstra's weakest preconditions [52, 53] and Karp's infinitary logic language $L_{\omega_1\omega}$ [84]. These transformations are described in [154]. Ward's transformations are being semi-automated in the "Maintainer's Assistant" [23, 24, 109, 155].

Whereas the transformations used in restructurers are based on preserving the order of statement execution, duplicating code if needed, Ward's transformations are based on deriving the specification for a piece of code, and then obtain an alternative coding that satisfies this specification. As the driving criterion is not

to preserve the order of statement execution, programs derived as a result of using Ward's transformations can often be smaller than the original program as is demonstrated in [153].

Restructurers analyse the control flow graph of a section of code and ensure that the control flow graph consists of a combination of structures that are deemed desirable. As restructurers analyse and manipulate the control flow graph of a program they are primarily intra-procedural code analysis tools. Formal transformation systems work with assertions about the code, and they are therefore more interested in the semantics of the code rather than the structure. This lack of interest in the structure of the system makes formal transformation systems amenable to both intra-procedural and inter-procedural code analysis. Sundblad [147], Cornelius and Kirby [39] and Ward [156] demonstrate the use of transformation techniques on routines implementing the Ackermann function in order to achieve a more efficient implementation. This form of transformation is not possible with restructurers.

2.6 Summary

The work on data flow analysis and program slicing is concerned with determining information on how a program is to perform in terms of the dependencies between the variables. Data flow analysis is the analysis of variable usage within a program. By using data flow analysis, anomalous variable usage within a program can be detected and as a result some previously undiscovered program errors found. Data flow analysis is a code analysis technique that can be used for both intra-procedural and inter-procedural code analysis. Some of the data flow anal-

ysis techniques allow the dependencies between variables to be established. This is the cornerstone of the program slicing techniques. With program slicing, it is important to be able to determine which variables are affected by which variables. This then allows redundant statements to be excluded from program slices. Lyle and Gallagher [100] show how this provides a means of abstracting out different views of a system.

Program slicing is less generally useful than data flow analysis, but it introduces the idea of stripping out statements that do not affect a given variable. In this thesis, this idea will be generalised to allow program slicing to be performed at a higher level than the statement level.

The call graph depicts the calling dependencies that exist between routines, i.e., which routines are called by which routines. This graph however does not show the dependency that exists between routines because they use the same global variable, i.e., which routines are commonly coupled. This is important form of dependency that needs to be identified in many inter-procedural code analysis situations.

The call graph also gives an architectural view of a system. This gives a high level description of how a system has been decomposed into different processes. However, with modern programming languages the routine is no longer the only unit of modularity. Therefore new forms of graphs are needed to give a satisfactory representation of a system's architectural structure.

Program transformation systems analyse and manipulate representations of a program in order to improve the system. Most of the program transformation systems are aimed at improving the statement level structure of a program. Formal

transformation systems do this, but they also improve the routine level structure of a system. Most of the work in this area has concentrated on transforming recursive routines into a more efficient form, by either removing the recursion, or by introducing global variables.

The system structure is important in helping a programmer to understand a system; the more complex the structure the harder it is to understand. Some high level system transformation techniques will be described later in this thesis. These transformation techniques will group related entities together in a way that helps a programmer to understand a system.

Chapter 3

Modules and Modularisation

3.1 What is a Module?

3.1.1 Definition

The term module is used by different authors to denote different programming constructs. Originally the term applied to a routine, but since Parnas published, "On the Criteria to be used in Decomposing a System into Modules" [119], the term module has been used to denote a clustering construct. Wirth [168] describes a module as being,

> "...a set of procedures, data types and variables, where the programmer has precise control over the names that are imported from

and exported to the environment".

This description is aimed at the module construct in Modula, but it can form the basis of a more general description. Some languages like Ada and Modula-2 allow local modules to be declared. This means that a module can be declared within another module. This has implications on how the exporting of an entity is to be interpreted.

When a global module exports an entity, it is exported to the "environment". When a local module exports an entity, the entity is exported to the environment defined by the module containing the declaration of the local module. Therefore, a module is said to export an entity to the *surrounding environment.*

Other languages are not as restrictive as Modula in what classes of entities they allow a module to export. For example Ada allows local modules (called packages in Ada) to be exported. Therefore a module is said to export *entities*, rather than particular classes of entities. For the purpose of this thesis, the term "module" is defined as,

> *a named collection of entities, where the programmer has precise control over the entities that are imported from and exported to the surrounding environment.*

The module constructs of existing programming languages differ in the way that they are used. Some module constructs provide a specialised service, for example an abstract data type, while others are designed to be more generally useful. Despite this difference all module constructs provide:

1. an *abstraction mechanism*, and

2. a *protection mechanism.*

These two facilities will now be discussed.

3.1.2 Abstraction Mechanism

The term *abstraction mechanism* refers to a module construct providing at least two perspectives of the same program segment:

1. the **client view**, and

2. the **supplier view.**

```
DEFINITION MODULE Stack;

   TYPE StackType;

   PROCEDURE IsFull(st: StackType): BOOLEAN;
   PROCEDURE IsEmpty(st: StackType): BOOLEAN;
   PROCEDURE NumberOfElements(st: StackType): CARDINAL;
   PROCEDURE Create(VAR st: StackType);
   PROCEDURE Top(st: StackType; VAR element: INTEGER);
   PROCEDURE Pop(VAR st: StackType);
   PROCEDURE Push(element: INTEGER; VAR st: StackType);

END Stack.
```

Figure 3.1: An Example from a Language that Provides a Simple Client View

The client view of a module is the information that a client module is given about the public entities of a supplier module. The amount of information that is given to a user of a module varies from language to language. In languages like Ada and Modula-2 a simple client view is provided. With these languages the client view consists of a set of entity names together with a set of entity attributes.

For example, if a module were to provide a stack abstract data type, then the client view in Modula-2 might be programmed as shown in Figure 3.1. This client view shows the information that is needed in order to use the routines. This form of client view does not indicate what each of the routines does. A stack is typically depicted as being a data structure that adheres to the axioms given in Figure 3.2. Ideally the client view of a module construct should describe these axioms in some form. Some of the more advanced client views attempt to do this.

```
pop(push(element, st)) → st
top(push(element, st)) → element
is_empty(create())
¬is_empty(push(element, st))
```

Figure 3.2: Axioms Depicting a Stack

Figure 3.3 gives the client view for a stack module written in Eiffel. The **require** and **ensure** clauses are the pre and post conditions for the routines. Although they do not express the stack axioms, more information is provided on the effect of each of the routines. This extra information can be of value to programmers when they are looking at the services that a module provides.

The number of client views is normally one, but some languages allow for multiple client views. For example, Extended Pascal [2] has a module construct to

class interface *STACK*

exported features *num_of_elements, is_full, is_empty, top, pop, push*

feature specification

num_of_elements: INTEGER

is_full: BOOLEAN

is_empty: BOOLEAN

top: INTEGER
  require
    not *is_empty*

push(element: INTEGER)
  require
    not *is_full*
  ensure
    not *is_empty; top* = *element; num_of_elements* = old *num_of_elements* + *1*

pop
  require
    not *is_empty*
  ensure
    not *is_full; num_of_elements* = old *num_of_elements* − *1*

Figure 3.3: An Example of an Advanced Client View

provide several interfaces, each interface being a different client view; and Eiffel has a restrictive export clause which allows specific entities to be visible only to a set of named modules.

The supplier view of a module is the view that the implementor of the module has. This is the more detailed view of the module consisting of the particular algorithmic solutions employed together with knowledge about which other modules are needed in order for this module to function. A module can acquire entities from other modules in three ways:

1. *Importing*

   A module can import an entity from another module only if that entity is in the client view applicable to the importing module.

2. *Inheriting*

   For a module to inherit entities from another module the inheriting module is built up as an extension or specialisation of the **bequeathing module.** When this occurs, the inheriting module normally obtains all of the entities in the bequeathing module. Simula allows a module to bequeath entities selectively to a descendent module by hiding entities it does not want to bequeath with the **hidden** clause.

3. *Injecting*

   An entity is injected into a module if the entity is exported by a local module and implicitly imported by the module containing the local module.

3.1.3 Protection Mechanism

A module can help to control the visibility of an entity and thereby help to restrict the use of that entity to only those modules which need to know about it. In order to do this, a module construct provides a block structure with the following scope rules:

- an entity that is visible outside the block can only be used within the block if the module associated with the block specifically requests permission to do so via an import clause;
- an entity that is declared within the block can be made available to the surrounding environment by exporting it.

When a program is being developed by a team of programmers, it is important that each programmer can design and implement his section of the system in semi-isolation from the other programmers. All that the programmer needs to know is what facilities other programmers are expecting him to provide, and what he can assume will be provided to him by other programmers. Both of these points will have been determined at an earlier design stage.

The scope rules of the module construct help enforce this. The facilities that other programmers are expecting are given in the client view(s), all other facilities are private to the implementor of a module and are not available to other programmers. This helps reduce the number of programming tricks employed by programmers because they cannot make assumptions about how somebody else has implemented a solution. This protection also helps in restricting the affect of a change to a system.

Suppose, for example, a module implements a stack by using an array. As the system evolves it is determined that the array representation is too restrictive, so it is changed to a linked list representation. The affect of this form of modification is isolated to the module implementing the stack, because users of this module only ever access the data structure through the routines provided (cf. Figures 3.1 and 3.3).

The only occasion in which modification to a module should necessitate changes to other modules is either when a structural change is being made, for example, the addition or removal of an entity from a module's set of public entities; or when a public entity is changed so that the way the entity is used has been altered, e.g, changing the number or order of parameters for a routine.

3.2 The Various Forms of Module Constructs

Many new languages have been developed that contain a module construct, while many of the established languages have introduced a module construct as they have evolved. Module constructs in programming languages have tended to place the emphasis on either providing data abstraction or on providing general information hiding. This has led to the creation of two distinct forms of module constructs which are described in the following subsections.

3.2.1 Abstract Data Types

The importance of types in programming is explained by Hoare [72]. In a companion paper, Dahl and Hoare [43] introduce the concept of the abstract data type. With an abstract data type, a programmer views a type in terms of the operations that are applicable to variables of that type. To implement an abstract data type in most languages, existing types are used to construct a new type that will represent the abstract data type. This type is then used to declare parameters of the routine that will provide the services of the abstract data type. A programmer is hampered by the fact that precise thinking is possible only in terms of a small number of elements at a time. Hierarchical abstract data types allow a large data structure to be partitioned into manageable portions. Since this initial work, the use of abstract data types in programming has been explored by many authors.

Naphtali and Rich [110] describe the use of abstract data types in developing the software of a real-time embedded system. The abstract data types were used to hide hardware features from other programmers. Often the use of abstract data types is found to be beneficial to the development process, even though the actual language being used does not contain a construct that facilitates the implementation of abstract data types. Linden [96] demonstrates that if a system is structured using abstract data types as the basic unit of modularity, then the resulting system is easier to extend and modify. Embley and Woodfield [54, 55] give some cohesion and coupling measures to help assess the quality of the implementation of abstract data types, and Osterbye [111] proposes a new method of implementing abstract data types so that abstract data types can share certain operations.

As the virtues of programming in terms of abstract data types became more accepted, the object-oriented programming paradigm emerged. According to this

paradigm, software systems consist of a set of communicating abstract data types. A module construct was designed specifically to implement abstract data types. In such languages the module construct is called a class. This form of module has been adopted by the class based languages like Clu [98] and object-oriented programming languages like C++ [146], Eiffel Simula and Smalltalk-80* [62].

As a class is a type, some of the existing work on type theories is now being applied to classes. The language Fun was developed by Cardelli and Wegner [26]. It is based on the typed λ-calculus in order to be able to model and reason about the type structures in several programming languages. Danforth and Thompson [44] survey some of the existing type theories, examining the manner and extent to which these theories are able to represent the **objects** and object interactions that arise in object-oriented programming.

3.2.2 Module Constructs Providing General Information Hiding

The class is a module construct that allows a system to be partitioned in terms of a data structure. This form of partitioning is not always applicable however; it may be desirable to partition a system into groups of related entities that do not work on a common data structure. For example, a module could contain a set of trigonometric functions. In a case study of using general information hiding modules, van Kiet [152] states that the need for abstract data types is rare, and that in the Modula compiler he developed using general information hiding modules, there was only a need to export two abstract data types. General

*Smalltalk-80 is a trademark of ParcPlace Systems, Inc.

information hiding module constructs have been adopted by languages like Ada, Fortran 8X [102, 103], Modula-2, Modula-3 [25] and Oberon [171, 172].

Several taxonomies for this form of module construct have been proposed. Two of these taxonomies are of interest to this thesis. These are: the functional classification given by Booch [16, pages 228–9], and the classification according to the degree of information hiding given by Ross [133]. Both of these taxonomies are aimed at Ada's package construct, but they are general enough to apply to other languages, although Ross' taxonomy has to be expanded.

Booch's Taxonomy

With Booch's taxonomy, modules are classified according to possible applications. These applications are characterised by the kind of entities they export. Booch classifies modules as follows:

- *Named Collection of Declarations*

  The exported entities are constants, variables and types only. The supplier view is usually empty.

- *Groups of Related Program Units*

  The exported entities are routines and modules only. The supplier view contains no state variables.

- *Abstract Data Types*

  The exported entities are constants, types, and routines. The supplier view contains no state variables.

- *Abstract-State Machines*

  The exported entities are constants, variables, types, routines and modules. The supplier view contains at least one state variable.

Booch's view of an abstract data type is not so strict as the definition given earlier (page 36). His view is that a collection of types, routines and constants constitutes an abstract data type if the routines and constants *use* the types. With the classical definition, the types would have to be used as parameters.

Booch's taxonomy represents an idealised view of the use of modules. In practice however, a module is often a combination of these classifications. Such modules are referred to as **potpourri modules**. A potpourri module provides a collection a collection of disparate services to the system. As a result of this, many of the client modules of a potpourri module will have conflicting interpretations on the nature of the service being provided.

Ross' Taxonomy

With the taxonomy proposed by Ross, modules are classified according to the degree of visibility of a type defined by the module. Ross classifies modules as follows:

- *The Open Module*

  An open module is one in which the type is exported and its declaration is fully visible in the client view. This means that a client module has access to the internal structure of a variable declared to be of this type.

- *The Private Module*

  A private module is one in which the type name only is in the client view. Client modules cannot therefore make use of knowledge about the internal structure of variables declared to be of this type, *but* a client module can assume that assignments and tests for equality are meaningful operations.

- *The Limited Module*

  A limited module is similar to the private module, but this time the client module cannot perform assignment and tests for equality.

- *The Opaque Module*

  An opaque module is a specialisation of the limited module. With the limited module if a type declaration is modified then any client module has to be recompiled even if it has not been modified. This problem can be avoided if the type is implemented as a pointer.

- *The Closed Module*

  In a closed module, the type defined is not exported and is therefore not available to client modules. This form of module is sometimes referred to as an encapsulated data type.

With Modula-2 an opaque type can be used by a client module in tests for equality and assignment operations, but there are problems in interpreting this in Modula-2 as has been identified by Cornelius [38]. The problem of interpreting assignment and test for equality operations with opaque types makes it more meaningful to classify a module with an opaque type as an opaque module.

Ross' taxonomy does not take into account all possible type visibilities however. It is possible for a type to be part open and part closed. For example, in Oberon it is possible to have a type declaration of the form

```
T1 = RECORD
        x, y:  INTEGER
     END (* T1 *)
```

in the definition part, and in the associated implementation part the same type could have the declaration

```
T1 = RECORD
        x, y:  INTEGER; (* externally visible fields *)
        a, b:  INTEGER (* private fields *)
     END (* T1 *)
```

In this way T1 is part open and part closed. With Oberon, if a type has a closed part then hints to a compiler are needed in order to state a maximum size for the record (see [170]). Modifications to Oberon have been suggested by van Delft [151] that remove this problem.

Wirth uses this form of type with his concept of type extensions [170] where a data structure is constructed as an extension of another type. For example, consider the following Oberon type declaration

```
T2 = RECORD(T1)
        z:  REAL
     END (* T2 *)
```

This declares the type T2 to be an extension of the type T1, and so variables of type T2 would have fields a, b, x and y as well as field z.

In order to cater for languages like Oberon the following classification will be added to Ross' taxonomy:

- *The Ringent Module*

  A ringent module is one in which part of the type declaration is visible and part is private. The client view contains the visible part of the type and client modules have full access to this part of the type. The closed part of the type however is concealed from client modules, and so they do not have direct access to this part of the data structure.

3.3 System Decomposition into Modules

Techniques for decomposing systems into modules at the design stage provide the much needed method by which the complexity of a large system can be brought under the control of a small number of people. Hoare [73] relates how the entire Elliot 503 Mark II software project had to be abandoned because the complexity had been allowed to get beyond the intellectual grasp of the programmers. Some of the major design methods are described in the following subsections.

3.3.1 Functional Decomposition

This is known as a *top-down* or *outside-in* decomposition method. Two of the best known forms of functional decomposition are Stepwise Refinement [51, 165, 166, 167] and Structured System Design [46, 116, 174, 175]. Both of these

techniques are characterised by the description given by Wirth,

> "...the program is gradually developed in a sequence of *refinement steps*. In each step, one of several instructions of the given program is decomposed into more detailed instructions. This successive decomposition or refinement of specifications terminates when all instructions are expressed in terms of an underlying computer or programming language." [165]

By this method, a system is decomposed into a collection of routines.

Page-Jones [116] shows that it is possible, in a limited way, to use functional decomposition techniques with modules. The module construct that Page-Jones uses is the **information-cluster** (a term coined by Parnas [118]). An information-cluster is a set of routines that have exclusive right of access to a particular item or items of data. The decomposition technique that Page-Jones shows involves determining which routines require the use of common data and then refining the routines around this data.

Page-Jones' technique is of limited value to systems that are to be written in **module languages** as the underlying technique is still routine oriented and Page-Jones says information-clusters should be implemented as a single routine with a separate entry point corresponding to each routine the cluster is supposed to have.

Stepwise refinement methods describe a procedure that, if followed, will help in the production of good software. However, a problem with stepwise refinement methods is that they tend to fix at an early stage the implementation details of each data structure. This has an effect on the maintainability and reusability

of the software produced. Parnas [122] demonstrates this by comparing the prime program which Dijkstra develops using stepwise refinement ([51, pages 26–39]) with the version that he developed using information hiding.

3.3.2 Information Hiding

The idea of using information hiding as one of the criteria for decomposing a system into modules was first proposed by Parnas [119]. Before implementation can begin, several design decisions must be made so that the system designers can successfully say how the system is to be decomposed into modules. It is necessary to try to predict which features of the system are likely to change and which are unlikely to change. In this way system designers can minimise the effect of a software change that was anticipated. Korson and Vaishnavi [88] give empirical evidence to show that information hiding does aid program modifiability.

In a system decomposed with respect to the information hiding criterion a module is referred to as a "**responsibility assignment**" ([119]) and a "work assignment" ([125]) by Parnas. This dual meaning for a module highlights two of the characteristics of a module.

The term "responsibility assignment" shows that a module is regarded as providing a service in the system. The responsibility of a module should be determinable without having to understand the module's internal design. Instead the responsibility should be determinable through the client views that the module provides to the rest of the system [123, 128]. Therefore, before any implementation can proceed, three important design decisions have to be made:

1. identification of features that are likely and unlikely to change

2. identification of the services that are needed within a system

3. identification of the interfaces that services are to provide.

These last two points indicate why a module is said to represent a design decision. In this context Parnas describes the connections between modules as,

> "... the assumptions which modules makes about each other." [118]

The term "work assignment" shows that the subsequent design and implementation of that module is the responsibility of a programmer. The work of this programmer should be done in isolation from other programmers. In this way in subsequent maintenance work it should be possible to modify the internal structure of a module without affecting the behaviour of other modules. This cannot be done if a programmer has made use of knowledge about how another module functions that is not in the module's client views (as was pointed out in subsection 3.1.3). The module languages prevent this, but in languages like C and Fortran, where module constructs have to be simulated, it is possible for a programmer to violate a design decision in this way.

Case studies have appeared in the literature to show that system decomposition into modules using the information hiding is a realistic choice in non-toy programs. A compiler for the language Modula is described by van Kiet [152], and Parnas was part of a programming team that implemented the software for the A-7E aircraft [17, 29, 30, 31, 32, 69, 117, 127, 129].

Parnas et. al. [125, 126] used the A-7E aircraft software to demonstrate how information hiding accompanied by hierarchically-structured documentation can be

of benefit to the reuse of software. The following problems of software reuse were ameliorated by use of this combination of information hiding and hierarchically-structured documentation:

- the specification of the software is either non-existent or too ambiguous;
- the desired software already exists, but the location of the code within the system is not known;
- the software that can perform the desired task is too general and inefficient; and
- the cost of modifying existing software is more than the cost of writing new software.

An important part of a system design to aid reuse is the idea of a hierarchy [121] and in particular a hierarchy of **virtual machines**, where a virtual machine is a software extension to the underlying hardware of the computer. This form of programming was first illustrated by Dijkstra in the T.H.E. system [49]. Parnas [122, 124] demonstrates that this hierarchy of virtual machines aids in the production of program families, which in turn aids in software reuse and software modifications because the system has been designed for this.

With the growth in popularity of information hiding in software design and implementation, several software specification techniques have emerged to facilitate information hiding. Heninger [68] describes the specification technique that is used with the A-7E aircraft system. Parnas [120] showed how a module could be specified by means of advanced client views. Bartussek and Parnas [11] extend Parnas' earlier specification technique by using *traces* to make it more useful. Hoffman [74]

proposes a different technique which he claims is easier to teach than traces are. Middelburg [106] proposes a new specification language for information hiding, VVSL, which is based on VDM.

3.3.3 Object-Oriented Design

The term *object-oriented* originated from the work on the Smalltalk-80 programming system. Object-oriented design is defined by Gardner [59] as being information hiding supplemented with the assumption that the difficult design decisions are those that concern the implementation of an object and the operations upon it. Goldberg and Robson [62] describe the structure of Smalltalk-80. This is probably the best described object-oriented system although other examples can be found. See for example, Abbott [6], Cox [41], Gardner [59] and Meyer [104, 105].

Given a problem, how does a system designer produce an object-oriented design. One method has been devised by Abbott [6] which has been expanded upon by Booch [15, 16]. The Abbott and Booch methods derive an object-oriented design from a natural language description of the desired system. Abbott describes the process as consisting of three steps.

1. *Develop an informal strategy for the problem.*

   The informal strategy should state the problem solution on the same conceptual level as the problem itself. It should be expressed in problem domain terms.

2. *Formalise the informal strategy.*

   The second step is to formalise the solution by formalising its types, objects,

operators and control constructs.

3. *Segregate the solution into parts.*

   Finally, the solution is broken up into a collection of modules and routines.

The first step consists of describing the desired solution in a natural language form. The second step consists of analysing the informal description and then identifying the:

- types
- objects (program variables) of those types
- operators to be applied to those objects

that are needed. When the types, objects and operators have been identified, they are then organised into the control structure suggested by the informal strategy. The suggested technique for identifying types, objects and operators is by associating common nouns with types, proper nouns with objects, and verb properties/characteristics with operators. Although this process appears mechanical, knowledge of the problem domain is essential in order to be able to derive the types, objects and operators correctly.

Booch extended the method in order to derive the module structure of a system by adding the following steps:

- establish the visibility of each object in relation to other objects
- establish the interface of each object

The object-oriented design methods of Abbott and Booch are primarily of value to module languages like Ada and Modula-2 where the module is a construct aimed at encapsulating a set of logically related entities and providing a programmer with some control over the visibility of the entities. The Abbott and Booch design methods are not well suited to the object-oriented programming languages, where a module is a user defined type, and where inheritance is used to declare a module as an extension or specialisation of one or more other modules [157]. With the object-oriented programming languages, inheritance is an important characteristic in program construction which is not addressed by Abbott or Booch.

3.3.4 Module Interconnection Languages

De Remer and Kron put forward the maxim that,

> "...structuring a large collection of modules to form a system is a distinct and different intellectual activity from that of constructing the individuals modules." [47]

Programming-in-the-large is regarded as being a different activity to programming-in-the-small and different languages are needed for both of these forms of programming. To assist with programming-in-the-large De Remer an Kron introduce the concept of a *module interconnection language.*

A module interconnection language allows a programmer to express the overall program structure. This information can then be used by a compiler to ensure that programmers do not violate the structure of the system. By doing this a module interconnection language helps to enforce the design decisions about the structure

of a system by preventing programmers from using entities which they should not have access to.

De Remer and Kron give the following as the main objectives of a module interconnection language:

- A *project management tool* encouraging and recording the stepwise refinement of a system.
- A *design tool* for establishing the connections between modules.
- Provides a *means of communication* between programmer working on different modules.
- Provides a means of documenting a system structure.

These objectives were met in MIL75, the module interconnection language developed by De Remer and Kron.

Other module interconnection languages have also emerged: Thomas [150] developed a module interconnection language for language that support data abstraction, and Cooprider [37] developed a module interconnection language that incorporates some version control.

Each of these module interconnection languages are primarily aimed at supporting the design phase of software development. They ignore languages that contain a module construct aimed at providing a protection mechanism as exist in Ada and Modula-2. The existence of a module construct in a programming language reduces the need for a module interconnection language, as the module itself enforces many of the programming restrictions that the module interconnec-

tion languages do. With this form of programming language, the main value of a module interconnection language is at the design stage where programmers are explicitly told what facilities from another module they can use.

This thesis describes work on analysing the inter-module dependencies, and part of this analysis process requires analysing the dependencies that exist between the entities within a module. This information is not recorded in any of the module interconnection languages as they all confine themselves to recording information inter-module connections only. Therefore, the information on inter-module connections derived in this thesis is not recorded in a format that conforms to any of the module interconnection languages.

3.4 Summary

The module provides the programmer with an abstraction mechanism more powerful than the routine. Entities can be grouped together because they are logically related, and the visibility of these entities can be controlled by the programmer that is implementing the module. Some of the entities are concealed entirely from other modules, while others are made visible. With some languages it is possible for a module to indicate to which other modules the selected entities are being revealed, and some languages even allow the class of an entity to be concealed from other modules.

This control over the visibility of entities helps act as a protection mechanism, as the risk of unauthorised or accidental use of entities from other modules can be minimised. By this means, a module can help ensure the software integrity of a

module, as programmers can be prevented from making use of knowledge to which they are not entitled.

As languages that contain a module construct as part of the have emerged, two forms of module constructs have appeared: the class module construct, used to implement a new type; and the general purpose module construct, used to encapsulate entities. Together with the emergence of these languages, software design paradigms have also emerged that use these two module constructs as the basic unit of modularity for a system.

Chapter 4

Interconnection Graphs

4.1 Introduction

In order to be able to discuss inter-module connections clearly, it is important that a suitable structure be used to represent particular properties of a program. The graph has been found to be a suitable structure for other code analysis techniques, so it will be used in this thesis to record and describe the inter-module connections of a system.

4.2 Graph Terminology

Graph theory suffers from a lack of a standard terminology. It is therefore necessary to give a brief explanation of the terminology used in this thesis. Most of this terminology is taken from [87].

Formally a graph is represented by $G(\mathcal{N}, \mathcal{E})$, where $\mathcal{N}$ is the set of nodes $\{n_1, \ldots, n_\xi\}$ and $\mathcal{E}$ is the set of ordered pairs called edges, $\{(n_1, n_2), \ldots, (n_{\xi-1}, n_\xi)\}$. The number of nodes is represented by $\mid \mathcal{N} \mid$ and the number of edges by $\mid \mathcal{E} \mid$. For all graphs $\mid \mathcal{E} \mid \leq \mid \mathcal{N} \mid^2$, but for most interconnection graphs, $\mid \mathcal{E} \mid \leq \kappa \mid \mathcal{N} \mid$, where κ is a small integer constant.

The graph $G_s(\mathcal{N}_s, \mathcal{E}_s)$ is a *subgraph* of $G(\mathcal{N}, \mathcal{E})$ if the nodes in $\mathcal{N}_s$ are also in $\mathcal{N}$ (i.e., $\mathcal{N}_s \subseteq \mathcal{N}$), and the edges in $\mathcal{E}_s$ are also in $\mathcal{E}$ (i.e., $\mathcal{E}_s \subseteq \mathcal{E}$). This subgraph relationship will be represented by the symbol $\sqsubseteq$ (or by $\sqsubset$ when strict subgraphing is being represented). Hence

$$G_s(\mathcal{N}_s, \mathcal{E}_s) \sqsubseteq G(\mathcal{N}, \mathcal{E})$$

For the graph $G_{ss}(\mathcal{N}_{ss}, \mathcal{E}_{ss})$ to be a *strict subgraph* of $G(\mathcal{N}, \mathcal{E})$, all the nodes and edges of $G_{ss}(\mathcal{N}_{ss}, \mathcal{E}_{ss})$ must occur in $G(\mathcal{N}, \mathcal{E})$. In addition, $G(\mathcal{N}, \mathcal{E})$ must contain some nodes or edges that do not occur in $G_{ss}(\mathcal{N}_{ss}, \mathcal{E}_{ss})$. The strict subgraph relationship will be represented by the symbol $\sqsubset$. Hence

$$G_{ss}(\mathcal{N}_{ss}, \mathcal{E}_{ss}) \sqsubset G(\mathcal{N}, \mathcal{E})$$

Both the subgraph and the strict subgraph operators are described more fully in

Chapter 5. The graph $G_{ps}(\mathcal{N}_{ps}, \mathcal{E}_{ps})$ is a *proper subgraph* of $G(\mathcal{N}, \mathcal{E})$ if

$$G_{ps}(\mathcal{N}_{ps}, \mathcal{E}_{ps}) \sqsubset G(\mathcal{N}, \mathcal{E})$$

and if the graph $G_{ps}(\mathcal{N}_{ps}, \mathcal{E}_{ps})$ is *isolated* from all the other nodes in $G(\mathcal{N}, \mathcal{E})$. The subgraph $G_{ps}(\mathcal{N}_{ps}, \mathcal{E}_{ps})$ is said to be isolated in $G(\mathcal{N}, \mathcal{E})$ if there is no edge in $\mathcal{E}$ which has only the start-node or the stop-node in $\mathcal{N}_{ps}$, but not both.

An ordered pair (n_α, n_β) denotes an edge going from node n_α to node n_β. Two nodes are classed as being *adjacent* if there exists an edge connecting the two nodes. The *neighbours* of a node n_γ is the set of nodes adjacent to n_γ. A *path* is a sequence of nodes such that successive pairs of nodes are adjacent. The path $n_1, n_2, \ldots, n_\eta$ is a path of length $\eta - 1$ from n_1 to n_η. If in a given path each node is visited only once, then the path contains no cycles and the path is called a *simple path.* A graph is termed *connected* if there is a path between every pair of its nodes; otherwise the graph is termed *disjoint.*

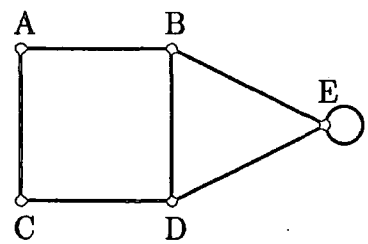


Figure 4.1: An Undirected Graph

Figure 4.1 shows an *undirected* graph, $G_u(\mathcal{N}_u, \mathcal{E}_u)$. With this form of graph no importance is placed on the order of the nodes of an edge. The only point of interest is whether or not there is an edge connecting two nodes. This implies that if there is an edge (n_α, n_β) then there is also an edge (n_β, n_α). Thus, for the graph

$G_u(\mathcal{N}_u, \mathcal{E}_u)$,

$$\mathcal{N}_u = \{A, B, C, D, E\}$$
$$\mathcal{E}_u = \{(A, B), (A, C), (B, A), (B, D), (B, E), (C, A), (C, D),$$
$$(D, B), (D, C), (D, E), (E, B), (E, D), (E, E)\}$$

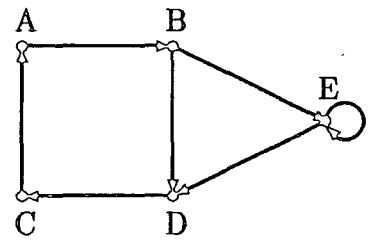


Figure 4.2: A Directed Graph

Figure 4.2 shows a *directed* graph, $G_d(\mathcal{N}_d, \mathcal{E}_d)$. With a directed graph the order of the nodes of an edge is important: the existence of the edge (n_α, n_β) cannot be used to infer the existence of an edge (n_β, n_α). This has implications when determining adjacent nodes. The edge (n_α, n_β) means that n_β is adjacent to n_α but does not mean that n_α is adjacent to n_β. With $G_d(\mathcal{N}_d, \mathcal{E}_d)$,

$$\mathcal{N}_d = \{A, B, C, D, E\}$$
$$\mathcal{E}_d = \{(A, B), (B, D), (B, E), (C, A), (D, C), (E, D), (E, E)\}$$

Since $\mathcal{N}_d = \mathcal{N}_u$ and $\mathcal{E}_d \subset \mathcal{E}_u$

$$G_d(\mathcal{N}_d, \mathcal{E}_d) \sqsubset G_u(\mathcal{N}_u, \mathcal{E}_u).$$

In general, it can be shown that a directed graph is a subgraph of its undirected version.

In many situations it is desirable to associate information with the nodes and

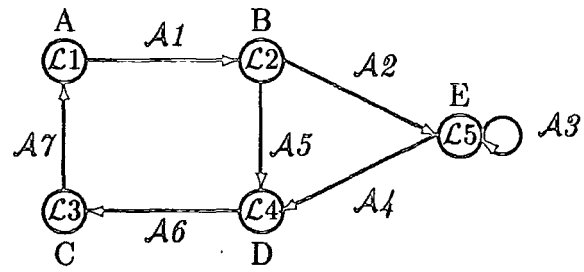


Figure 4.3: A Labelled Graph

edges of a graph. Figure 4.3 shows the graph, $G_l(\mathcal{N}_l, \mathcal{E}_l)$ which has information associated with both the nodes and the edges. When information is associated with the nodes of a graph it is called a *labelled* graph. The information associated with an edge can be numeric, as occurs with Markov chains, or the information can be about the nature of the dependency between the nodes, as occurs with semantic nets. When information is associated with an edge, it is possible for $\mathcal{E}$ to have multiple occurrences of an edge as the information associated with each of the occurrences is different.

4.3 Interconnection Graph

In the thesis, a graph called the **interconnection graph** is introduced. It is used to record some of the connections between the entities of a program. Most kinds of interconnection graphs are specialised to record particular dependencies. For example, a call graph records which entities are connected by an `invokes` dependency, whereas a control flow graph and a process augmented graph record which statements are connected by a `execution-can-follow` dependency. The specialised nature of these graphs is also reflected in the kind of entities the nodes

represent and the information that is associated with each node. A call graph records the dependencies between routine entities only, and with many call graphs the node merely records the name of the associated routine. With a control flow graph and a process augmented graph, each node is associated with a program statement rather than with a routine entity.

In order for a graph to be useful in inter-module code analysis the graph must be capable of recording several forms of dependencies between different entity classes.

```
TYPE ValueType = CARDINAL;

VAR  value: ValueType;

PROCEDURE Setvalue(valuePara: ValueType);
BEGIN
  value:= valuePara
END Setvalue;

PROCEDURE Getvalue(): ValueType;
BEGIN
  RETURN value
END Getvalue;

PROCEDURE Printvalue;
BEGIN
  WriteCard(value, 5)
END Printvalue;
```

Figure 4.4: A Modula-2 Fragment for Showing Entity Dependencies

Consider the Modula-2 code in Figure 4.4. Figure 4.5 gives an interconnection graph that shows the dependencies that exist between the entities. This graph records the interconnections between the five global entities that belong to three entity classes. The parameter `valuePara` from the routine `Setvalue` is not shown in the interconnection graph because it is not being considered as a

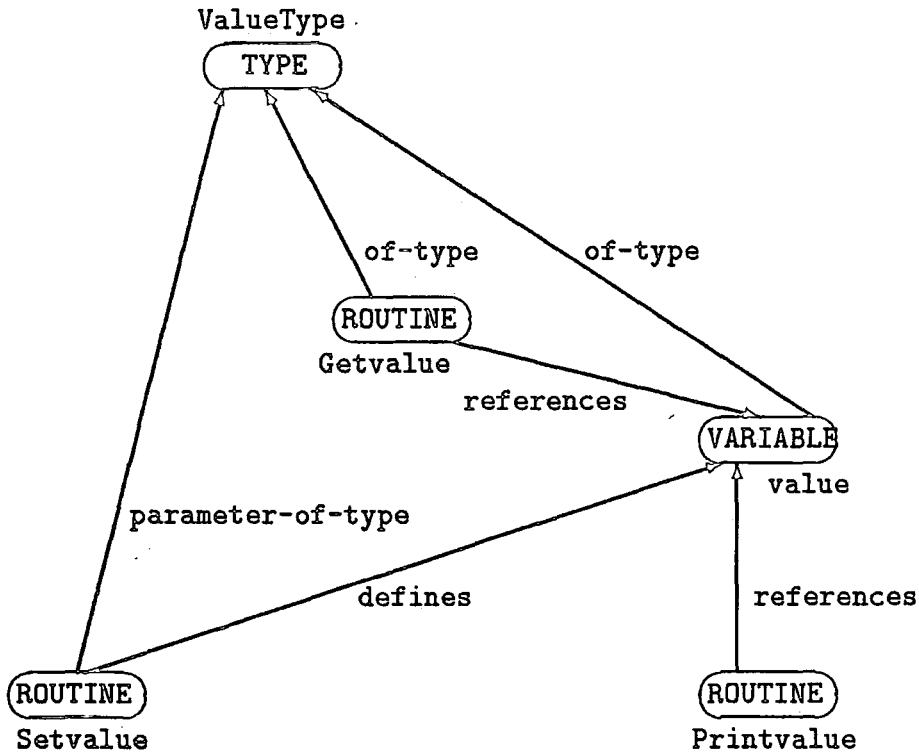


Figure 4.5: Interconnection Graph Representation of Figure 4.4

global entity. This is because `valuePara` cannot be used by any other global entities other than the routine that declares it. This makes it comparable to a local variable. The dependency that does appear in the interconnection graph is the dependency that shows the routine `Setvalue` to be dependent on the type `ValueType`, where `ValueType` is the type of the parameter `valuePara`. The edge (`Setvalue`, `ValueType`) shows that the routine `Setvalue` is dependent on the type `ValueType`, and the dependency `parameter-of-type` that is associated with the edge (`Setvalue`, `ValueType`) records that the type `ValueType` is used to declare a parameter of the routine `Setvalue`. Likewise, the edge (`Getvalue`, `ValueType`) shows that the routine `Getvalue` is dependent on the type `ValueType`, but this time the edge is associated with the dependency `of-type` which shows that the routine `Getvalue` is a function that returns a value of type `ValueType`. This form

of interconnection graph is called an entity-to-entity interconnection graph and is one of the three forms of interconnection graphs that will be used in this thesis to perform inter-module code analysis.

4.3.1 Existing Interconnection Graphs

Several graph notations have been proposed for depicting the interconnections between modules. Some of these graph structures will be discussed in this subsection.

Cunningham and Beck [42] introduce a notation for diagraming the message sending dialogue that takes place between objects participating in an object-oriented computation. The notation can be used to show: a module being created as an extension or specialisation of another module when **subclassing** is employed as the inheritance mechanism, and the calling dependency between the methods (routines) of the modules is also shown. Cunningham and Beck's notation can only record a subset of the dependencies that can exist in a program, e.g., the dependencies that involve the use of a state variable are not recorded, and it is not possible to record the existence of multiple inheritance or module instantiation.

Booch [16] and Buhr [18] introduce a notation that is aimed at languages like Ada. The notation records the inter-module connections in terms of the entities that are imported to and exported from modules. No distinction is made between a module importing an entity and a module acquiring an entity because the entity is explicitly exported to the module. This is a weakness when analysing programs written in languages like Eiffel that allow an entity to be exported to a set of named modules. Another problem with the notations of Booch and Buhr, is that

inheritance is not catered for. As a result, the object-oriented programming languages cannot be properly represented using these notations. Booch's notation suffers from an additional weakness in that the dependencies between the entities in a module cannot be shown. This means that these graphs cannot be used for intra-module code analysis.

Ince [77, 78] describes the use of a semantic net data structure to represent the dependencies between entities in a program. Each dependency is represented by a relation linking the two entities, i.e.,

$$entity1 \text{ — } relation \longrightarrow entity2$$

The direction of the relation indicates the order of the dependency.

Ince's approach to recording the dependencies between entities has several weaknesses when it come to inter-module coda analysis. A pattern match could only be performed with respect to the second entity of a relation. As a result, for every relation that can link two entities there must also be an inverse relation so that pattern matching can be performed on the first entity of a relation.

Consider for example, a relation of the form,

$$\texttt{R1} \text{ — } \texttt{invokes} \longrightarrow \texttt{R2}$$

In order to ascertain that `R2` is invoked by `R1` a relation of the form,

$$\texttt{R2} \text{ — } \texttt{invoked-by} \longrightarrow \texttt{R1}$$

is needed. The number of relations that Ince's semantic net can record is 20. In practice this would be 10 because of the need for the inverse relations to be recorded.

Another problem with Ince's representation is that it becomes difficult to analyse entities that are subject to several levels of nesting. Consider for example the following module declarations,

```
MODULE Level0;
   MODDULE Level1;
      MODDULE Level2;
         ...
      END Level2;
   END Level1;
END Level0.
```

In order to establish that the module `Level2` is within the module `Level0`, the semantic net representation would have to be traversed several times. When dealing with large systems the amount of graph traversing would almost certainly become prohibitively large, especially when it is necessary to establish the interchange of entities between modules.

To overcome these weaknesses, a new graph structure is developed that will combine characteristics of all of the interconnection graph structures described in this subsection.

4.3.2 Interconnection Graph used in this Thesis

Figure 4.6 describes the directed labelled graph structure which will be used in this thesis to perform inter-module code analysis. The graph structure is described in VDM [83]. This allows the graph operations developed in Chapter 5 to be formally specified in VDM. The graph is a labelled graph with information also associated with each of the edges. For the purpose of simplicity, the name of a node is assumed to be the name of the associated entity, and the node's label contains additional information about the associated entity. In particular, it records the entity class of the associated entity. This is necessary because it is not always possible to derive the entity class from the dependency connecting two nodes. For example, if a node `Ent` is connected to a node `T1` by an `of-type` dependency, then `Ent` could be either a type, a variable or a routine.

With module languages it is not possible to use the entity name alone to determine uniquely an entity in a system; extra information is needed. This extra information is provided by the name of the declaring module and the block number in which the entity is declared. Each block in a module is given a unique number. Figure 4.7 shows how blocks could be numbered in Modula-2. Note that the module `BlockNumbersExample` has two parts but only one block number. This is because `BlockNumbersExample` creates a single block. The entities that appear in the definition part of the module are automatically also visible in the associated implementation part. The physical separation of the definition and implementation parts is syntactic detail that does not affect the block numbering. Similar numbering can be done with other languages. The name of the declaring module and the block number in which an entity is declared are represented by the fields, *entity-source* and *entity-declaration-block*.

$Graph$ :: $nodes$: set of $Node$
$edges$: set of $Edge$

where

$inv\text{-}Graph(mk\text{-}Graph(nodes, edges)) \triangleq$
$\forall e \in edges \cdot (start\text{-}node(e) \in nodes \wedge stop\text{-}node(e) \in nodes)$

$Node$ :: $node\text{-}name$: $Name$
$node\text{-}label$: $Label$

$Edge$ :: $start\text{-}node$: $Node$
$stop\text{-}node$: $Node$
$dependency$: $Dependency$

$Name = \ldots$ /* The set of entity names */

$Dependency = \ldots$ /* The set of dependencies that the graph records */

$Label$ :: $entity\text{-}class$: $Class$
$entity\text{-}source$: $Source$
$entity\text{-}declaration\text{-}block$: $Block\text{-}Number$

$Class$ = `CONSTANT` | `TYPE` | `VARIABLE` | `ROUTINE` | `MODULE`

$Source = \ldots$ /* The set of module names in the system */

$Block\text{-}Number = \mathbb{N}$

Figure 4.6: Description of an Interconnection Graph in VDM

```
DEFINITION MODULE BlockNumbersExample;
(* This is block 0 (part a) *)
  :
END BlockNumbersExample.


IMPLEMENTATION MODULE BlockNumbersExample;
(* This is block 0 (part b) *)

  PROCEDURE P1;
  (* This is block 1 *)
    PROCEDURE P2;
    (* This is block 2 *)
    BEGIN
      ...
    END P2;
  BEGIN
    ...
  END P1;

  PROCEDURE P3;
  (* This is block 3 *)
  BEGIN
    ...
  END P3;

BEGIN
  ...
END BlockNumbersExample.
```

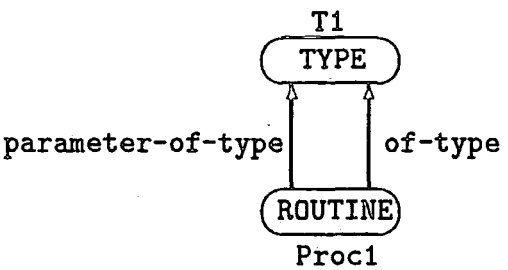
Figure 4.7: Example of Block Numbers in a Module

The *entity-class* field is used to record the class of an entity. When the entity is a procedure or function, the term `ROUTINE` is stored in the *entity-class* field. The distinction between a procedure and a function is not important to the code analysis work of this thesis.

An edge in the inter-module connection graph records the dependency between two entities. Since it is possible that two entities may have more than one dependency, two nodes may have multiple edges connecting them because each edge records a different dependency. Consider for example the following declarations

```
TYPE T1 = ...
PROCEDURE Proc1(para:  T1):  T1;
    ⋮
```

The corresponding section in an inter-module connection graph would be



The `parameter-of-type` dependency is because `Proc1` declares a formal parameter of type `T1`. The `of-type` dependency is because the routine is a function that returns a value of type `T1`.

4.4 Three Forms of Interconnection Graphs

An interconnection graph records all the dependencies between all of the components of a system, irrespective of whether they are high level entities like modules, or low level entities like variables. For the purposes of inter-module code analysis, it is useful to partition the interconnection graph for a system into the following three specialised forms of interconnection graphs.

1. The module-to-module interconnection graph

   Shows the dependencies between the modules that comprise a system. This graph will be discussed in Chapter 6.

2. The entity-to-module interconnection graph

   Shows the dependencies between modules, in terms of the entities that modules provide each other. This graph will be discussed in Chapter 7.

3. The entity-to-entity interconnection graph

   Shows the dependencies between the entities of a module. This graph will be discussed in Chapter 8.

Each of these interconnection graphs comply with the VDM description given in Figure 4.6.

These three kinds of graphs can be extracted from a general interconnection graph that shows how all of the entities within a system are dependent. The module-to-module interconnection graph, entity-to-module interconnection graph and the various entity-to-entity interconnection graphs do not contain all the information that is contained in a general interconnection graph. This is because none of the graphs record how a module uses entities acquired from other modules. This

information is important for detailed code analysis work, but based upon observations it was not found to be needed for the inter-module code analysis work of this thesis.

Chapter 5

Graph Operations

The proposed inter-module code analysis techniques make extensive use of analysing different graphical representations of a program. It is therefore necessary to introduce a notation that will help discuss the interplay between different graphs. This chapter explains the graph operations that will be performed and introduces the notation that will be used. Section 5.1 describes the subgraph operation that was briefly introduced in Chapter 4. Section 5.2 describes graph union operations that provide a means of combining two or more graphs. Section 5.3 describes graph intersection operations that provide a means of determining the features shared by two or more graphs. Finally, section 5.4 describes some graph slicing operations that allow a graph satisfying some specified constraints to be extracted from a given graph. Appendix B serves as a reference guide to the notation being introduced.

The graph operations will be described using the specification language of VDM. This allows the operations to be described in a formal way, and hence enables proofs

to be performed on particular operations. The predicate logic and set theory upon which VDM is based ensures the soundness and completeness of these proofs.

5.1 Subgraph Operators

Consider again the subgraph relation, $\sqsubseteq$, that was introduced on page 54. A VDM specification for this operation is,

$$\_ \sqsubseteq \_ : \mathit{Graph} \times \mathit{Graph} \to \mathbb{B}$$

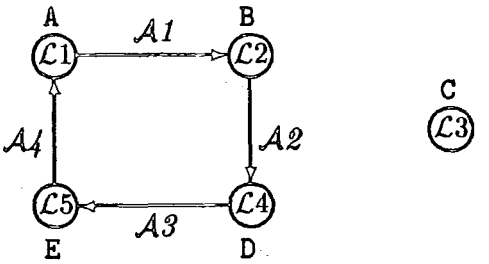
$$\mathit{mk\text{-}Graph}(\mathit{nodes1}, \mathit{edges1}) \sqsubseteq \mathit{mk\text{-}Graph}(\mathit{nodes2}, \mathit{edges2}) \triangleq$$

$$(\mathit{nodes1} \subseteq \mathit{nodes2}) \wedge (\mathit{edges1} \subseteq \mathit{edges2})$$

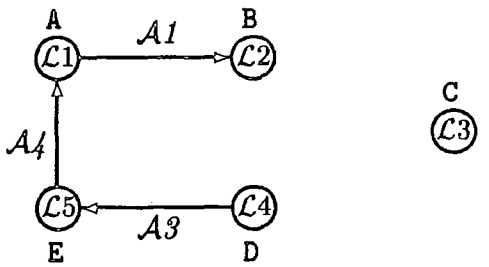
The signature for this relation indicates that $\sqsubseteq$ is a binary relation that works with two graphs, i.e., the subgraph operator is used as follows,

$$G_1(\mathcal{N}_1, \mathcal{E}_1) \sqsubseteq G_2(\mathcal{N}_2, \mathcal{E}_2)$$

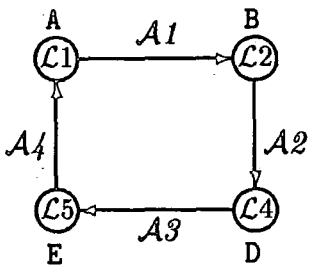
The subgraph specification indicates that for $G_1(\mathcal{N}_1, \mathcal{E}_1)$ to be a subgraph of $G_2(\mathcal{N}_2, \mathcal{E}_2)$ all the nodes and edges of $G_1(\mathcal{N}_1, \mathcal{E}_1)$ must exist in $G_2(\mathcal{N}_2, \mathcal{E}_2)$. $G_2(\mathcal{N}_2, \mathcal{E}_2)$ need not have any extra nodes or edges, i.e., the graphs $G_1(\mathcal{N}_1, \mathcal{E}_1)$ and $G_2(\mathcal{N}_2, \mathcal{E}_2)$ can be equal.



(a)



(b)



(c)

Figure 5.1: Some Examples to Illustrate Subgraphs

The strict subgraphing operator $\sqsubset$ can be defined by,

$$\_ \sqsubset \_ : Graph \times Graph \rightarrow \mathbb{B}$$

$$mk\text{-}Graph(nodes1, edges1) \sqsubset mk\text{-}Graph(nodes2, edges2) \triangleq$$

$$((nodes1 \subseteq nodes2) \wedge (edges1 \subset edges2)) \vee$$

$$((nodes1 \subset nodes2) \wedge (edges1 \subseteq edges2))$$

The signature of the strict subgraph relation indicates that its usage is similar to the first subgraph relation, e.g.,

$$G_1(\mathcal{N}_1, \mathcal{E}_1) \sqsubset G_2(\mathcal{N}_2, \mathcal{E}_2)$$

The strict subgraph relation is more complex than the first subgraph relation. All the nodes and edges of $G_1(\mathcal{N}_1, \mathcal{E}_1)$ must be contained in $G_2(\mathcal{N}_2, \mathcal{E}_2)$, and $G_1(\mathcal{N}_1, \mathcal{E}_1)$ can have either all the nodes or all the edges of $G_2(\mathcal{N}_2, \mathcal{E}_2)$ but not both.

Consider for example the three graphs in Figure 5.1. Let the graph in Figure 5.1(a) be the graph $G_2(\mathcal{N}_2, \mathcal{E}_2)$. Either of the graphs in Figure 5.1(b) or Figure 5.1(c) can be $G_1(\mathcal{N}_1, \mathcal{E}_1)$ and satisfy the relation,

$$G_1(\mathcal{N}_1, \mathcal{E}_1) \sqsubset G_2(\mathcal{N}_2, \mathcal{E}_2)$$

because with Figure 5.1(b) $\mathcal{N}_1 = \mathcal{N}_2$ and $\mathcal{E}_1 \subset \mathcal{E}_2$, and with Figure 5.1(c) $\mathcal{N}_1 \subset \mathcal{N}_2$ and $\mathcal{E}_1 = \mathcal{E}_2$.

The supergraph operators $\sqsupseteq$ and $\sqsupset$ can be defined in a similar way.

5.2 Graph Union

Graph union is the process of creating a graph as a combination of existing graphs. There are two forms of graph union: *simple* graph union and *distributed* graph union. These will be considered in turn.

5.2.1 Simple Graph Union

Simple graph union is the creation of a graph that contains all the nodes and edges from two given graphs. This operation is denoted by,

$$G_1(\mathcal{N}_1, \mathcal{E}_1) \sqcup G_2(\mathcal{N}_2, \mathcal{E}_2)$$

and can be specified as follows:

$\_ \sqcup \_ : Graph \times Graph \rightarrow Graph$

$mk\text{-}Graph(nodes1, edges1) \sqcup mk\text{-}Graph(nodes2, edges2) \triangleq$

$mk\text{-}Graph((nodes1 \cup nodes2), (edges1 \cup edges2))$

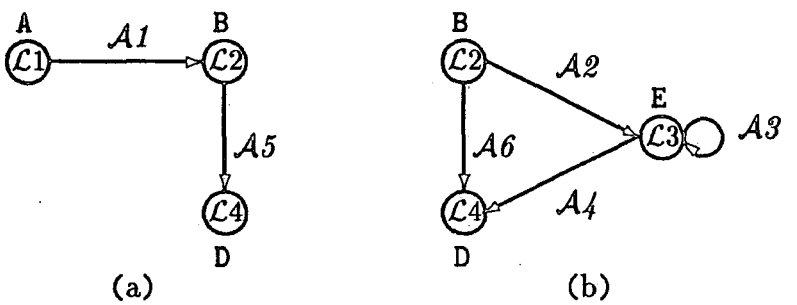


Figure 5.2: Two Labelled Graphs

Problems appear to arise if the two graphs concerned have nodes and edges in common. However, because of the property of set union upon which the graph

union operator is based, these are not in fact problems. Consider the graph $G_a(\mathcal{N}_a, \mathcal{E}_a)$ in Figure 5.2(a) and $G_b(\mathcal{N}_b, \mathcal{E}_b)$ in Figure 5.2(b). These two graphs have the nodes B and D in common. The result of the operation,

$$G_a(\mathcal{N}_a, \mathcal{E}_a) \sqcup G_b(\mathcal{N}_b, \mathcal{E}_b)$$

is given in Figure 5.3.

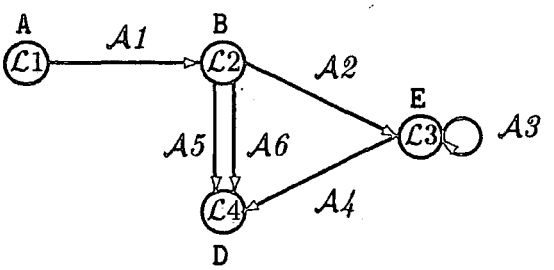


Figure 5.3: Union of the Graphs in Figure 5.2

Consider first the problem of node duplication. The property of set union has ensured that only one occurrence of each of the nodes B and D appears in the new graph. A node is only the same as a node in another graph, if they have the same name and label values. If two nodes have the same name but different label values then they are different nodes and therefore must both appear in the resulting graph. Figure 5.4 gives the result of the operation,

$$G_a(\mathcal{N}_a, \mathcal{E}_a) \sqcup G_b(\mathcal{N}_b, \mathcal{E}_b)$$

where the label for node B in $G_b(\mathcal{N}_b, \mathcal{E}_b)$ has the value $\mathcal{L}5$ (rather than $\mathcal{L}2$) and is therefore different to the node B in $G_a(\mathcal{N}_a, \mathcal{E}_a)$.

This ability to have two or more nodes with the same name in the same graph is important in inter-module code analysis. In a module language an entity is

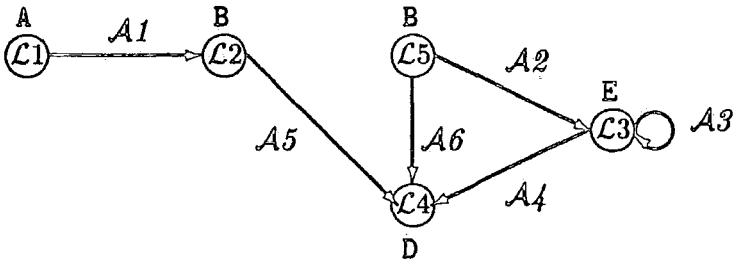


Figure 5.4: Graph Union for Two Nodes with the same Names

uniquely identified by combining its name, the name of the module in which it is declared, the block number in which it is declared, and if overloading of identifiers is allowed the entity class as well. A graph showing the dependencies between entities in a system implemented in one of the module languages will have to be able to distinguish between different entities with the same name. It is for this reason that each entity not only has a name, but a label which is used to record this additional information (see Figure 4.6 on page 64). In this way, the nodes for different entities with the same name can be distinguished in an interconnection graph.

Consider now the problem of edge duplication. The graph in Figure 5.3 has two edges (B, D). This is because the edge (B, D) in $G_a(\mathcal{N}_a, \mathcal{E}_a)$ has the attribute $\mathcal{A}5$ associated with it, while the edge (B, D) in $G_b(\mathcal{N}_b, \mathcal{E}_b)$ has the attribute $\mathcal{A}6$ associated with it. These different attributes mean that the two edges (B, D) are different edges and therefore the graph resulting from the operation,

$$G_a(\mathcal{N}_a, \mathcal{E}_a) \sqcup G_b(\mathcal{N}_b, \mathcal{E}_b)$$

contains the two edges. If the attributes had been the same, then the resulting graph would have had only one occurrence of the edge (B, D).

5.2.2 Distributed Graph Union

The distributed graph union operation builds on the simple graph union operation. Distributed graph union is denoted by the notation,

$$\bigsqcup\{G_1(\mathcal{N}_1, \mathcal{E}_1), \ldots, G_n(\mathcal{N}_n, \mathcal{E}_n)\}$$

This is equivalent to n − 1 applications of the simple graph union operation, i.e.,

$$\bigsqcup\{G_1(\mathcal{N}_1, \mathcal{E}_1), \ldots, G_n(\mathcal{N}_n, \mathcal{E}_n)\} \equiv G_1(\mathcal{N}_1, \mathcal{E}_1) \sqcup \ldots \sqcup G_n(\mathcal{N}_n, \mathcal{E}_n)$$

This operation has the following specification,

$\bigsqcup$: set of *Graph* → *Graph*

$\bigsqcup$*graphset* $\triangleq$

if *graphset* = { }

then *mk-Graph*({ }, { })

else let $g \in$ *graphset* in

$\bigsqcup\{$*graphset* − $\{g\}\} \sqcup g$

5.3 Graph Intersection

Graph intersection is the process of creating a graph in terms of characteristics that are common to two or more graphs. Just as with graph union, graph intersection has a simple form for two graphs, and a distributed form for n graphs (where n ≥ 0). Graph intersection can be further categorised as *strict* or *full* depending

upon which criterion is used to add edges to the resulting graph. With both categories of graph intersection, a node is added to the resulting graph only if the node is common to all the graphs involved in the intersection operation.

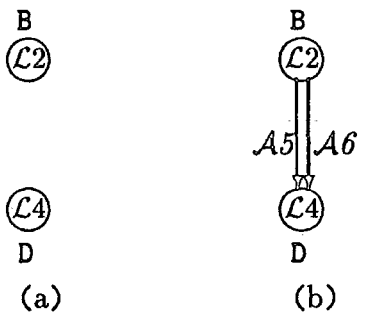


Figure 5.5: The Results of Two Graph Intersection Operations

5.3.1 Simple Graph Intersection

Strict graph intersection employs the criterion that an edge is included in the resulting graph only if that edge is common to all the graphs involved in the intersection operation. The simple form of strict graph intersection will be denoted by the symbol $\sqcap$ and is used as follows,

$$G_1(\mathcal{N}_1, \mathcal{E}_1) \sqcap G_2(\mathcal{N}_2, \mathcal{E}_2)$$

The graph formed by this operation contains only the nodes and edges that are common to both graphs. If this operation is applied to $G_a(\mathcal{N}_a, \mathcal{E}_a)$ and $G_b(\mathcal{N}_b, \mathcal{E}_b)$ (the graphs in Figure 5.2), then the resulting graph is given in Figure 5.5(a). Only the nodes B and D appear in this graph. Neither of the (B, D) edges appear because they do not occur in both graphs. Strict simple graph intersection can be specified as follows,

$\_ \sqcap \_ : Graph \times Graph \rightarrow Graph$

$mk\text{-}Graph(nodes1, edges1) \sqcap mk\text{-}Graph(nodes2, edges2) \triangleq$

let $nodeset = nodes1 \cap nodes2$ in

let $edgeset = edges1 \cap edges2$ in

$mk\text{-}Graph(nodeset, edgeset)$

Full graph intersection employs the criterion that an edge is included in the resulting graph if that edge forms part of at least one of the graphs involved in the intersection operation, and the stop-node and start-node of the edge are in the set of common nodes. The simple form of full graph intersection will be denoted by the symbol $\sqcap_+$ and is used as follows,

$$G_1(\mathcal{N}_1, \mathcal{E}_1) \sqcap_+ G_2(\mathcal{N}_2, \mathcal{E}_2)$$

The graph formed by this operation contains the nodes that the two graphs have in common together with all the edges from either of the two graphs that connect the common nodes. The result of applying this operation to $G_a(\mathcal{N}_a, \mathcal{E}_a)$ and $G_b(\mathcal{N}_b, \mathcal{E}_b)$ is given in Figure 5.5(b). The nodes B and D appear in this graph as do both the (B, D) edges, because each edge appears in at least one of the graphs involved in the operation. Full simple graph intersection can be specified as follows,

$\_ \sqcap_+ \_ : Graph \times Graph \rightarrow Graph$

$mk\text{-}Graph(nodes1, edges1) \sqcap_+ mk\text{-}Graph(nodes2, edges2) \triangleq$

let $nodeset = nodes1 \cap nodes2$ in

let $edgeset = \{edge \mid (edge \in (edges1 \cup edges2) \wedge$

$start\text{-}node(edge) \in nodeset \wedge$

$stop\text{-}node(edge) \in nodeset)\}$ in

$mk\text{-}Graph(nodeset, edgeset)$

5.3.2 Distributed Graph Intersection

The distributed graph intersection operations are built up in terms of the corresponding simple graph intersections in a similar way to distributed graph union. The distributed form of strict graph intersection,

$$\sqcap\{\mathrm{G}_1(\mathcal{N}_1, \mathcal{E}_1), \ldots, \mathrm{G}_n(\mathcal{N}_n, \mathcal{E}_n)\}$$

is equivalent to,

$$\mathrm{G}_1(\mathcal{N}_1, \mathcal{E}_1) \sqcap \ldots \sqcap \mathrm{G}_n(\mathcal{N}_n, \mathcal{E}_n)$$

and it can be specified as follows,

$\sqcap$: set of *Graph* $\rightarrow$ *Graph*

$\sqcap$*graphset* $\triangleq$

if *graphset* $= \{\}$

then *mk-Graph*$(\{\}, \{\})$

else let $g \in$ *graphset* in

$\sqcap\{$*graphset* $- \{g\}\} \sqcap g$

Similarly the distributed form of full graph intersection will be denoted by the notation,

$$\sqcap_+\{\mathrm{G}_1(\mathcal{N}_1, \mathcal{E}_1), \ldots, \mathrm{G}_n(\mathcal{N}_n, \mathcal{E}_n)\}$$

and is equivalent to,

$$\mathrm{G}_1(\mathcal{N}_1, \mathcal{E}_1) \sqcap_+ \ldots \sqcap_+ \mathrm{G}_n(\mathcal{N}_n, \mathcal{E}_n)$$

and it can be specified as follows,

$\sqcap_+$: set of *Graph* $\rightarrow$ *Graph*

$\sqcap_+ graphset \triangleq$

if $graphset = \{\}$

then $mk\text{-}Graph(\{\},\{\})$

else let $g \in graphset$ in

$\sqcap_+ \{graphset - \{g\}\} \sqcap_+ g$

5.4 Graph Slicing

The graph union and graph intersection operations provide a means of creating a new graph from two or more graphs. The process of creating a new graph by extracting a subgraph from a given graph according to some criterion is called *graph slicing.* In this thesis two forms of graph slicing are proposed:

1. δ-slicing, and
2. $\alpha\beta$-slicing

δ-slicing slices a graph with respect to the dependency that an edge represents, whereas $\alpha\beta$-slicing slices a graph with respect to the nodes. Each of these forms of graph slicing will be discussed in the following subsections.

5.4.1 δ-Slicing

δ-slicing is a form of graph slicing where the dependency represented by an edge is used to determine which edges from the original graph appear in the resulting

graph. A δ-slice operation is denoted as follows,

$$\delta(\mathrm{G}(\mathcal{N}, \mathcal{E}), \mathcal{C})$$

where $\mathcal{C}$ is the slicing criterion being applied to $\mathrm{G}(\mathcal{N}, \mathcal{E})$. The slicing criterion for δ-slicing is a set of dependencies that can appear in the resulting graph. The nodes that appear in the graph resulting from a δ-slicing operation, are those nodes from the given graph, that are connected by one of the named dependencies. In order for an isolated node to appear in the resulting graph, the special dependency `$ISOLATED$` is introduced.

The δ-slicing operation can be specified as follows,

Delta-Criterion :: *dependency-set* : set of *Dependency*

δ : *Graph* $\times$ *Delta-Criterion* $\rightarrow$ *Graph*

$\delta(graph, delta\text{-}criterion) \triangleq$

let *edgeset* $= \{e \mid e \in edges(graph) \wedge$

$dependency(e) \in dependency\text{-}set(delta\text{-}criterion)\}$ in

let *nodeset* $= \{n \mid n \in nodes(graph) \wedge$

$\exists e \in edgeset \cdot n = start\text{-}node(e) \vee$

$n = stop\text{-}node(e)\}$ in

if `$ISOLATED$` $\in$ *dependency-set*(*delta-criterion*)

then *mk-Graph*(*nodeset* $\cup$ *get-isolated-nodes*(*graph*), *edgeset*)

else *mk-Graph*(*nodeset*, *edgeset*)

where the function *get-isolated-nodes* has the following specification,

$$get\text{-}isolated\text{-}nodes : Graph \rightarrow \text{set of } Node$$

$$get\text{-}isolated\text{-}nodes(mk\text{-}Graph(nodes, edges)) \triangleq$$
$$\{n \mid n \in nodes \wedge \forall e \in edges \cdot n \neq start\text{-}node(e) \wedge$$
$$n \neq stop\text{-}node(e)\}$$

```
TYPE T1 = INTEGER;
     T2 = CHAR;
     T3 = RECORD
            F1: T1;
            F2: T2
          END; (* T3 *)

VAR  V1: T3;
     V2: INTEGER;

PROCEDURE P1;
  VAR LocalVar: T1;
BEGIN
   ...
   V1.F1:= 10;
   ...
END P1;

PROCEDURE P2;
BEGIN
  WriteString("Hello World")
END P2;
```

Figure 5.6: Program Segment to be Used for Graph Slicing

We now consider an example. Figure 5.6 gives a segment of program code and Figure 5.7 gives an interconnection graph, $G_e(\mathcal{N}_e, \mathcal{E}_e)$ that records the dependencies present. This is an example of an entity-to-entity graph. This form of interconnection graph will be discussed in Chapter 8. This graph has two isolated nodes V2 and P2 and shows three forms of dependencies between the entities,

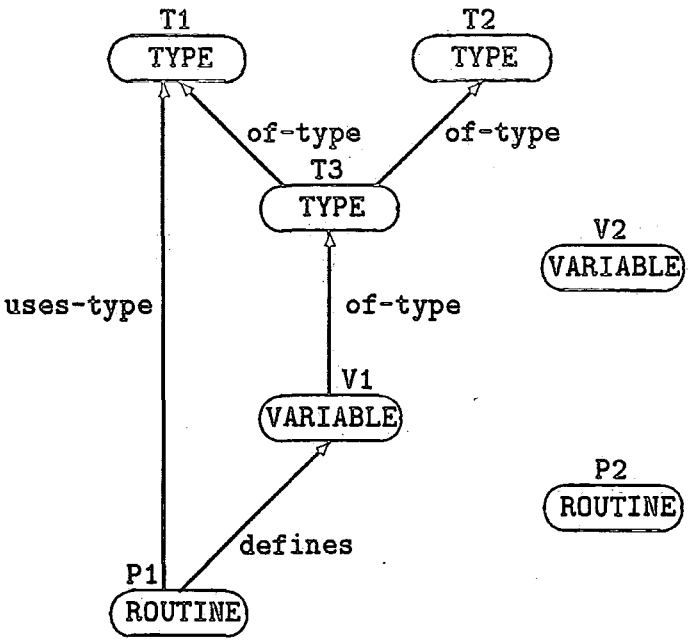


Figure 5.7: Entity-to-Entity Graph for Figure 5.6

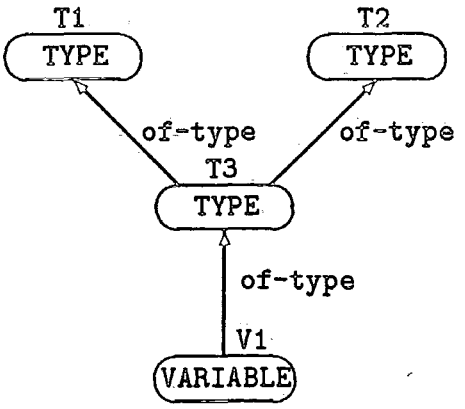
namely: `defines`, `of-type` and `uses-type`. Figure 5.8 gives the result of applying three δ-slicing operations on $G_e(\mathcal{N}_e, \mathcal{E}_e)$.

Figure 5.8(a) gives the result of slicing $G_e(\mathcal{N}_e, \mathcal{E}_e)$ with respect to the dependency `of-type`. The node `P1` is connected to the nodes `T1` and `V1` in $G_e(\mathcal{N}_e, \mathcal{E}_e)$ but `P1` does not appear in the graph resulting from,

$$\delta(G_e(\mathcal{N}_e, \mathcal{E}_e), \{\texttt{of-type}\})$$

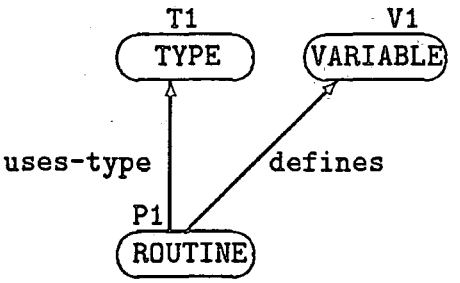
because neither of the dependencies between the node `P1` and the nodes `T1` and `V1` is the `of-type` dependency.

Figure 5.8(b) gives the result of slicing $G_e(\mathcal{N}_e, \mathcal{E}_e)$ with respect to the dependencies `uses-type` and `defines`. The nodes `T1` and `P1` are connected by the



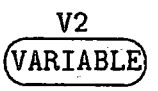
$\delta(\mathtt{G}_e(\mathcal{N}_e,\ \mathcal{E}_e), \{\mathtt{of\text{-}type}\})$

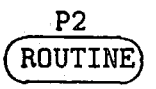
(a)



$\delta(\mathtt{G}_e(\mathcal{N}_e,\ \mathcal{E}_e), \{\mathtt{uses\text{-}type}, \mathtt{defines}\})$

(b)





$\delta(\mathtt{G}_e(\mathcal{N}_e,\ \mathcal{E}_e), \{\mathtt{\$ISOLATED\$}\})$

(c)

Figure 5.8: Three δ slices of Figure 5.7

uses-type dependency and the nodes V1 and P1 are connected by the defines dependency. The graph resulting from,

$$\delta(\mathrm{G}_e(\mathcal{N}_e, \mathcal{E}_e), \{\texttt{uses-type}, \texttt{defines}\})$$

is the union of the δ-slicing with respect to uses-type and the δ-slicing with respect to defines, i.e.,

$$\delta(\mathrm{G}_e(\mathcal{N}_e, \mathcal{E}_e), \{\texttt{uses-type}, \texttt{defines}\}) \equiv$$
$$\delta(\mathrm{G}_e(\mathcal{N}_e, \mathcal{E}_e), \{\texttt{uses-type}\}) \sqcup \delta(\mathrm{G}_e(\mathcal{N}_e, \mathcal{E}_e), \{\texttt{defines}\})$$

In general, a δ-slicing operation,

$$\delta(\mathrm{G}(\mathcal{N}, \mathcal{E}), \{\mathtt{dep}_1, \ldots, \mathtt{dep}_n\})$$

that slices with respect to n dependencies, $\mathtt{dep}_1$, ..., $\mathtt{dep}_n$, can be shown to be equivalent to,

$$\bigsqcup\{\delta(\mathrm{G}(\mathcal{N}, \mathcal{E}), \{\mathtt{dep}_1\}), \ldots, \delta(\mathrm{G}(\mathcal{N}, \mathcal{E}), \{\mathtt{dep}_n\})\}$$

Finally, Figure 5.8(c) shows the result of slicing $\mathrm{G}_e(\mathcal{N}_e, \mathcal{E}_e)$ with respect to the special dependency '$ISOLATED$'. The resulting graph consists of the isolated nodes P2 and V2.

5.4.2 $\alpha\beta$-Slicing

$\alpha\beta$-slicing is a form of graph slicing where information associated with the node of a graph is used to determine which nodes appear in the resulting graph. Since the dependency represented by an edge is not considered in $\alpha\beta$-slicing, no restriction is placed on the edges that can appear in the graph resulting from $\alpha\beta$-slicing. An $\alpha\beta$-slicing operation is denoted as follows,

$$_{\alpha}\|_{\beta}(\mathrm{G}(\mathcal{N}, \mathcal{E}), \mathcal{C})$$

In this notation the slicing criterion for an $\alpha\beta$-slicing operation is in two parts. The first part of the slicing criterion appears as the argument $\mathcal{C}$, and the second part of the slicing criterion are the α and β constraints on the slicing operator $\|$. Both parts of the slicing criterion will be described in more detail below.

The Argument $\mathcal{C}$

The argument $\mathcal{C}$ represents the node based slicing criterion where the nodes that can appear in the resulting graph are named. The argument $\mathcal{C}$ is an ordered binary tuple, where the first element is the set of nodes that can be the start-node for an edge, and the second element is the set of nodes that can be the stop-node for an edge.

Consider, for example, the slicing operation,

$$_{\epsilon}\|_{\epsilon}(\mathrm{G}_e(\mathcal{N}_e, \mathcal{E}_e), \langle\{\mathtt{T1}, \mathtt{V1}, \mathtt{V2}\}, \{\mathtt{P1}, \mathtt{T3}\}\rangle) \tag{5.1}$$

where $G_e(\mathcal{N}_e, \mathcal{E}_e)$ is the graph in Figure 5.7. The nodes T1, V1, and V2 are named as valid start-nodes, and the nodes P1 and T3 are named as valid stop-nodes.

It is often useful to be able to specify that no restriction is being placed either on the set of possible nodes or on the α and β constraints. The symbol used to denote this is ξ. When ξ is used as the α or β constraints, as in the operation (5.1), then this means that no restrictions are being placed on the labels associated with the nodes in the resulting graph. When ξ is used as part of the argument $\mathcal{C}$, then this means that no restriction is being placed on the set of start-nodes, the set of stop-nodes or both.

The graph resulting from the operation (5.1) is given in Figure 5.9. The set of nodes in this graph is a subset of the nodes named in the argument of the operation (5.1), because not all the named nodes satisfy the start-node and stop-node requirements.

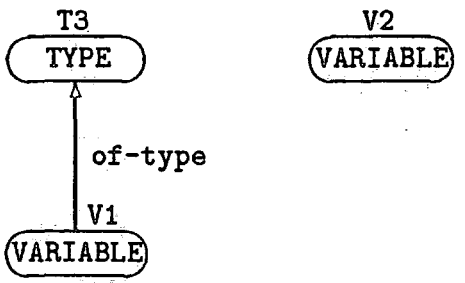


Figure 5.9: The Slice $_{\xi}\|_{\xi}(G_e(\mathcal{N}_e, \mathcal{E}_e), \langle\{T1, V1, V2\}, \{P1, T3\}\rangle)$

For the purpose of graph slicing, an isolated node will pass through to the resulting graph if it is named as a valid start-node or a valid stop-node. For a non-isolated node to appear in the graph resulting from an $\alpha\beta$-slicing then that node must be named as being either a valid start-node or stop-node for the resulting graph, and one of the other nodes to which it is connected in the original graph

must also be named so that the connecting edge can appear in the resulting graph.

Consider the graph $G_e(\mathcal{N}_e, \mathcal{E}_e)$. In the set $\mathcal{E}_e$ there are the edges (V1, T3) and (P1, T1). The edge (V1, T3) appears in the graph resulting from the operation (5.1) because V1 is named as a valid start-node and T3 as a valid stop node. The edge (P1, T1) does not appear however as P1 is not named as a valid start-node and T1 is not named as a valid stop-node. The node T1 has been classed as a valid start-node and P1 as a valid stop-node, but this does not allows the edge (P1, T1) to appear in the graph resulting from the operation (5.1). The node V2 is isolated in $G_e(\mathcal{N}_e, \mathcal{E}_e)$, however it appears in the resulting graph because it is named as a valid stop-node.

A VDM specification for the $\alpha\beta$-slicing operation with respect to the argument C is,

Node-Criterion :: *start-set* : set of *Node*
stop-set : set of *Node*

$_\alpha\|_\beta : Graph \times Node\text{-}Criterion \rightarrow Graph$

$$_{\alpha}\|_{\beta}(mk\text{-}Graph(nodes, edges), mk\text{-}Node\text{-}Criterion(start\text{-}set, stop\text{-}set)) \quad \triangleq$$

$$\begin{aligned}
&\textsf{let } edgeset = \{e \mid e \in edges \wedge \\
&\qquad start\text{-}node(e) \in start\text{-}set \wedge \\
&\qquad stop\text{-}node(e) \in stop\text{-}set\} \quad \textsf{in} \\
&\textsf{let } isolated\text{-}nodeset = \{n \mid n \in (start\text{-}set \cup stop\text{-}set) \wedge \\
&\qquad \forall e \in edges \cdot n \neq start\text{-}node(e) \wedge \\
&\qquad n \neq stop\text{-}node(e)\} \quad \textsf{in} \\
&\textsf{let } non\text{-}isolated\text{-}nodeset = \{n \mid n \in (start\text{-}set \cup stop\text{-}set) \\
&\qquad \exists e \in edgeset \cdot (n = start\text{-}node(e) \vee \\
&\qquad n = stop\text{-}node(e))\} \quad \textsf{in} \\
&\textsf{let } nodeset = isolated\text{-}nodeset \cup non\text{-}isolated\text{-}nodeset \quad \textsf{in} \\
&mk\text{-}Graph(nodeset, edgeset)
\end{aligned}$$

The α and β Constraints

The argument $\mathcal{C}$ allows the nodes that appear in the resulting graph to be named. It is often useful to be able to extract a subgraph by specifying properties that must be held by the labels of the nodes in the resulting graph. This is done through the α and β constraints on the slicing operator $\|$. The α constraints apply to the labels for the start-nodes, and the β constraints apply to the labels for the stop-nodes. In the VDM description of an interconnection graph given in Figure 4.6 (page 64) the label of a node is described as having three attributes, namely: the *entity-class*, which records the class of the entity associated with node; the *entity-source* which records the module in which the entity is declared; and the *entity-declaration-block*, which records the block number for the block in which the entity is declared.

The α and β constraints of the $\alpha\beta$-slicing operation will be considered in turn. The α constraint is considered first. An example of such a constraint is,

$$_{class=ROUTINE}\|_{\xi}(\mathrm{G}_e(\mathcal{N}_e, \mathcal{E}_e), \langle \xi, \xi \rangle) \tag{5.2}$$

This operation only places restrictions on the labels for the nodes that can be start-nodes in the resulting graph. The argument $\langle \xi, \xi \rangle$ says that any of the nodes in $\mathcal{N}_e$ can be nodes in the resulting graph. The result of the operation (5.2) is given in Figure 5.10.

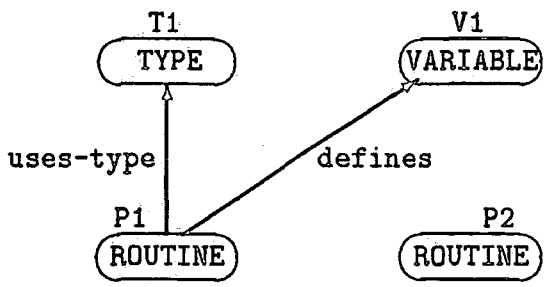


Figure 5.10: The Slice $_{class=ROUTINE}\|_{\xi}(\mathrm{G}_e(\mathcal{N}_e, \mathcal{E}_e), \langle \xi, \xi \rangle)$

$\mathrm{G}_e(\mathcal{N}_e, \mathcal{E}_e)$ has two nodes associated with routines: the node `P2`, which is isolated; and the node `P1`, which is connected to the node `T1` by the edge (`P1`, `T1`) and to the node `V1` by the edge (`P1`, `V1`). As both these edges have `P1` as the start-node they both appear in the resulting graph, which implies that the nodes `T1` and `V1` also appear.

Since a label has three attributes (class, source and block number) three functions are needed to extract the appropriate subgraph. These three functions can be specified as:

$alpha\text{-}class : Graph \times Class \rightarrow Graph$

$alpha\text{-}class(mk\text{-}Graph(nodes, edges), class\text{-}name) \triangleq$

let $start\text{-}nodeset = \{n \mid n \in nodes \wedge$

$entity\text{-}class(node\text{-}label(n)) = class\text{-}name\}$ in

let $edgeset = \{e \mid e \in edges \wedge$

$start\text{-}node(e) \in start\text{-}nodeset\}$ in

let $stop\text{-}nodeset = \{stop\text{-}node(e) \mid e \in edgeset\}$ in

$mk\text{-}Graph((start\text{-}nodeset \cup stop\text{-}nodeset), edgeset)$

$alpha\text{-}source : Graph \times Source \rightarrow Graph$

$alpha\text{-}source(mk\text{-}Graph(nodes, edges), module\text{-}name) \triangleq$

let $start\text{-}nodeset = \{n \mid n \in nodes \wedge$

$entity\text{-}source(node\text{-}label(n)) = module\text{-}name\}$ in

let $edgeset = \{e \mid e \in edges \wedge$

$start\text{-}node(e) \in start\text{-}nodeset\}$ in

let $stop\text{-}nodeset = \{stop\text{-}node(e) \mid e \in edgeset\}$ in

$mk\text{-}Graph((start\text{-}nodeset \cup stop\text{-}nodeset), edgeset)$

$alpha\text{-}block\text{-}number : Graph \times Block\text{-}Number \rightarrow Graph$

$alpha\text{-}block\text{-}number(mk\text{-}Graph(nodes, edges), block\text{-}number) \triangleq$

let $start\text{-}nodeset = \{n \mid n \in nodes \wedge$

$entity\text{-}declaration\text{-}block(node\text{-}label(n)) = block\text{-}number\}$ in

let $edgeset = \{e \mid e \in edges \wedge$

$start\text{-}node(e) \in start\text{-}nodeset\}$ in

let $stop\text{-}nodeset = \{stop\text{-}node(e) \mid e \in edgeset\}$ in

$mk\text{-}Graph((start\text{-}nodeset \cup stop\text{-}nodeset), edgeset)$

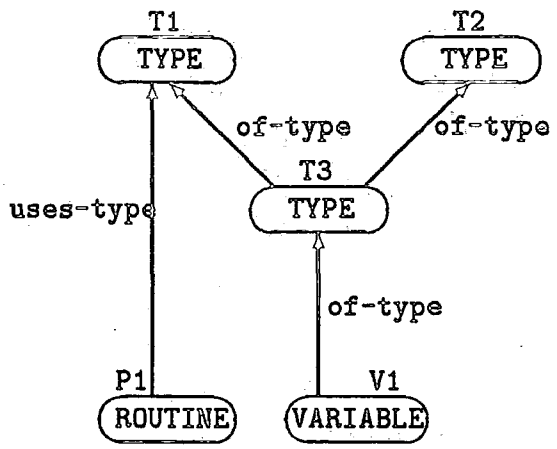


Figure 5.11: Result of the Slice $\xi \| _{class=TYPE}(\mathrm{G}_e(\mathcal{N}_e, \mathcal{E}_e), \langle \xi, \xi \rangle)$

We now consider the β constraints. Consider the slicing operation,

$$\xi \| _{class=TYPE}(\mathrm{G}_e(\mathcal{N}_e, \mathcal{E}_e), \langle \xi, \xi \rangle) \tag{5.3}$$

This operation places a restriction on the labels of the nodes that can be stop-nodes in the resulting graph. In the graph resulting from operation (5.3) the label of all the stop-nodes must have the value 'TYPE' associated with the *entity-class* attribute. Figure 5.11 gives the graph resulting from operation (5.3).

The β slicing functions can be specified in a similar way to the α slicing functions.

$beta\text{-}class : Graph \times Class \rightarrow Graph$

$beta\text{-}class(mk\text{-}Graph(nodes, edges), class\text{-}name) \triangleq$

let $stop\text{-}nodeset = \{n \mid n \in nodes \wedge$

$entity\text{-}class(node\text{-}label(n)) = class\text{-}name\}$ in

let $edgeset = \{e \mid e \in edges \wedge$

$stop\text{-}node(e) \in stop\text{-}nodeset\}$ in

let $start\text{-}nodeset = \{start\text{-}node(e) \mid e \in edges\}$ in

$mk\text{-}Graph((start\text{-}nodeset \cup stop\text{-}nodeset), edgeset)$

$beta\text{-}source : Graph \times Source \rightarrow Graph$

$beta\text{-}source(mk\text{-}Graph(nodes, edges), module\text{-}name) \triangleq$

let $stop\text{-}nodeset = \{n \mid n \in nodes \wedge$

$entity\text{-}source(node\text{-}label(n)) = module\text{-}name\}$ in

let $edgeset = \{e \mid e \in edges \wedge$

$stop\text{-}node(e) \in stop\text{-}nodeset\}$ in

let $start\text{-}nodeset = \{start\text{-}node(e) \mid e \in edges\}$ in

$mk\text{-}Graph((start\text{-}nodeset \cup stop\text{-}nodeset), edgeset)$

$beta\text{-}block\text{-}number : Graph \times Block\text{-}Number \rightarrow Graph$

$beta\text{-}block\text{-}number(mk\text{-}Graph(nodes, edges), block\text{-}number) \triangleq$

let $stop\text{-}nodeset = \{n \mid n \in nodes \wedge$

$entity\text{-}declaration\text{-}block(node\text{-}label(n)) = block\text{-}number\}$ in

let $edgeset = \{e \mid e \in edges \wedge$

$stop\text{-}node(e) \in stop\text{-}nodeset\}$ in

let $start\text{-}nodeset = \{start\text{-}node(e) \mid e \in edges\}$ in

$mk\text{-}Graph((start\text{-}nodeset \cup stop\text{-}nodeset), edgeset)$

Finally, operations that involve both an α and a β constraint will be considered. The $\alpha\beta$-slicing operation,

$$_{\alpha}\|_{\beta}(\mathrm{G}(\mathcal{N}, \mathcal{E}), \mathcal{C})$$

can be considered in terms of the operations:

- $_{\epsilon}\|_{\epsilon}(\mathrm{G}(\mathcal{N}, \mathcal{E}), \mathcal{C})$
- $_{\alpha}\|_{\epsilon}(\mathrm{G}(\mathcal{N}, \mathcal{E}), \langle \xi, \xi \rangle)$
- $_{\epsilon}\|_{\beta}(\mathrm{G}(\mathcal{N}, \mathcal{E}), \langle \xi, \xi \rangle)$

which have already been discussed. The result of the operation $_{\epsilon}\|_{\epsilon}(\mathrm{G}(\mathcal{N}, \mathcal{E}), \mathcal{C})$ is the subgraph of $\mathrm{G}(\mathcal{N}, \mathcal{E})$ containing all the nodes satisfying the argument $\mathcal{C}$ together with any of the edges that connect the nodes. Similarly, the results of the operations $_{\alpha}\|_{\epsilon}(\mathrm{G}(\mathcal{N}, \mathcal{E}), \langle \xi, \xi \rangle)$ and $_{\epsilon}\|_{\beta}(\mathrm{G}(\mathcal{N}, \mathcal{E}), \langle \xi, \xi \rangle)$ are the subgraphs of $\mathrm{G}(\mathcal{N}, \mathcal{E})$ satisfying the α and β constraints respectively. Intuitively, the result of the operation $_{\alpha}\|_{\beta}(\mathrm{G}(\mathcal{N}, \mathcal{E}), \mathcal{C})$ should be the subgraph of $\mathrm{G}(\mathcal{N}, \mathcal{E})$ that is common in all three graphs, i.e.,

$$\sqcap\{_{\epsilon}\|_{\epsilon}(\mathrm{G}(\mathcal{N}, \mathcal{E}), \mathcal{C}), _{\alpha}\|_{\epsilon}(\mathrm{G}(\mathcal{N}, \mathcal{E}), \langle \xi, \xi \rangle) _{\epsilon}\|_{\beta}(\mathrm{G}(\mathcal{N}, \mathcal{E}), \langle \xi, \xi \rangle)\}$$

The result of this operation, however, is not the same as that for $_{\alpha}\|_{\beta}(\mathrm{G}(\mathcal{N}, \mathcal{E}), \mathcal{C})$.

Consider the slicing operation,

$$_{class=ROUTINE}\|_{class=TYPE}(\mathrm{G}_e(\mathcal{N}_e, \mathcal{E}_e), \langle \xi, \xi \rangle)$$

where $\mathrm{G}_e(\mathcal{N}_e, \mathcal{E}_e)$ is the graph given in Figure 5.7. If the argument $\mathcal{C}$ is $\langle \xi, \xi \rangle$ then

the result of,

$$\varepsilon\|_{\xi}(\mathrm{G}_e(\mathcal{N}_e, \mathcal{E}_e), \langle \xi, \xi \rangle)$$

is $\mathrm{G}_e(\mathcal{N}_e, \mathcal{E}_e)$ itself. The result of the operations,

$$_{class=ROUTINE}\|_{\xi}(\mathrm{G}_e(\mathcal{N}_e, \mathcal{E}_e), \langle \xi, \xi \rangle) \text{ and } \varepsilon\|_{class=TYPE}(\mathrm{G}_e(\mathcal{N}_e, \mathcal{E}_e), \langle \xi, \xi \rangle)$$

are given in Figure 5.10 and Figure 5.11 respectively.

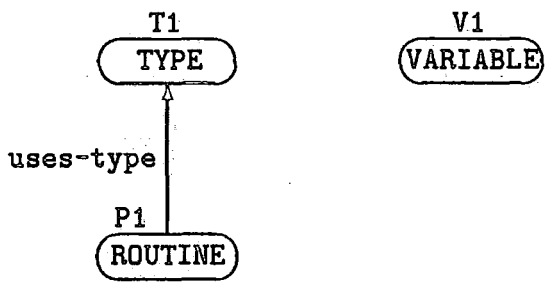


Figure 5.12: The Strict Intersection of the Graphs in Figures 5.7, 5.10 and 5.11

The result of the operation

$$\sqcap\{\varepsilon\|_{\xi}(\mathrm{G}(\mathcal{N}, \mathcal{E}), \langle \xi, \xi \rangle),\ _{class=ROUTINE}\|_{\xi}(\mathrm{G}(\mathcal{N}, \mathcal{E}), \langle \xi, \xi \rangle)$$
$$\varepsilon\|_{class=TYPE}(\mathrm{G}(\mathcal{N}, \mathcal{E}), \langle \xi, \xi \rangle)\} \qquad (5.4)$$

is given in Figure 5.12. The isolated node V1 satisfies neither the α or β constraints, therefore it should not appear. The node V1 belongs to the result of operation (5.4), because it is a property of distributed strict graph intersection that a node common to all the given graphs appears in the resulting graph.

The node V1 has appeared in the result of operation (5.4) because V1 is not a real isolated node. If the node V1 had been isolated in $\mathrm{G}_e(\mathcal{N}_e, \mathcal{E}_e)$ then the graphs resulting from slicing $\mathrm{G}_e(\mathcal{N}_e, \mathcal{E}_e)$ with respect to the α and β constraints would

have omitted V1 as it does not satisfy the given constraint. The node V1 appears in the graph resulting from slicing $G_e(\mathcal{N}_e, \mathcal{E}_e)$ with respect to the α constraint because it is the stop-node for the edge (P1, V1), where P1 satisfies the α constraint. The node V1 also appears in the graph resulting from slicing with respect to the β constraint because it is the start-node for the edge (V1, T3), where T3 satisfies the β constraint.

The result of the operation

$$_{class=ROUTINE}\|_{class=TYPE}(G_e(\mathcal{N}_e, \mathcal{E}_e), \langle \xi, \xi \rangle)$$

is the result of the operation (5.4) with the false isolated node V1 removed. A function *remove-false-isolated-nodes* can be specified as follows,

$$\begin{aligned}
&\textit{remove-false-isolated-nodes} : \textit{Graph} \times \textit{Graph} \rightarrow \textit{Graph} \\
&\textit{remove-false-isolated-nodes}(\textit{graph1}, \textit{graph2}) \quad \triangleq \\
&\quad \textsf{let } \textit{false-isolated-nodes} = \{n \mid n \in \textit{nodes}(\textit{graph2}) \wedge \\
&\qquad\qquad \textit{isolated}(n, \textit{graph2}) \wedge \\
&\qquad\qquad \neg \textit{isolated}(n, \textit{graph1})\} \quad \textsf{in} \\
&\quad \textsf{let } \textit{nodeset} = \textit{nodes}(\textit{graph2}) - \textit{false-isolated-nodes} \quad \textsf{in} \\
&\quad \textit{mk-Graph}(\textit{nodeset}, \textit{edges}(\textit{graph2}))
\end{aligned}$$

where the function *isolated* determines if a given node is isolated within a given graph. This function can be specified as,

$$isolated : Node \times Graph \rightarrow \mathbb{B}$$

$$isolated(node, graph) \triangleq$$
$$\forall e \in edges(graph) \cdot$$
$$(node \neq start\text{-}node(e) \wedge node \neq stop\text{-}node(e))$$

In general,

$$_{\alpha}\|_{\beta}(\mathrm{G}(\mathcal{N}, \mathcal{E}), \mathcal{C}) \equiv remove\text{-}false\text{-}isolated\text{-}nodes(\mathrm{G}(\mathcal{N}, \mathcal{E}), \mathrm{G}_i(\mathcal{N}_i, \mathcal{E}_i))$$

where $\mathrm{G}_i(\mathcal{N}_i, \mathcal{E}_i)$ is the graph resulting from,

$$\prod\{_{\xi}\|_{\xi}(\mathrm{G}(\mathcal{N}, \mathcal{E}), \mathcal{C}),\ _{\alpha}\|_{\xi}(\mathrm{G}(\mathcal{N}, \mathcal{E}), \langle \xi, \xi \rangle),\ _{\xi}\|_{\beta}(\mathrm{G}(\mathcal{N}, \mathcal{E}), \langle \xi, \xi \rangle)\}$$

5.5 Examples of the Uses of Graph Operations

The graph operations specified in this chapter are used in inter-module code analysis as they provide a means of reasoning about a graph. Three examples of how the graph operations can be used are given here to help clarify how the graph operations can help in code analysis work.

5.5.1 Disjoint Graphs

Chapter 4 describes a disjoint graph as being a graph for which there is not a path between every pair of nodes. Determining whether a graph is disjoint is important in code analysis as it reveals independence properties between entities. Consider for example a call graph. If the call graph shows the dependencies between all the routines in a system rather than just within a module, then it is expected that the call graph will be connected, with the program body being considered as a routine. If the call graph is disjoint then the system contains routines which are never called. A function to detect whether a graph is disjoint can be specified as follows. This specification is written in a language that is an extension of VDM which considers graphs as primitive structures. The symbol ϕ denotes the empty graph.

$$
\begin{aligned}
&\mathit{is\text{-}disjoint} : \mathit{Graph} \rightarrow \mathbb{B} \\
&\mathit{is\text{-}disjoint}(\mathit{graph}) \quad \triangleq \\
&\quad \exists \mathit{subgraph1}, \mathit{subgraph2} \sqsubset \mathit{graph} \cdot \\
&\qquad \mathit{subgraph1} \neq \phi \wedge \\
&\qquad \mathit{subgraph2} \neq \phi \wedge \\
&\qquad (\mathit{subgraph1} \sqcap \mathit{subgraph2}) = \phi \wedge \\
&\qquad (\mathit{subgraph1} \sqcup \mathit{subgraph2}) = \mathit{graph}
\end{aligned}
$$

This specification states that a graph is disjoint if two non-empty subgraphs can be found such that the two subgraphs have no nodes or edges in common, but the two subgraphs combined give the original graph.

5.5.2 Proper Subgraphs

Chapter 4 describes a proper subgraph as being a strict subgraph that is isolated in the given graph. The specification of the function *get-proper-subgraphs*, given below, describes the characteristics of a function that finds all the proper subgraphs in a given graph. If the given graph is connected and therefore contains no proper subgraphs, the function *get-proper-subgraphs* returns the graph itself. This is to ensure the property,

$$graph = \bigsqcup get\text{-}proper\text{-}subgraphs(graph)$$

$get\text{-}proper\text{-}subgraphs : Graph \rightarrow$ set of $Graph$

$get\text{-}proper\text{-}subgraphs(graph) \triangleq$

if $is\text{-}disjoint(graph)$

then $\{g \mid g \sqsubset graph \wedge$

$g \neq \phi \wedge$

$\exists not\text{-}g \sqsubset graph \cdot (not\text{-}g \neq \phi \wedge$

$g \sqcap not\text{-}g = \phi \wedge$

$g \sqcup not\text{-}g = graph)\}$

else $graph$

5.5.3 Abstract Data Types

It is often desirable to extract subgraphs that satisfy particular constraints. An example of this is finding the abstract data types in a graph. Booch gives a very

loose interpretation of an abstract data type in [16, pages 228–9]. Booch classifies an abstract data type as being a set of *associated* types, constants and routines. The slicing operation to extract this form of abstract data type is

$$\begin{aligned}(_{class=ROUTINE \vee CONSTANT} \| _{class=TYPE}(\mathrm{G}(\mathcal{N}, \mathcal{E}), \langle \xi, \xi \rangle)) \sqcup \\ (_{class=TYPE} \| _{class=TYPE}(\mathrm{G}(\mathcal{N}, \mathcal{E}), \langle \xi, \xi \rangle)) \sqcup \\ (_{class=TYPE} \| _{class=CONSTANT}(\mathrm{G}(\mathcal{N}, \mathcal{E}), \langle \xi, \xi \rangle))\end{aligned} \tag{5.5}$$

The second slicing operation in (5.5) is to allow for when a constant is used to delimit a value in a type declaration, e.g.,

```
CONST MaxSize = 100;
TYPE  IntStack = ARRAY [1..MaxSize] OF INTEGER;
...
```

The more classical description of an abstract data type stipulates that the routine would have to use the type to either declare a parameter, or to declare the type of value returned by the routine. The appropriate slicing operation for this form of abstract data type is

$$\begin{aligned}&\delta(_{class=ROUTINE \vee CONSTANT} \| _{class=TYPE} \\ &\qquad (\mathrm{G}(\mathcal{N}, \mathcal{E}), \langle \xi, \xi \rangle), \{\texttt{parameter-of-type, of-type}\}) \sqcup \\ &\delta(_{class=TYPE} \| _{class=CONSTANT} \\ &\qquad (\mathrm{G}(\mathcal{N}, \mathcal{E}), \langle \xi, \xi \rangle), \{\texttt{delimited-by}\})\end{aligned}$$

The dependencies `parameter-of-type`, `of-type` and `delimited-by` are dependencies that appear in the entity-to-entity interconnection graph that is discussed in Chapter 8.

Chapter 6

Module-to-Module Interconnection Graph

6.1 Introduction

A module-to-module interconnection graph is a specialised form of interconnection graph that shows the dependencies between the modules in a system. This graph gives a high level representation of the structure of a system, showing how a system has been decomposed into subsystems (represented by modules) during the initial system design stage and subsequent maintenance stages.

This form of graph is important for a maintenance programmer, as it provides a useful form of documentation that is generated from the program code. This makes the module-to-module interconnection graph especially useful for a maintenance

programmer who is working with unfamiliar code that is either undocumented, or incorrectly documented because the documentation has either not been maintained along with the software, or was hurriedly written at the end of the system development.

The module-to-module interconnection graph has several uses in software maintenance: it can serve as a map of a system, showing in which modules are dependent on each other; and it can be analysed to help classify modules. As a map of a system, the module-to-module interconnection graph can guide a maintenance programmer's scanning of the associated code and can help a maintenance programmer to consider in which order to look at the modules. As a module classification aid, the module-to-module interconnection graph can help a maintenance programmer classify a module according to its apparent role within a system. In this way a maintenance programmer can identify modules that require closer examination in order to understand a system properly.

This chapter will concentrate on the analysis aspects of the module-to-module interconnection graph, showing how modules can be classified according to their role within a system. Section 6.2 describes in more detail the module-to-module interconnection graph, describing the dependencies that are recorded. Section 6.3 describes how the module-to-module interconnection graph can be used to classify modules, and finally section 6.4 demonstrates these ideas on module classification by considering the module-to-module interconnection graph for a particular system.

6.2 Characteristics of the Module-to-Module Interconnection Graph

The module-to-module interconnection graph is a specialised form of interconnection graph, that only records the dependencies between modules. This makes the module-to-module interconnection graph a high level graph that shows the dependencies between the main components of a system. In so doing the module-to-module interconnection graph provides an interpretation of the architectural structure of a system.

The nodes of an module-to-module interconnection graph represent the modules that comprise a system, and an edge between two nodes denotes the existence of a dependency between the modules associated with the nodes. The modules that comprise a system can be a combination of **global modules**, where each module is an outermost entity, and **local modules**, where each module is declared within some block.

```
MODULE GlobalModule;
   (* Block 0 *)
   ...
   MODULE LocalModule;
     (* Block 1 *)
     ...
   END LocalModule;
   ...
END GlobalModule.
```

Figure 6.1: A Program Module Containing a Local Module Declaration

The nodes for module entities record that the associated entity is a module by

storing the value MODULE in the label attribute *entity-class*. As with other classes of entities, the name of the global module in which an entity is declared is stored in the label attribute *entity-source* and the number associated with the block in which the module is declared is stored in the label attribute *entity-declaration-block*. Consider for example the Modula-2 program module given in Figure 6.1. The nodes associated with these two modules is given in Figure 6.2.

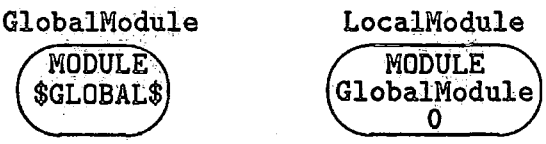


Figure 6.2: Nodes for a Module-to-Module Interconnection Graph

The node for LocalModule has the value MODULE stored as the entity class, GlobalModule as the entity source and 0 (zero) as the number of the block in which LocalModule is declared. If a module is global, it is not declared within another module, this necessitates that the special value $GLOBAL$ be recorded in the label attribute *entity-source*. The module GlobalModule is a global module, therefore the value $GLOBAL$ is recorded. Similarly, as a global module is external to all blocks, there is no value stored in the label attribute *entity-declaration-block*. Hence, the node for GlobalModule has only two values associated with it.

In the interconnection graphs in this thesis, only the information needed for the example will appear in each graph. This is done solely to reduce the amount of information being conveyed to a minimum, so that the reader is not overwhelmed with unnecessary information and can concentrate on the important aspects that the graph is trying to convey.

An edge in a module-to-module interconnection graph represents the depen-

dency that exists between two modules. The dependencies that can be recorded in a module-to-module interconnection graph are:

- `local-to`

  Shows the dependency of a local module on a global or local module.

- `uses`

  Shows the dependencies between two global modules. This form of dependency denotes the existence of a client/supplier relationship between the modules involved.

- `instantiates-to`

  Shows the dependency of a **concrete module** on a **generic module**.

- `inherits-from`

  Shows the dependency between two modules, where one module is an extension or specialisation of the other.

The number of dependencies connecting two modules is normally one, as the existence of one dependency tends to exclude the others. For example, if two modules are connected by an `instantiates-to` dependency, then the other dependencies would be meaningless between the two modules concerned. This also applies for the `local-to` dependency. Two modules can be connected twice by the `uses` dependency, because the edges (`A`, `B`) and (`B`, `A`), where `A` and `B` are modules, can both represent a `uses` dependency.

It is possible to have two modules connected by the `uses` and `inherits-from` dependencies, because the dependencies are not mutually exclusive. Consider the situation described in Figure 6.3. The module `A` is built as an extension or specialisation of module `B`, and therefore it has direct access to all the entities in `B`

together with those it declares itself. One of the entities declared in module A could take the form,

class *A*
inherits *B*
feature *P*: *B*
⋮
end -- class *A*

This declares a variable P to be of type B, therefore the uses dependency appears in Figure 6.3.

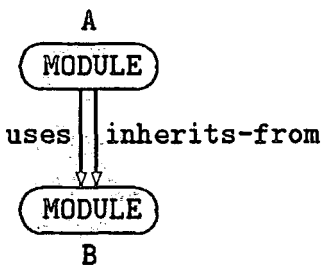


Figure 6.3: Example of Two Modules Connected by Two Dependencies

Each of the dependencies that can exist between two modules will be discussed in the following subsections.

6.2.1 The local-to Dependency

The local-to dependency states that the module associated with the start-node of the edge is local to the module associated with the stop-node of the edge.

Consider the TEXT_IO package (module) given in section 14.3.10 of *The Reference Manual for the Ada Programming Language*. (The parts of TEXT_IO that are relevant to this discussion are given in Figure 6.4.) TEXT_IO is declared to have four local packages. The fact that these local packages are generic does not affect the dependency between them and TEXT_IO. Figure 6.5 gives the module-to-module interconnection graph for TEXT_IO. This graph has four edges each of which is representing a `local-to` dependency.

```
package TEXT_IO is
   ...
   generic
      ...
   package INTEGER_IO is
      ...
   end INTEGER_IO;

   generic
      ...
   package FLOAT_IO is
      ...
   end FLOAT_IO;

   generic
      ...
   package FIXED_IO is
      ...
   end FIXED_IO;

   generic
      ...
   package ENUMERATION_IO is
      ...
   end ENUMERATION_IO;
end TEXT_IO;
```

Figure 6.4: The Packages in Ada's TEXT_IO

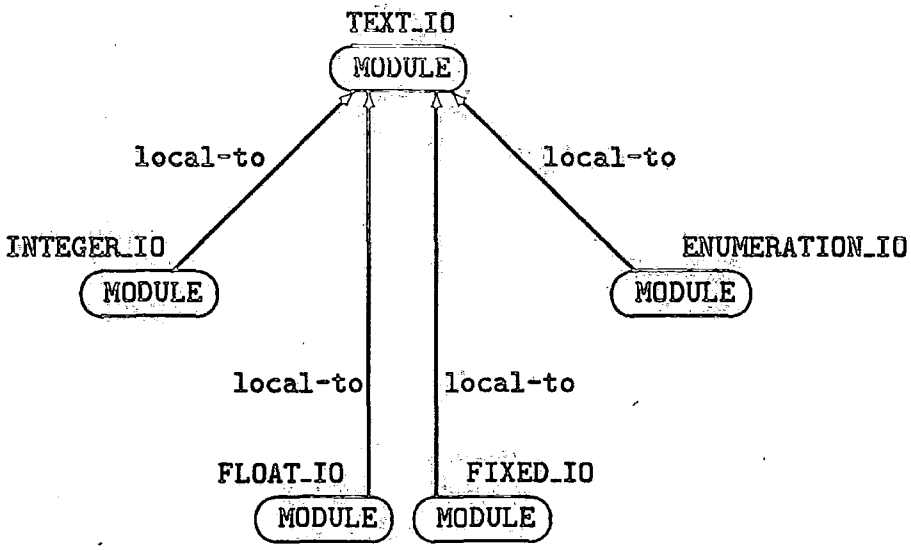


Figure 6.5: The Module-to-Module Interconnection Graph for TEXT_IO

The reasons for using local modules in a system are not easily defined. If the code of a local module is providing a logically distinct set of entities to those of the encapsulating module, then it better to separate the two modules textually by making the local module a global module. Wirth makes the observation,

> "Experience with Modula over the last eight years has shown that *local modules* were rarely used." [171]

Wirth uses this observation to justify the omission of local modules from Oberon. Many other module languages also do not provide local modules, so in the module-to-module interconnection graphs for programs written in these languages there will be no `local-to` dependencies. With languages like Ada and Modula-2 which do provide local modules, a closer examination of what the `local-to` dependency represents is needed.

The reason for using local modules can vary from language to language. In Modula-2, for example it is not possible to initialise a variable at its point of declaration. To do this in Modula-2 the programmer has to use a local module in the following manner,

```
MODULE DeclarationOfi;
  EXPORT i;
  VAR i:  INTEGER;
BEGIN
  i:= 1
END DeclarationOfi;
```

Here a module is being used to overcome a perceived weakness in Modula-2.

More generally however, a local module provides a programmer with a mechanism by which he can secure the code of part of the subsystem he is implementing. The code placed within the local module is therefore perceived to be logically related to the rest of the encapsulating module by the programmer implementing the module, because he has chosen not to use a global module.

6.2.2 The uses Dependency

The uses dependency is employed when one global module imports, or is permitted use of, an entity from another global module. This is an important form of inter-module connection as it is available in all the module languages. The uses dependency is employed when building program families, and denotes the existence of a client/supplier relationship between the two modules involved. The

start-node for an edge representing a uses dependency denotes the client module, while the stop-node denotes the supplier module. With the classical design methods, the structure of a system under the uses dependency is a tree or directed acyclic graph. In practice this is not always true (see Figure 6.16 on page 133).

```
package WORLD_SYSTEM is
   ...
end WORLD_SYSTEM;

with WORLD_SYSTEM;
package ACTUAL_TARGET is
   ...
end ACTUAL_TARGET;

with WORLD_SYSTEM;
package ARMAMENT_STATUS is
   ...
end ARMAMENT_STATUS;

with WORLD_SYSTEM;
package FLIGHT_PARAMETER is
   ...
end FLIGHT_PARAMETER;

with WORLD_SYSTEM;
package TARGET_BOX is
   ...
end TARGET_BOX;

package body HEADS_UP_DISPLAY is
   ...
end HEADS_UP_DISPLAY;
```

Figure 6.6: The Packages for Heads-Up Display

Consider the "Heads-Up Display" system developed by Booch [16, Chapter 21]. A Heads-Up Display system is a way of providing important flight information to the pilot of a fighter aircraft without the pilot having to look away from a target

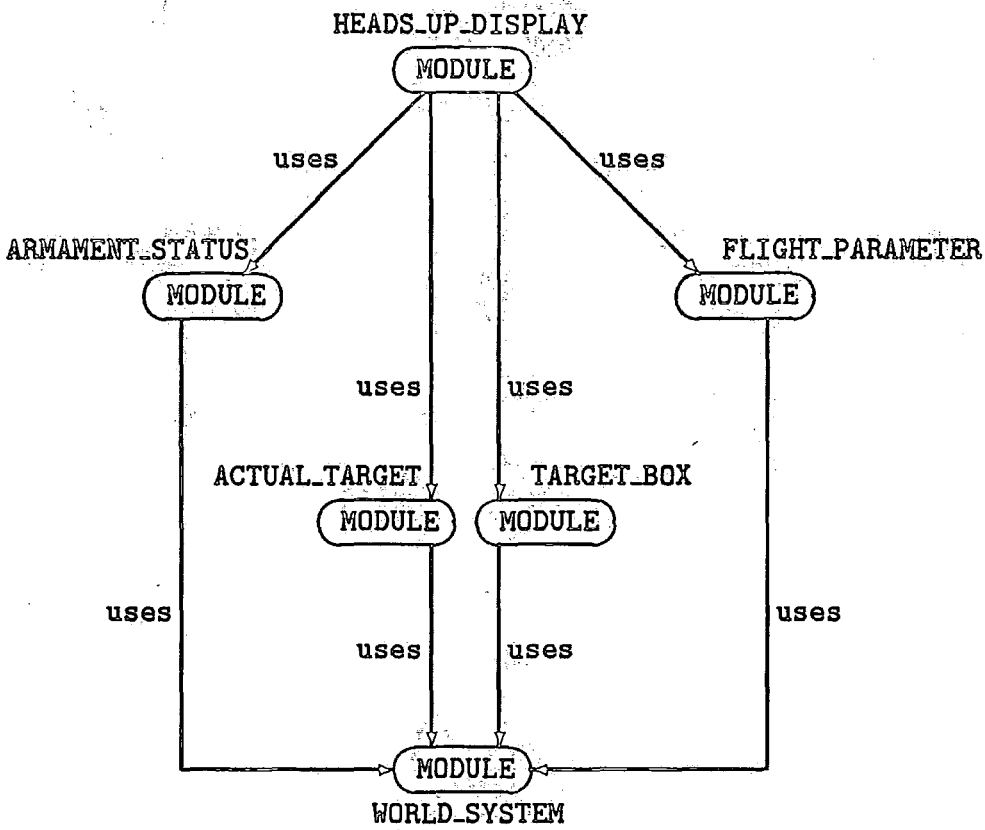


Figure 6.7: The Module-to-Module Interconnection Graph for Heads-Up Display

aircraft. The Heads-Up Display system developed by Booch consists of six packages. The appropriate package declarations are given in Figure 6.6. Figure 6.7 gives the module-to-module interconnection graph for this system. The edges connecting ACTUAL_TARGET, ARMAMENT_STATUS, TARGET_BOX and FLIGHT_PARAMETER with WORLD_SYSTEM record that the package WORLD_SYSTEM is providing some facilities to the other packages. (In this instance WORLD_SYSTEM is providing a co-ordinate system of the world to the other packages.)

A uses dependency only reveals which module is the client module and which is the supplier module. It does not reveal which entities are involved in the client/supplier relationship, nor does it give any information on their visibility.

```
MODULE ImportingModule;
   IMPORT ExportingModuleA;
   FROM   ExportingModuleB IMPORT ...;
   :
END ImportingModule.
```

Figure 6.8: Modula-2 Module Declaration with Two Forms of Imports

Consider the Modula-2 module declaration in Figure 6.8. Both of the modules `ExportingModuleA` and `ExportingModuleB` are providing entities to the module `ImportingModule`. The entities from `ExportingModuleA` are imported in a manner that necessitates qualified referencing of the imported entities, while the entities from `ExportingModuleB` are directly imported thereby allowing simple or unqualified referencing of the imported entities. The module-to-module interconnection graph does not show this difference, instead the module `ImportingModule` is connected to both of the modules `ExportingModuleA` and `ExportingModuleB` by a `uses` dependency, as is shown in Figure 6.9.

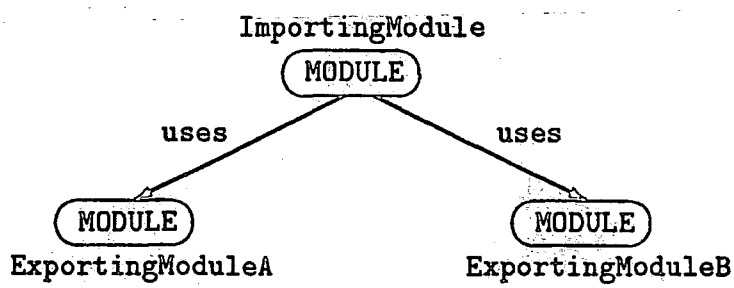


Figure 6.9: The Module-to-Module Interconnection Graph for Figure 6.8

6.2.3 The instantiates-to Dependency

The instantiates-to dependency is the dependency that exists between a generic module and a concrete module. A generic module is a template module that describes the main characteristics of the entities but the description is not complete enough for the entities to be used, e.g., the data type may have been named but not described, or the interface to a routine may have been given but not the algorithmic information. This information is given when a concrete module is created by *instantiating* a generic module. A concrete module is a fully elaborated version of the generic module. It is possible for a generic module to have several different concrete module instances in the same program. With an edge representing an instantiates-to dependency, the start-node denotes the generic module and the stop-node the concrete module.

Generic module instantiation can be performed statically (at compile time) as in Ada and Clu, or dynamically (at run time) as in Eiffel. The inter-module code analysis techniques of this thesis are aimed at static code analysis, and so only static module instantiation will be discussed.

There are three forms of generic module instantiations:

1. *Type instantiations.*

   When type elaboration is performed.

2. *Value instantiations.*

   When the entity associated with a variable or constant is given a value.

3. *Routine instantiations.*

   When algorithmic information is bound to a routine name.

In practice, module instantiation can be a combination of the above forms.

```
set = cluster [t: type] is create, insert, delete, member,
                           size, choose
      where t has equal: proctype (t, t) returns (bool)

      rep = array[t]

      create = ...
      insert = ...
      delete = ...
      member = ...
      size   = ...
      choose = ...
      getind = proc (s: rep, x: t) returns (int)
               i: int:= rep$low(s)
               while i <= rep$high(s) do
                 if t$equal(x, s[i])
                   then return (i)
                 end
                 i:= i + 1
               end
               return (i)
      end getind
end set
```

Figure 6.10: Partial Declaration of a Generic Cluster in Clu

Figure 6.10 gives a partial declaration of the cluster `set` based on the one given by Liskov and Guttag in [99, page 80]. It shows the declaration of a set abstract data type with the operations `create`, `insert`, `delete`, `member`, `size` and `choose`. With this particular cluster, the set abstract data type is implemented by means of an array. This particular information is hidden from user of the cluster `set`. The routine `getind` is private to the cluster `set`. This routine is used by some of the public entities to find the location of a desired element within the array being used to implement the set data structure. The implementation of `getind` is in turn

dependent on the generic parameter t as getind performs a test for equality on elements of type t. This occurs with the statement,

```
t$equal(x, s[i])
```

It is therefore necessary to constrain the set of types that can be used to instantiate the cluster set to the types that provide a routine called equal that takes two arguments of the same type and returns a boolean value. This is denoted by the clause,

```
where t has equal proctype (t, t) returns (bool)
```

The generic module declaration given in Figure 6.10, called a parameterised cluster in Clu, describes a metaclass. The metaclass set has to be instantiated to create class modules that can be used to declare variables. In Clu, the cluster set can be instantiated by statements of the form,

```
intset = set[int]        (6.1)
pset = set[poly]         (6.2)
```

The declaration of the class intset in (6.1) instantiates set with the predefined type int, while the declaration of the class pset in (6.2) instantiates set with the user defined type poly. This type will have been declared as providing a routine called equal thereby satisfying the constraints imposed by the where clause in the declaration of set. The instantiation statements in (6.1) and (6.2) are shown graphically in Figure 6.11. Any instantiations of the class modules intset and pset would also be represented by an instantiates-to dependency. The

instantiates-to dependency does not distinguish between instantiating a metaclass to a class or to another metaclass, because this can be derived from the graph structure as is explained in section 6.3.2.

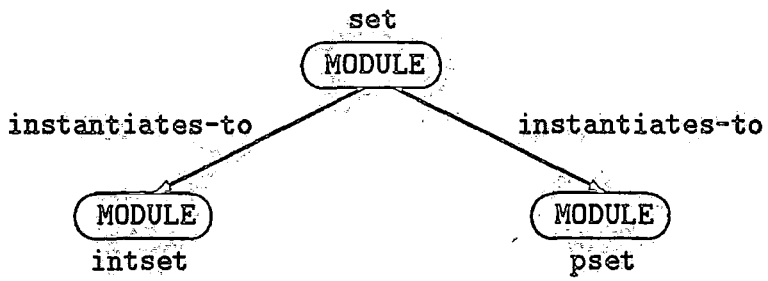


Figure 6.11: Two Instantiations of a Parameterised Cluster

The instantiation of set can be described as being both type and routine instantiation. Type instantiation is needed because the generic parameter for set is a type, and routine instantiation is needed because a particular routine has to be bound to the call to t$equal in getind. In Clu, type elaboration will always be performed on a parameterised cluster, because the only form of generic parameter is a type. Other languages like Ada do not impose this restriction, and so with these languages it is possible to have routine or value instantiations without a type instantiation.

6.2.4 The inherits-from Dependency

The inherits-from dependency applies only to the object-oriented programming languages, where a module can be created as an extension or specialisation of other modules, for example, C++, Eiffel, Simula and Smalltalk-80. Oberon provides an inheritance mechanism, but the inheritance is obtained by means of type

extensions [170] rather than module extensions. As a result the `inherits-from` dependency does not appear in module-to-module interconnection graphs for Oberon programs.

```
class TREE
    export ...
    inherit
        LINKABLE;
        LINKED_LIST;
    :
end -- class TREE
```

Figure 6.12: Declaration of a Module using Multiple Inheritance

There are two forms of inheritance within object-oriented programming languages: **subclassing** which is used in Simula and Smalltalk-80, and **multiple inheritance** which is used in Eiffel and more recently C++. (Originally C++ used subclassing as its inheritance mechanism.) With subclassing, a module is created as an extension of one other module (which itself may be an extension of a module). Multiple inheritance is a generalisation of the subclassing mechanism. With multiple inheritance, a module can be created as a extension of one or more modules (which themselves can be extensions of other modules). Within this scenario, programs written in languages that employ subclassing can be viewed in the same way as programs written in languages that provide multiple inheritance, but which choose to create modules from only one other module. This means that a graph structure that can represent multiple inheritance can also represent subclassing.

Consider the partial Eiffel **class** given in Figure 6.12. This class declaration gives an example of multiple inheritance, where a module is created as an extension

of two modules. The module-to-module interconnection graph for the Eiffel class declaration in Figure 6.12 is given in Figure 6.13.

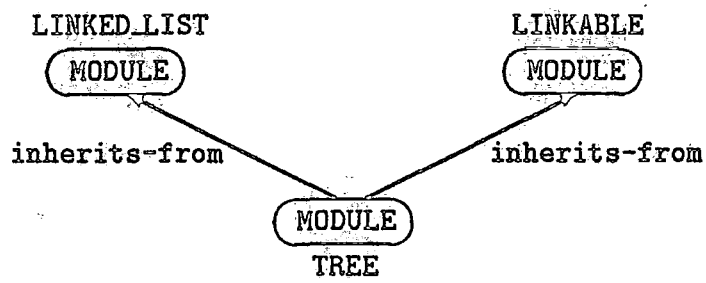


Figure 6.13: The Module-to-Module Interconnection Graph for Figure 6.12

The start-node of the edge that is an `inherits-from` dependency represents the heir module, while the stop-node represents the bequeathing module. When subclassing is used as the inheritance mechanism, the resulting module-to-module interconnection graph for nodes that have an `inherits-from` should have a tree structure. The structure for a system using multiple inheritance should be either a tree or a directed acyclic graph.

In theory it is possible for cyclic structures to exist in a module-to-module interconnection graph for the nodes that have an `inherits-from` dependency. Consider for example the declaration of the two pseudo classes in Figure 6.14. A call to the routine `P1` from the class `Parent-Class` would result in a call to the routine `C1` from the class `Child-Class` which in turn results in a call to `P2` from `Parent-Class`. A cyclic structure with respect to the `inherits-from` dependency is anomalous because the concept of a module being an extension or specialisation of another is obscured. The module-to-module interconnection graph used in the thesis can represent such a cycle, and can therefore help a maintenance programmer detect such anomalous dependencies. More information on the analysis of the module-to-module interconnection graph is given in the following section.

```
CLASS Parent-Class

  INHERIT FROM Child-Class;

  PROCEDURE P1;
  BEGIN
    WriteLn("P");
    Child-Class.C1
  END P1;

  PROCEDURE P2;
  BEGIN
     WriteLn("End")
  END P2;
END Parent-Class.


CLASS Child-Class

  INHERIT FROM Parent-Class;

  PROCEDURE C1;
  BEGIN
    WriteLn("C");
    Parent-Class.P2
  END C1;
END Child-Class.
```

Figure 6.14: Example of a Cyclic Inheritance Declaration

6.3 Analysis of the Module-to-Module Interconnection Graph

The module-to-module interconnection graph shows which modules are dependent on each other, together with information on the nature of the dependency. The module-to-module interconnection graph can be analysed in two ways:

- *Single Dependency Analysis*

  The module-to-module interconnection graph is analysed with respect to a single dependency thereby helping to classify a module.

- *Mixed Dependency Analysis*

  The module-to-module interconnection graph is analysed with respect to two or more dependencies, and this helps to locate modules that have more than one role within a system.

With single dependency analysis, specialised subgraphs of a module-to-module interconnection graph are obtained by performing appropriate δ-slicing operations on the module-to-module interconnection graph. For example, the graph showing the `uses` dependency is obtained by the operation,

$$\delta(\mathrm{G}(\mathcal{N}, \mathcal{E}), \{\texttt{uses}\})$$

The δ-slicing operation ensures that only the nodes involved in a `uses` dependency appear in the resulting graph. Similar operations can be used to obtain the specialised subgraphs for the `local-to`, `instantiates-to` and `inherits-from`

dependencies. In general, the expression,

$$\begin{aligned} &\mathrm{G}(\mathcal{N}, \mathcal{E}) = \\ &\quad \delta(\mathrm{G}(\mathcal{N}, \mathcal{E}), \{\texttt{instantiates-to}\}) \sqcup \delta(\mathrm{G}(\mathcal{N}, \mathcal{E}), \{\texttt{uses}\}) \sqcup \\ &\quad \delta(\mathrm{G}(\mathcal{N}, \mathcal{E}), \{\texttt{local-to}\}) \sqcup \delta(\mathrm{G}(\mathcal{N}, \mathcal{E}), \{\texttt{inherits-from}\}) \end{aligned} \tag{6.3}$$

is true, where $\mathrm{G}(\mathcal{N}, \mathcal{E})$ is a module-to-module interconnection graph. The expression (6.3) cannot always be guaranteed, as $\mathrm{G}(\mathcal{N}, \mathcal{E})$ may contain nodes that are associated with modules which are never used within the system. The node associated with a module that is never used within a system, appears as an isolated node in the module-to-module interconnection graph. To overcome this, the graph resulting from the δ-slicing operation,

$$\delta(\mathrm{G}(\mathcal{N}, \mathcal{E}), \{\texttt{\$ISOLATED\$}\})$$

has to be included in the expression (6.3).

If the graph showing the isolated nodes is non-empty, i.e.,

$$\delta(\mathrm{G}(\mathcal{N}, \mathcal{E}), \{\texttt{\$ISOLATED\$}\}) \neq \phi$$

then this denotes the existence of redundant modules which can be removed from the system without affecting its execution. A module becomes redundant within a system when the services it is providing are no longer required and therefore all the connections to that module have been removed.

With mixed dependency analysis, the graph is analysed with respect to more than one form of dependency. Specialised subgraphs of the module-to-module interconnection graph that show only the dependencies of interest can be obtained

by using the δ-slicing operation. For example,

$$\delta(\mathrm{G}(\mathcal{N}, \mathcal{E}), \{\texttt{uses}, \texttt{inherits-from}\})$$

results in a graph showing the uses and the inherits-from dependencies only.

This thesis concentrates on the analysis with respect to the uses dependency. Therefore, in subsection 6.3.1 modules are classified by analysing the module-to-module interconnection graph with respect to the uses dependency, and subsection 6.3.2 will discuss how the other forms of dependencies can be analysed and mixed dependency analysis performed.

6.3.1 Module Classification via the uses Dependency

With the virtual machine approach to software development, a system is built up by creating software levels. The software at level i provides facilities to the software at level $i + 1$. The software at level 0 provides a software interface to the underlying hardware. Software developed using the virtual machine concept is an example of software that employs the uses dependency. In general, the uses dependency appears in the module-to-module interconnection graph for any system where the concept of one module providing facilities for another is employed. This makes the uses dependency very important, because it can be applied to all the module languages, and software design techniques like information hiding and object-oriented design employ the uses dependency to develop systems.

In this thesis it is suggested that five forms of modules can be identified by analysing the module-to-module interconnection graph with respect to the uses

dependency. They are:

1. *The Specialised Module*

   A module providing a specialised service.

2. *The Terminal Module*

   A low level module within a system.

3. *The Fundamental Module*

   A module that plays a critical role within a system.

4. *The Root Module*

   A module that represent the system or the properties of part of the system.

5. *The Solitary Module*

   A module that is not part of the module-to-module interconnection graph with respect to the uses dependency.

These forms of module are described below.

The **specialised module** is a module that provides a specialised service within a system of module. Such a module can easily be identified because the node associated with the module is the stop-node for an edge that represents a **uses** dependency, and that node is the stop node for only one edge representing a **uses** dependency, i.e., it provides facilities to only one other module. With classical design methods that result in the module-to-module interconnection graph having a tree structure, each module (bar the actual root module) is a specialised module, because it is providing facilities needed by the module associated with the parent node.

The terminal module is a module that is totally self contained, requiring none of the services provided by the other modules in the system. With directed acyclic graphs and trees a terminal module is a module that occurs at the lowest level. A terminal module appears in the module-to-module interconnection graph as a module that is the stop-node of at least one edge that represents the uses dependency but is not the start-node for an edge representing a uses dependency.

The fundamental module is a module that plays a critical role within a system. This form of module appears in the module-to-module interconnection graph as a module that is the stop-node for many edges denoting uses dependencies, and is thus required to support many other modules. Fundamental modules can exist in a system for many reasons. For example, the nature of the service provided by the module may be the reason it is a critical module of a system (e.g., an I/O module); or the system may have been poorly designed or maintained with the result that the module is a potpourri module, providing several logically unrelated services.

The root module is a module that appears to represent either the entire system, or part of the system. A root module appears in the module-to-module interconnection graph in two forms.

1. Uses facilities but provides none, e.g., the program module of Modula-2 program.

2. Uses facilities from a large number of modules and provides some services. This form of dependency highlights a module that requires the services of many modules in order to provide its services to the system. As such, this module tends to represent the modules it uses in a more general form. For example, an I/O module is often constructed from more specialised I/O modules.

The **solitary module** is a module that is not used with any **uses** dependency. The node for this form of module appears in the module-to-module interconnection graph as an isolated node, i.e., a node that is neither the start-node or the stop-node of an edge representing a **uses** dependency. To understand the role of this module within a system, the module-to-module interconnection graph would have to be analysed with respect to the other dependencies recorded.

It is possible for a module to be classified as being of several forms. Consider for example, the structure of the Heads-Up Display given in Figure 6.7 (page 111). The package `WORLD_SYSTEM` can be classified as being both a terminal module, because it requires no services from the other packages, and a fundamental module, because it is providing a service to a large number of packages.

6.3.2 Other Forms of Analysis and Module Classification

When the module-to-module interconnection graph is analysed with respect to the **uses** dependency, the levels of the virtual machines upon which a system is constructed are revealed to the maintenance programmer. The module-to-module interconnection graph can also identify which modules correspond to each virtual machine level. However, the virtual machine hierarchy is not the only technique that is used to structure a system in the module languages. The `local-to`, `instantiates-to` and `inherits-from` dependencies help reveal other system structuring techniques.

The `local-to` dependency can reveal how a particular subsystem is further subdivided into other modules, but these modules provide entities which are logically related, i.e., each module does not properly represent a subsystem. The

`instantiates-to` dependency shows how some modules within a system are particular *instances* of another more general module whose components cannot be used in the system. The `inherits-from` dependency shows another way in which modules are built up from other modules, but this time the components of each of the modules can be used in the system.

Each of these dependencies is considered below.

The `local-to` Dependency

The subgraph of the module-to-module interconnection graph $G(\mathcal{N}, \mathcal{E})$ showing only the `local-to` dependencies can be extracted by applying the following δ-slicing operation,

$$\delta(G(\mathcal{N}, \mathcal{E}), \{\texttt{local-to}\}) \tag{6.4}$$

The graph resulting from (6.4) shows which modules have been further decomposed by the programmer implementing the main module. Unless the local modules are being used to circumvent a perceived language weakness, as is demonstrated on page 109, the graph resulting from (6.4) shows portions of a subsystem that the programmer implementing a module felt required extra protection from accidental misuse. Identifying these portions of a module helps a maintenance programmer identify important parts of a module. This in turn helps a maintenance programmer understand a module.

The `instantiates-to` Dependency

Analysing the module-to-module interconnection graph with respect to the `instantiates-to` dependency can help the maintenance programmer classify modules. The δ-slicing operation,

$$\delta(\mathrm{G}(\mathcal{N}, \mathcal{E}), \{\texttt{instantiates-to}\}) \tag{6.5}$$

extracts a subgraph from $\mathrm{G}(\mathcal{N}, \mathcal{E})$ that shows which modules are connected via an `instantiates-to` dependency.

Let $\mathrm{G}_i(\mathcal{N}_i, \mathcal{E}_i)$ be the graph that results from (6.5). Each proper subgraph within $\mathrm{G}_i(\mathcal{N}_i, \mathcal{E}_i)$ denotes an **instantiation tree**. An instantiation tree is a tree structure where each of the non-terminal nodes is associated with a generic module. A terminal module is normally associated with a concrete module, but it is possible that the terminal module is a generic module. In this case, the module-to-module interconnection graph has helped locate a module (or set of modules) that is not being used within a system, as a generic module cannot be used unless it is instantiated.

If the language in which the system is implemented allows for the dynamic instantiation of a generic module, then the existence of a terminal node in the instantiation tree which is associated with a generic module cannot be used to infer that the module is redundant. The existence of such a node, however, does inform the maintenance programmer that they need to examine the usage of the generic module to see if the module is redundant.

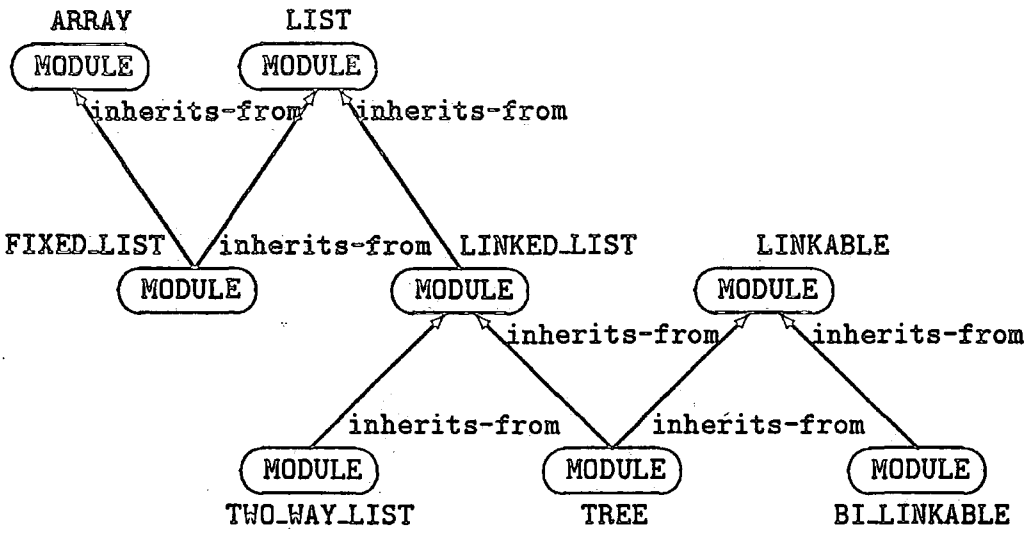


Figure 6.15: The Inheritance Graph for part of the Eiffel Library

The `inherits-from` Dependency

When analysing the module-to-module interconnection graph with respect to the `inherits-from` dependency, the maintenance programmer is in fact analysing **inheritance graphs** which show the modules that are created as extensions or specialisation of other modules. A module-to-module interconnection graph can contain several inheritance graphs. Each inheritance graph is the family tree for the set of modules associated with the nodes in the inheritance graph.

Consider the inheritance graph given in Figure 6.15. This inheritance graph is taken from "Object-Oriented Software Construction" [105, page 246] by Meyer and it shows part of the inheritance graph for the Eiffel library. This graph shows the lineage for the modules involved in the `inherits-from` dependency. For example, the module *TREE* is built up from the modules *LINKABLE* and *LINKED_LIST*, which in turn is built up from the module *LIST*.

The inheritance graph for a system employing subclassing as the inheritance mechanism is a tree structure. This is because each module can only be built-up from one module, but a module can be the base for many modules. This is analogous to the idea that a node in a tree can only have one parent node but can have many children nodes.

In order to obtain the subgraph of a module-to-module interconnection graph that only records the `inherits-from` dependency, the following δ-slicing operation is used,

$$\delta(\mathrm{G}(\mathcal{N}, \mathcal{E}), \{\texttt{inherited-from}\}) \tag{6.6}$$

where $\mathrm{G}(\mathcal{N}, \mathcal{E})$ is the module-to-module interconnection graph being sliced. In order to derive the inheritance graphs contained within a module-to-module interconnection graph, the graph resulting from (6.6) is given as the argument to the *get-proper-subgraphs* function specified on page 99.

Mixed Dependency Analysis

Analysing the module-to-module interconnection graph with respect to particular dependencies, can help a maintenance programmer classify or understand the function that a module provides within a system. With module languages that allow more that one form of dependency between modules, it is often fruitful to analyse a module-to-module interconnection graph with respect to several dependencies in order to ascertain which modules are playing a dual role within a system. The grouping of dependencies used in mixed dependency analysis with often be a language dependent decision as some dependencies or combination of dependencies

will not be possible in some languages.

A useful form of mixed dependency analysis is to analyse a module-to-module interconnection graph with respect to the `uses` and the `inherits-from` dependencies. This form of mixed dependency analysis will be discussed below, but other forms of mixed dependency analysis are possible. For example, it may be useful to analyse the module-to-module interconnection graph with respect to the `inherits-from` and the `instantiates-to` dependencies for programs written in Eiffel, or with respect to the `uses` and the `instantiates-to` dependencies for programs written in Ada or Clu.

The number of dependencies involved in mixed dependency analysis need not be confined to just two, but as the amount of information that the maintenance programmer is going to have to process is likely to be quite large, performing several mixed dependency analysis operation with respect to two dependencies and then combining the results may well prove more fruitful.

When analysing a module-to-module interconnection graph with respect to the `uses` dependency, a hierarchy of virtual machines is revealed. This hierarchy can be analysed and the modules classified by the way they are used within the system. When analysing a module-to-module interconnection graph with respect to the `inherits-from` dependency, a data abstraction hierarchy is obtained. Each level up this hierarchy describes a more specialised abstract data type, or an enlargement of an abstract data type. Analysing a module-to-module interconnection graph with respect to both of these dependencies can provide the maintenance programmer with more information than might be obtained by analysing each dependency separately.

Let $G_u(\mathcal{N}_u, \mathcal{E}_u)$ be the subgraph of a module-to-module interconnection graph showing the `uses` dependencies, and let $G_i(\mathcal{N}_i, \mathcal{E}_i)$ be the subgraph showing the `inherits-from` dependencies. It is possible for the node n_M associated with the module `M` to be in both $G_u(\mathcal{N}_u, \mathcal{E}_u)$ and $G_i(\mathcal{N}_i, \mathcal{E}_i)$ (i.e. $n_M \in \mathcal{N}_u$ and $n_M \in \mathcal{N}_i$). When this occurs, `M` is said to have a dual role within a system, as it appears in two of the hierarchies that describe the system.

In order to determine which modules within a system have a dual role with respect to the `uses` and the `inherits-from` hierarchies, a strict graph intersection operation is used.

$$G_u(\mathcal{N}_u, \mathcal{E}_u) \sqcap G_i(\mathcal{N}_i, \mathcal{E}_i) \tag{6.7}$$

The strict graph intersection operation will not pass on an edge to the resulting graph that does not exist in both the given graphs. As both graphs record different dependencies, they have no edges in common. Therefore, the resulting graph contains no edges. However, it is possible for $G_u(\mathcal{N}_u, \mathcal{E}_u)$ and $G_i(\mathcal{N}_i, \mathcal{E}_i)$ to have nodes in common, and these common nodes will appear in the resulting graph.

If the result of (6.7) is the empty graph, then this indicates that the modules associated with $G_i(\mathcal{N}_i, \mathcal{E}_i)$ may be redundant within the system. In order to confirm this, $G_i(\mathcal{N}_i, \mathcal{E}_i)$ would have to be analysed with the subgraphs of the module-to-module interconnection graph recording the `local-to` and `instantiates-to` dependencies.

6.4 An Example of the Analysis of the Module-to-Module Interconnection Graph

`m2dep` is a program that Sun Microsystems Inc. provide with their Modula-2 system. The `m2dep` program analyses the import lists of Modula-2 modules, and generates the PostScript* commands for drawing a grid which shows the modules that import entities from each other. `m2dep` is written in Modula-2 and its module-to-module interconnection graph is given in Figure 6.16. Since it is composed of global modules only, the module-to-module interconnection graph contains only `uses` dependencies.

By examining the module-to-module interconnection graph for `m2dep`, the modules `Scanner` and `UnixSupport` are identified as being terminal modules of the `m2dep` system. This means that these two modules are low level modules within the `m2dep` system. The module `Scanner` performs lexical analysis, while the module `UnixSupport` provides a Modula-2 interface to the underlying Unix† operating system.

Further analysis of the module-to-module interconnection graph shows that `Scanner` is only used by `ModuleHandling` within the `m2dep` program. Therefore, `Scanner` is classed as being a specialised terminal module within the `m2dep` program. Similarly the module `UnixSupport` is seen to be used by three modules. This means that the services represented by the public entities of `UnixSupport` are used by a relatively large number of the modules of the `m2dep` system. As a result of this, `UnixSupport` is classified as being a fundamental module within the `m2dep`

*PostScript is a registered trademark of Adobe Systems Incorporated
†Unix is a trademark of AT&T

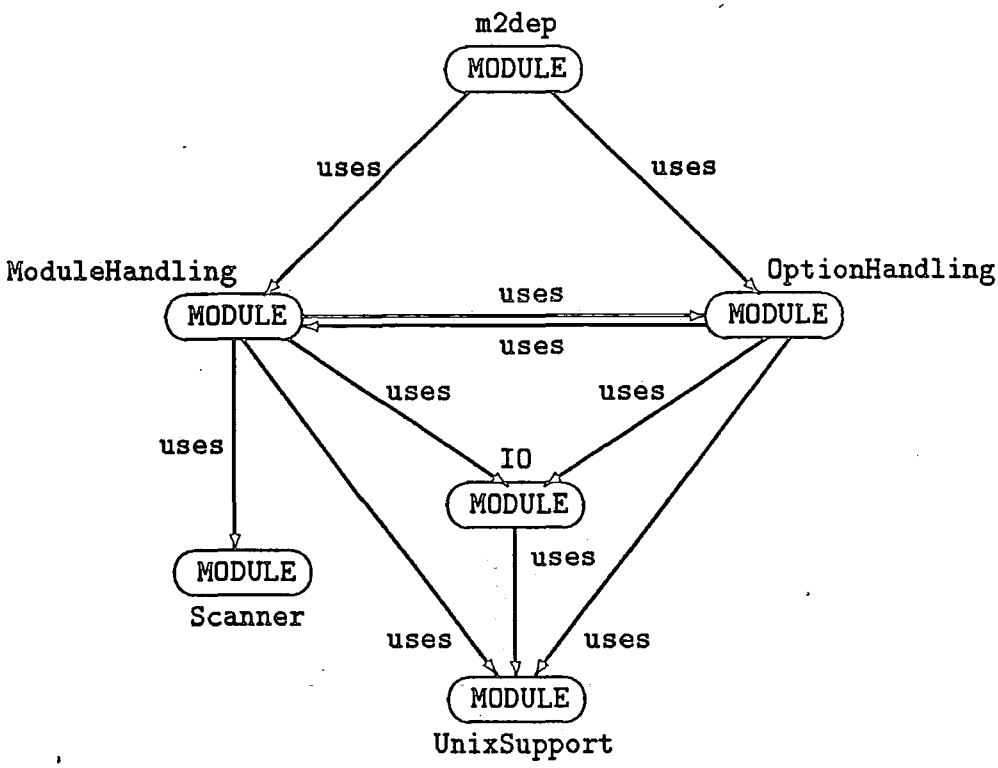


Figure 6.16: Module-to-Module Interconnection Graph for `m2dep`

system. A closer examination of UnixSupport is needed in order to determine if UnixSupport is providing several logically unrelated services, or one service that is important to the m2dep program.

The module IO is not a terminal module, as it uses the module UnixSupport. The module-to-module interconnection graph shows that the module IO is used by two other modules. Within the m2dep system this is a relatively high number of modules (two modules are a third of the modules in the system), and so the module IO can also be classified as a fundamental module. In Chapter 9, techniques for analysing the contents of modules in order to derive the different services are given, and the techniques are demonstrated by using the modules IO and UnixSupport.

The module m2dep is a root module. Using knowledge about the language Modula-2, it can be derived that the root module is the program module that is used to initiate program execution. The module m2dep is therefore a high level module of this system.

The modules ModuleHandling and OptionHandling are mutually dependent, i.e., the module ModuleHandling uses the module OptionHandling and the module OptionHandling uses the module ModuleHandling. This mutual dependency indicates that the m2dep system was not developed using any of the classical software design methods, because such methods cannot produce a mutual dependency with respect to the uses dependency.

Chapter 7

Entity-to-Module Interconnection Graph

7.1 Introduction

An entity-to-module interconnection graph is another form of interconnection graph that shows the dependencies between modules. The entity-to-module interconnection graph gives a more detailed description of the inter-module connections than is given by the module-to-module interconnection graph. In particular, it shows which *entities* are exported from, imported by or inherited by each of the modules of a system.

Just as with module-to-module interconnection graph, the entity-to-module interconnection graph provides a useful form of documentation that is generated

from the program code. This information is especially useful to the maintenance programmer, as it reveals details about the nature of a module.

For example, if a module exports a type and some routines to a module, then the supplier module is likely to be exporting an abstract data type. This cannot be confirmed without having analysed the entity-to-entity interconnection graph associated with the exporting module, as is explained in section 7.3, but this information helps give a maintenance programmer a feel for what sort of service the module is providing within the system.

The module classifications given in Chapter 6 can help identify which modules require closer examination. The entity-to-module interconnection graph can be analysed with respect to the taxonomy given by Booch (see Chapter 3) in order to help determine what form of modules the system is comprised of. The entity-to-module interconnection graph can also be used to help improve the structure of a system by a technique known as **module factoring** which is described in Chapter 9.

In section 7.2, the characteristics of the entity-to-module interconnection graph are identified, and section 7.3 describes how the entity-to-module interconnection graph can be analysed and used to help classify modules and find inconsistent interpretations of a design decision.

7.2 Characteristics of the Entity-to-Module Interconnection Graph

The entity-to-module interconnection graph is a specialised form of interconnection graph that shows the dependencies between modules. The dependencies shown in the entity-to-module interconnection graph are more refined than those in a module-to-module interconnection graph. Whereas the module-to-module interconnection graph shows the dependencies between modules in terms of which modules require each other, the entity-to-module interconnection graph shows the dependencies in terms of which public entities are exported by one module and imported by another.

The nodes of an entity-to-module interconnection graph represent the modules that comprise a system, together with the **global entities** that are involved in the module connections. Normally not all the global entities appear in the entity-to-module interconnection graph as some of them are private entities that cannot be involved in the inter-module connections.

Consider for example the case when two modules are connected by a **uses** dependency. Then the only entities from the supplier module that need appear in the entity-to-module interconnection graph are the entities that the supplier module exports, i.e., its public entities. The connections between the supplier module and the client module show the entities that a supplier module either explicitly exports to a client module, or that a client module requests the use of.

Languages like Ada do not allow a client module to state explicitly which entities are to be imported. Instead, all the entities are imported. With this form

of language, it is more meaningful from the code analysis perspective, for the entity-to-module interconnection graph to record which of the imported entities are actually *used*. Thus, the same dependency will be used to record an entity being selectively imported, in languages like Modula-2, and an imported entity being used, in languages like Ada. This will help simplify the discussion on performing some inter-module code analysis work described in later chapters of this thesis.

Similarly, no distinction will be made between qualified import, where the identifier of an imported entity has to be prefixed by the identifier of the exporting module, and direct import, where an imported entity does not need to be qualified in order to be used in the client module. This reduces the number of dependencies being considered in this thesis. Each of the dependencies given below can be taken to mean that either qualified or direct access to the imported entity is valid.

An edge in an entity-to-module interconnection graph represents the dependencies that can exist between modules and global entities from other modules. The dependencies that can be recorded in an entity-to-module interconnection graph are:

- `injected`

  When a local module exports an entity into the surrounding module.

- `imported`

  When an entity is imported into a module.

- `exported`

  When an entity is provided to a group of named modules.

- `inherited`

  When an entity is inherited from another module.

Each of these dependencies will be discussed in the following subsections.

7.2.1 The injected Dependency

The `injected` dependency indicates that the entity associated with the start-node of an edge is `injected` into the module associated with the stop-node of the edge. An entity is injected into a module if the exporting module is a local module. A local module has to export the entity into the surrounding environment, and this environment is the block containing the local module's declaration. This means that the module associated with this block has acquired an entity it did not explicitly ask for. This form of dependency can exist in languages like Ada and Modula-2, which both contain a local module construct.

```
package body PLANE_TRACKER is
   ...
   package ACTIVE_PLANES is
      procedure ADD(P: PLANE; ID: out PLANE_ID);
      procedure DELETE(ID: out PLANE_ID);
      function  INTERNAL_NAME(ID: PLANE_ID) return PLANE;
   end ACTIVE_PLANES;
   ...
end PLANE_TRACKER;
```

Figure 7.1: Example of Entities being Injected into a Package

Consider the declaration of the local package given in Figure 7.1 (taken from [60, page 261–9]). The package `PLANE_TRACKER` is part of a real-time radar surveillance system that keeps track of plane positions. Each plane being tracked has two names within the system: the external name which is displayed to the user, and the name

used within the code of PLANE_TRACKER. The mapping between the two names is performed by the local package ACTIVE_PLANES. This package injects three entities into PLANE_TRACKER: ADD, which adds a plane to the set being tracked; DELETE, which removes a plane from the set being tracked; and INTERNAL_NAME which raises an exception when trying to use an invalid external name. This dependency is shown in Figure 7.2.

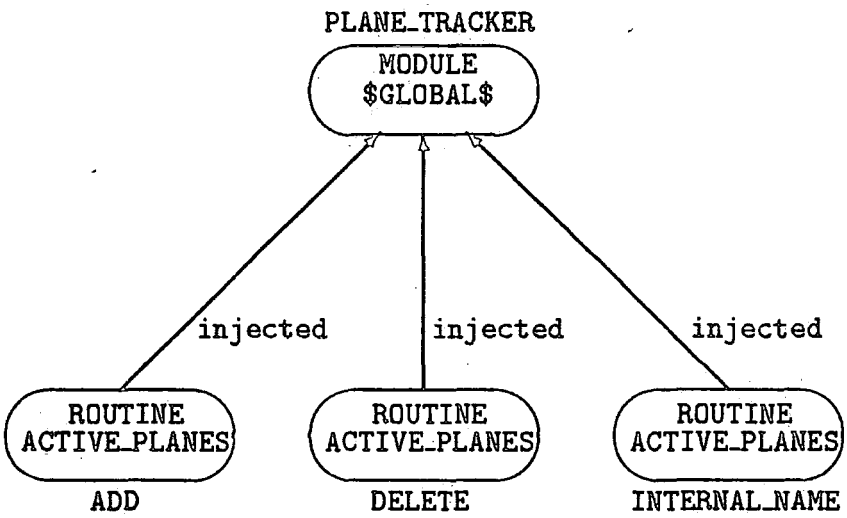


Figure 7.2: The Entity-to-Module Interconnection Graph for Figure 7.1

The nodes for the injected entities record that they are from the package ACTIVE_PLANES. The entity-to-module interconnection graph does not show directly that the package PLANE_TRACKER is dependent on the package ACTIVE_PLANES. This would be shown in the module-to-module interconnection graph. The only time the entity-to-module interconnection graph shows two modules to be directly dependent, is when one of the modules is a public entity, and is involved in an `injected`, `imported` or `exported` dependency.

7.2.2 The imported Dependency

With the injected dependency, the module acquiring an entity does so because a local module exported it into the environment of the encapsulating module. Another way for a module to acquire an entity is to import it. An import can occur between global modules, and between a local module and its encapsulating module, when the local module acquires entities from the encapsulating module.

The imported dependency indicates that a module imports an entity from another module, or in the case of languages like Ada, where a module has to import all the entities, it indicates which of the entities that are available from the supplier module are actually used by the client module.

With the imported dependency, the entity associated with the start-node of an edge is imported or used by the module associated with the stop-node of the edge. This form of dependency is the most common in an entity-to-module interconnection graph and is supported by all the module languages.

Consider again the Heads-Up Display system that is given in Figure 6.7 on page 111. The package WORLD_SYSTEM is used by several packages, including the packages: FLIGHT_PARAMETERS and TARGET_BOX. The specification parts of these three packages are given in Figure 7.3. From this figure it can be seen that the package TARGET_BOX uses the entities: STATE_VECTOR and DIMENSION; while the package FLIGHT_PARAMETERS uses the entities: ALTITUDE and STATE_VECTOR. These dependencies are shown graphically in Figure 7.4.

The entities that are used by the packages FLIGHT_PARAMETERS and TARGET_BOX are shown as being imported entities. The remaining public entities from the pack-

```
package WORLD_SYSTEM is

   type STATE_VECTOR is private;
   type LATITUDE     is private;
   type LONGITUDE    is private;
   type ALTITUDE     is private;
   type DIMENSION    is private;

private
   ...
end WORLD_SYSTEM;

with WORLD_SYSTEM;
package FLIGHT_PARAMETERS is

   type ALTITUDE is new WORLD_SYSTEM.ALTITUDE;
   type HEADING  is new WORLD_SYSTEM.STATE_VECTOR;

   task COUPLER is
      entry STATUS (THE_ALTITUDE : out WORLD_SYSTEM.ALTITUDE;
                    THE_HEADING  : out WORLD_SYSTEM.ALTITUDE);

end FLIGHT_PARAMETERS;

with WORLD_SYSTEM;
package TARGET_BOX is

   type CENTRE is new WORLD_SYSTEM.STATE_VECTOR;
   type SIZE   is new WORLD_SYSTEM.DIMENSION;

   task COUPLER is
      entry STATUS (THE_CENTRE : out WORLD_SYSTEM.STATE_VECTOR;
                    THE_SIZE   : out WORLD_SYSTEM.DIMENSION);

end TARGET_BOX;
```

Figure 7.3: Package Specification for Showing imported Dependencies

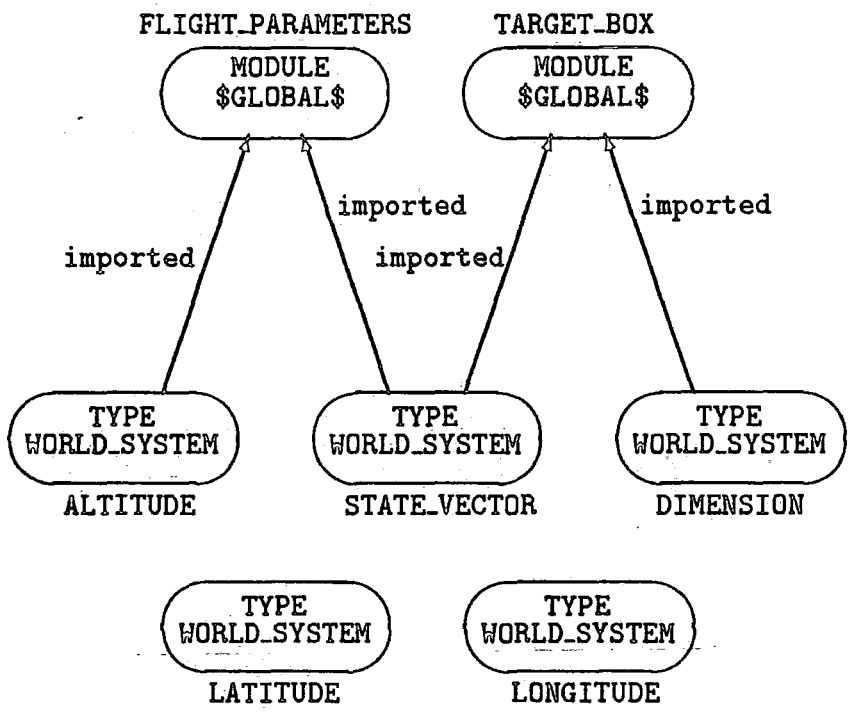


Figure 7.4: The Entity-to-Module Interconnection Graph for Figure 7.3

age WORLD_SYSTEM are shown as being isolated in entity-to-module interconnection graph because they are not used.

7.2.3 The exported Dependency

The exported dependency is applicable only to languages like Eiffel that allow a module to state to which other modules the public entities are being exported. The exported dependency is applied only to those entities that are exported to a named module. Entities that are exported and made generally available to any global module that wishes to import them are not the subject of this dependency.

```
class LINKABLE
    export change_value{LINKED_LIST}, value{LINKED_LIST},
           change_right{LINKED_LIST}, right
    ...
end -- LINKABLE
```

Figure 7.5: An Eiffel class with Selective Export

With the exported dependency, the entity associated with the start-node of the edge is exported to the module associated with the stop-node. Consider for example, the Eiffel class declaration given in Figure 7.5.

The entities: *value*, *change_value* and *change_right* are selectively exported to the module *LINKED_LIST*, while the entity *right* is exported normally. The entity-to-module interconnection graph for this declaration is given in Figure 7.6. The entity *right* appears as an isolated node because it is exported normally and is therefore not the subject of an exported dependency.

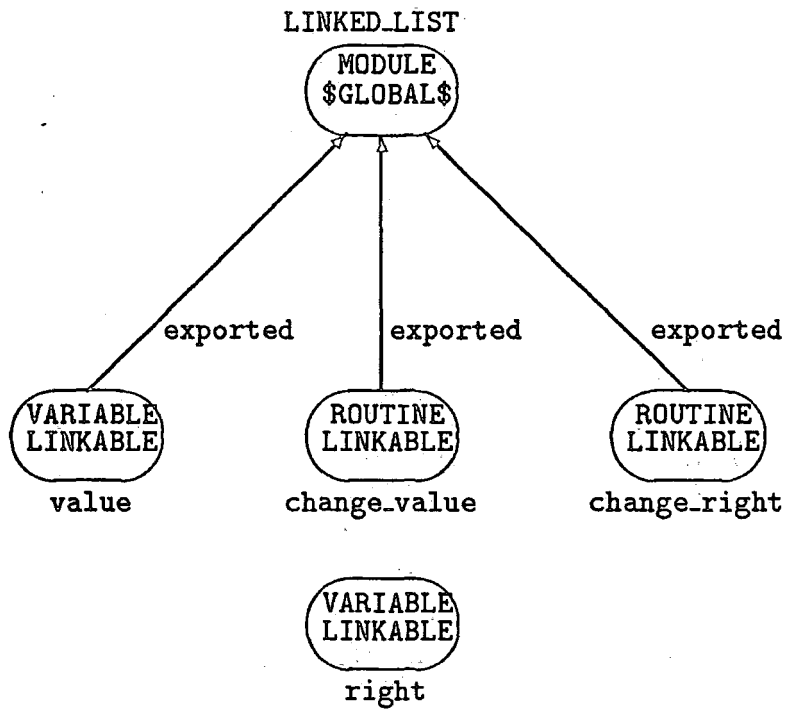


Figure 7.6: The Entity-to-Module Interconnection Graph for Figure 7.5

7.2.4 The inherited Dependency

With inheritance, a class module is created as an extension or specialisation of another class module. Consider for example the following Eiffel class declaration.

class *Q* export ...
inherit *P*
...
end -- class *Q*

This declares the class *Q* to be an extension or specialisation of the class *P*. If the class *P* declares the entities *ent1* and *ent2*, and the class *Q* declares the entities *ent3* and *ent4*, then the class *Q* has the entities: *ent1*, *ent2*, *ent3* and *ent4*. This is because the entities declared in the class *P* are inherited by the class *Q*. Figure 7.7 gives the entity-to-module interconnection graph that shows the class *Q* inheriting the entities *ent1* and *ent2*.

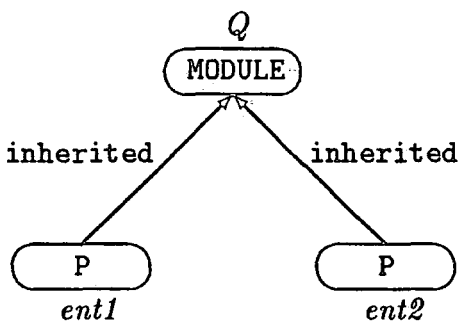


Figure 7.7: The Entity-to-Module Interconnection Graph Showing Inheritance

The form of inheritance shown in Figure 7.7 corresponds to the idea of class *Q* being an extension of class *P*. The class *Q* has all the entities in class *P* and

some extra entities. A class module can also be regarded as a specialisation of another module, and this effects which entities appear in the entity-to-module interconnection graph showing the `inherited` dependency.

When a class module is created as a specialisation of another module, this commonly involves redeclaring entities that are inherited. This redeclaring of an inherited entity is called **overriding**.

Consider the scenario given above where the class *P* declares the entities *ent1* and *ent2*, but this time the class *Q* overrides the declaration of *ent2* to make it more appropriate to the task being performed. Then in the entity-to-module interconnection graph, given in Figure 7.8, the entity *ent2* is not shown as being inherited, because the entity *ent2* is declared in *Q*.

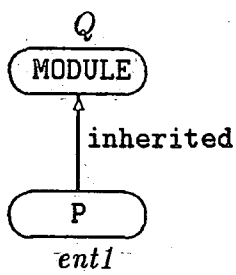


Figure 7.8: The Entity-to-Module Interconnection Graph Showing Inheritance with Overriding

With Simula, it is possible for a module to refuse to bequeath an entity, by declaring the entity hidden. When this occurs, the entity-to-module interconnection graph does not record the hidden entities as being part of an inherited dependency.

7.3 Analysis of the Entity-to-Module Interconnection Graph

The information recorded in the entity-to-module interconnection graph can be used for various types of software maintenance activities. In this section, two techniques for analysing a entity-to-module interconnection graph are described that reveal different kinds of information about a system.

In subsection 7.3.1 the entity-to-module interconnection graph is analysed with respect to the `imported` and `exported` dependencies, in order to determine any anomalous dependencies between modules. In subsection 7.3.2 the entity-to-module interconnection graph is analysed to classify modules according to the taxonomies of modules given by Booch. In this way modules are further classified, giving a maintenance programmer more details about the role of a module within a system by determining the service that the module is providing. Note that this is being done without the maintenance programmer having to examine the implementation details of a particular module.

7.3.1 Anomaly Detection

With the `injected` dependency, the entity exported by a local module is automatically imported by the module containing the local module. This is not the case with entities involved in an `exported` dependency. It is therefore possible for a module to export an entity to another module, but for that module not to import it. In Chapter 3 it is shown how, as part of the design phase, the interface for each module is determined. If therefore, there exists within a system a module which

exports an entity to another module, but that module does not import the entity, then there is an inconsistency between the way the programmers are implementing a design decision. Detecting this form of inconsistency within a system is a form of anomaly detection that can be performed by analysing the entity-to-module interconnection graph for a system.

To detect this form of anomaly, the entity-to-module interconnection graph has to be analysed with respect to both the `exported` and `imported` dependencies. The appropriate graph for this analysis can be extracted from $G(\mathcal{N}, \mathcal{E})$, the entity-to-module interconnection graph for a system by the following δ-slicing operation,

$$\delta(G(\mathcal{N}, \mathcal{E}), \{\texttt{exported}, \texttt{imported}\}) \tag{7.1}$$

Let $G_{ei}(\mathcal{N}_{ei}, \mathcal{E}_{ei})$ denote the graph resulting from (7.1).

The function specified below can be used to determine if there exists an anomalous dependency.

$$\begin{aligned}
&\mathit{exist\text{-}anomaly} : \mathit{Graph} \rightarrow \mathbb{B} \\
&\mathit{exist\text{-}anomaly}(\mathit{mk\text{-}Graph}(\mathit{nodes}, \mathit{edges})) \quad \triangleq \\
&\quad \exists \mathit{e1} \in \mathit{edges} \cdot \\
&\qquad \mathit{dependency}(\mathit{e1}) = \texttt{exported} \\
&\qquad \neg(\exists \mathit{e2} \in \mathit{edges} \cdot \\
&\qquad\quad \mathit{dependency}(\mathit{e2}) = \texttt{imported} \wedge \\
&\qquad\quad \mathit{start\text{-}node}(\mathit{e1}) = \mathit{start\text{-}node}(\mathit{e2}) \wedge \\
&\qquad\quad \mathit{stop\text{-}node}(\mathit{e1}) = \mathit{stop\text{-}node}(\mathit{e2}))
\end{aligned}$$

The following call of this function, would detect any anomalous dependencies within $G_{ei}(\mathcal{N}_{ei}, \mathcal{E}_{ei})$

$$exist\text{-}anomaly(G_{ei}(\mathcal{N}_{ei}, \mathcal{E}_{ei}))$$

7.3.2 Module Classification

When analysing the module-to-module interconnection graph with respect to the **uses** dependency, modules are classified according to their apparent usage within a system. With the entity-to-module interconnection graph, the maintenance programmer is given information on which entities are imported by or exported to the modules of a system.

Some languages, like Extended Pascal, allow a module to explicitly provide several client views, while other languages like Ada and Modula-2 allow only one client view to be explicitly declared. By providing only one client view of a module, the fact that the supplier module may be providing a different service to each of the client modules is obscured. This makes the maintenance programmer's task of understanding a module more difficult, because the maintenance programmer has first to determine that a module has several client views. Analysing the entity-to-module interconnection graph with respect to the exporting and importing of entities can help a maintenance programmer derive the actual client views of a supplier module. This information can then be used by a maintenance programmer to start to classify a module according to Booch's taxonomy.

Deriving the Client Views

The $\alpha\beta$-slicing operation can be used to derive the subgraph of $G_{ei}(\mathcal{N}_{ei}, \mathcal{E}_{ei})$ that shows the client view between a named supplier module and a named client module. Consider for example, the $\alpha\beta$-slicing operation

$$_{source=\texttt{SupplierModule}}\|_{\xi}(G_{ei}(\mathcal{N}_{ei}, \mathcal{E}_{ei}), \langle \xi, \texttt{ClientModule} \rangle) \tag{7.2}$$

This $\alpha\beta$-slicing operation returns the graph that shows the public entities from `SupplierModule` that are either exported to or imported by `ClientModule`. This is because the α constraint,

$$source=\texttt{SupplierModule}$$

ensures that, in the graph resulting from (7.2), all the edges have as a start-node a node with the value `SupplierModule` stored in the label attribute *entity-source*. This means that the start-node of each edge, is associated with an entity from `SupplierModule`. Naming the module `ClientModule` as the second element of the slicing criterion argument, ensures that the node associated with this module is the stop-node for all the edges in the resulting graph.

In order to derive all the client views of a module, a function satisfying the following specification can be used.

$$get\text{-}client\text{-}views : Graph \times Source \rightarrow \text{ set of } Graph$$

$$get\text{-}client\text{-}views(graph, supplier\text{-}module) \triangleq$$
$$\text{let } moduleset = \{m \mid m \in nodes(graph) \wedge$$
$$\exists e \in edges(graph) \cdot m = stop\text{-}node(e)\} \text{ in}$$
$$\{_{source=supplier\text{-}module}\|_{\xi}(graph, \langle \xi, m \rangle) \mid m \in moduleset\}$$

In order to derive all the client views of the module `SupplierModule`, the function *get-client-views* is used in the following manner,

$$get\text{-}client\text{-}views(\mathrm{G}_{ei}(\mathcal{N}_{ei}, \mathcal{E}_{ei}), \texttt{SupplierModule})$$

Analysing a Client View

In order to analyse a client view of a module so that it is possible to start classifying the module in terms of Booch's taxonomy, it is necessary to consider the classes of the entities in the client view.

Module Classification	Classes of Entities That can Occur
Named Collection of Declarations	Variables, Types
Group of Related Program Units	Constants, Routines, Modules
Abstract Data Type	Constants, Routines, Types (a Type and a Routine must occur)
Abstract-State Machine	Any (but must contain a Routine)

Table 7.1: Classes of Entities Associated with Booch's Taxonomy

Table 7.1 gives a breakdown of the classes of entities associated with each of the module classifications in Booch's taxonomy. By examining the classes of entities in the client view, it is possible to see which of Booch's classifications the

module can conform to. It is possible for a module to conform to several of the classifications. For example, if a module has routine and type entities then the module could be classified as providing an abstract data type, a group of related program units and a named collection of declarations. The actual classification can be obtained by analysing the entity-to-entity interconnection graph, but the maintenance programmer could dismiss some of these classifications by scanning the relevant modules. For example, if the module contains no state variables, then it cannot be an abstract-state machine; or if the type entities are not used to declare parameters for the routines then the module is unlikely to be providing an abstract data type.

Chapter 8

Entity-to-Entity Interconnection Graph

8.1 Introduction

The entity-to-entity interconnection graph differs from the module-to-module interconnection graph and the entity-to-module interconnection graph in that the dependencies recorded are not those that exist between modules. Instead, the entity-to-entity interconnection graph shows the dependencies that exist between the global entities of a module. As a result, a system that comprises of η modules, has η entity-to-entity interconnection graphs.

The entity-to-entity interconnection graph shows the dependencies between all the global entities of a module. As a result it can be used by a maintenance pro-

grammer to determine the dependencies that exist between the public and private entities of a module. This helps classify a module.

In section 8.2, the characteristics of the entity-to-entity interconnection graph are identified, and section 8.3 describes how the entity-to-entity interconnection graph can be analysed and used to help classify modules.

8.2 Characteristics of the Entity-To-Entity Interconnection Graph

The entity-to-entity interconnection graph for a module shows the dependencies between the global entities of the module. The entity-to-entity interconnection graph for a module M can be extracted from a general interconnection graph $G_{ig}(\mathcal{N}_{ig}, \mathcal{E}_{ig})$ by an $\alpha\beta$-slicing operation of the form,

$$_{block=0 \wedge source=M} \| _{block=0 \wedge source=M} (G_{ig}(\mathcal{N}_{ig}, \mathcal{E}_{ig}), \langle \xi, \xi \rangle)$$

With the entity-to-entity interconnection graph for a module, the nodes are associated with the global entities of the module, and an edge between two nodes records that the entities associated with the start-node and stop-node are dependent. Some of the dependencies recorded in the entity-to-entity interconnection graph will be discussed in the following subsections.

8.2.1 The delimited-by Dependency

The delimited-by dependency is associated with an edge of the form,

$$\left(type, \begin{matrix} constant \\ routine \end{matrix}\right)$$

Where the stop-node represents a constant or function entity that is used to mark the upper and lower bounds of the subrange type associated with the start-node.

```
constant C: INTEGER:= 2;

function F: return INTEGER is
begin
  return 10;
end F;

type T1 is range C..2*C;
type T2 is range 1..F;
type T3 is range C..F;
```

Figure 8.1: The Declaration of Range Types in Ada

Consider the fragment of Ada code in Figure 8.1. In this figure, three range types are declared. The range T1 uses the constant entity C to delimit both the upper and lower bounds, while the range T2 uses the constant function F to delimit the upper bound. The range type T3 uses the constant C to delimit the lower bound of the range, while the function F delimits the upper bound of the range. The entity-to-entity interconnection graph showing these dependencies is given in Figure 8.2. The range type T1 is delimited twice by the constant C, but the entity-to-entity

interconnection graph only shows one occurrence of the dependency `delimited-by`. This is because it was decided that an interconnection graph should only record the existence of a dependency and not the number of times a dependency occurs.

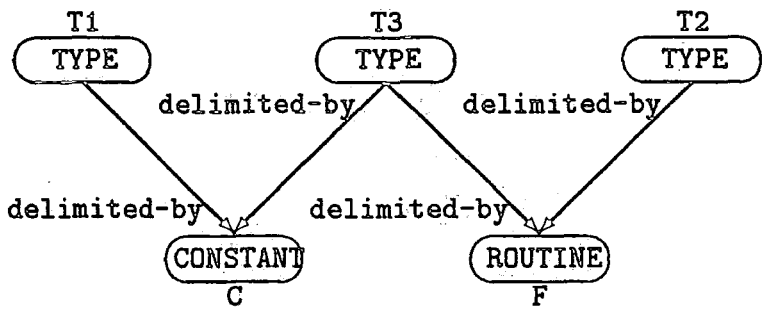


Figure 8.2: The Entity-To-Entity Interconnection Graph for Figure 8.1

8.2.2 The `of-type` Dependency

The edge representing an `of-type` dependency is of the form,

$$\left(\begin{array}{c} constant, \\ type, \\ variable, \\ routine, \end{array} \right) type$$

With an edge representing the `of-type` dependency, the start-node is associated with an entity whose type is the type entity associated with the stop-node of the edge.

Figure 8.3 gives a Modula-2 program module containing entity declarations. The entity-to-entity interconnection graph for this program module is given in Figure 8.4. The edges associated with the variable `V` show that a node can be

```
MODULE EntityDeclaration;

  TYPE T1 = [1..10];
       T2 = RECORD
              Field: T1
            END; (* T2 *)

  VAR V: RECORD
           Field1: T1;
           Field2: T2
         END;

  PROCEDURE F(): T1;
  BEGIN
    RETURN 5
  END F;

BEGIN
  ...
END EntityDeclaration;
```

Figure 8.3: Entity Declaration for Showing of-type Dependencies

the subject of more than one `of-type` dependency. The type associated with the variable `V` is an anonymous record type with a field of type `T1` and another of type `T2`. Thus the variable `V` is dependent on both the types `T1` and `T2`.

If a routine entity is the subject of an `of-type` dependency then the routine must be a function. This is because a function stands for a computed result as well as a computation. And so a function has to have an associated type.

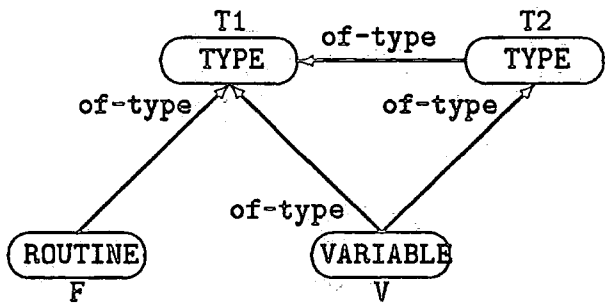


Figure 8.4: The Entity-To-Entity Interconnection Graph for Figure 8.3

8.2.3 The parameter-of-type Dependency

The `parameter-of-type` dependency is associated with an edge of the form,

$$\begin{pmatrix} routine, \\ type, \end{pmatrix} type$$

A `parameter-of-type` dependency shows that the entity associated with the start-node of an edge has at least one parameter of the type associated with the stop-node of the edge.

With most languages, a `parameter-of-type` dependency is confined to edges where the start-node is associated with a routine entity, but some languages like Modula-2 and Oberon allow for routines types. With languages like these, it is possible to have a `parameter-of-type` dependency between type entities.

```
TYPE T1 = CARDINAL;
     ProcType = PROCEDURE(T1);

PROCEDURE P(para1: T1; para2: ProcType);
```

Figure 8.5: Entity Declarations for Showing `parameter-of-type` Dependencies

Figure 8.5 gives Modula-2 declarations of routine and type entities. These declarations are represented by `parameter-of-type` dependencies in the entity-to-entity interconnection graph given in Figure 8.6

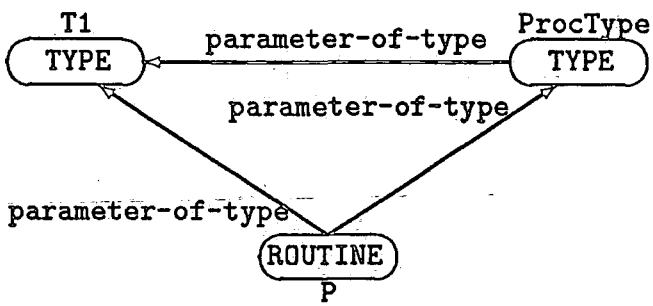


Figure 8.6: The Entity-To-Entity Interconnection Graph for Figure 8.5

8.2.4 The *used-within* Dependencies

The *used-within* dependencies are the dependencies that record a global entity being used within a global routine. Four examples of this form of dependency are:

- `uses-constant`

  This dependency is associated with an edge of the form

  (*routine*, *constant*)

- `uses-type`

  This dependency is associated with an edge of the form

  (*routine*, *type*)

- `uses-variable`

  This dependency is associated with an edge of the form

  (*routine*, *variable*)

- `invokes`

  This dependency is associated with an edge of the form

  (*routine*, *routine*)

Each of these dependencies show that the entity associated with the stop-node of an edge is used within the routine associated with the start-node of the edge. The way in which these entities are used within the routine is not recorded with these dependencies.

Consider for example the entity declarations in Figure 8.7. This figure gives the Modula-2 declaration of two global routines, a global constant, a global type and assorted local entities within the two global routines. The entity-to-entity interconnection graph for these declarations is given in Figure 8.8.

Within the routine `SillyReader`, the constant `MaxString` is used in two different ways. Firstly, it is used to delimit the upper bound of the local type `String`, and secondly, it is used as a sentinel value. These different uses have not been recorded by different dependencies because the code analysis techniques described

```
CONST MaxString = 20;

TYPE Value = INTEGER;

PROCEDURE SillyReader;
  TYPE String = ARRAY [1..MaxString] OF CHAR;
  VAR  i : CARDINAL;
       ch: CHAR;
BEGIN
  WHILE i<= MaxString DO
    ReadChar(ch)
  END
END SillyReader;

PROCEDURE GlobalRoutine;
  PROCEDURE LocalRoutine(para: Value): Value;
  BEGIN
    RETURN MaxString
  END LocalRoutine;
BEGIN
  SillyReader
END GlobalRoutine;
```

Figure 8.7: Entity Declarations for Showing *used-within* Dependencies

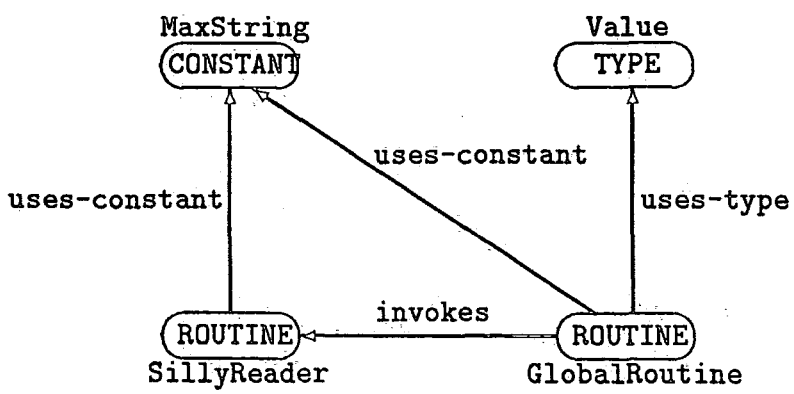


Figure 8.8: The Entity-To-Entity Interconnection Graph for Figure 8.7

in this thesis do not need information on how an entity is being used within a routine. If this information were desired, then other dependencies could be recorded in the entity-to-entity interconnection graph, e.g., the `delimits-local-type` and `sentinel-value` dependencies could be refinements of the `uses-constant` dependency, and `defines` and `references` could be refinements of the `uses-variable` dependency.

Similarly for the routine `LocalRoutine` declared in the routine `GlobalRoutine`. The type entity `Value` is used to declare the parameter of `LocalRoutine` as well as the resulting type, but these different uses are not shown in the entity-to-entity interconnection graph.

The constant `MaxString` is not used by the routine `GlobalRoutine` directly, but the routine `LocalRoutine` does. As a result, the entity-to-entity interconnection graph shows `GlobalRoutine` to be dependent on `MaxString`. This is because a global routine is credited with being dependent on a global entity if this entity is used by a routine local to the global routine. This is because the entity-to-entity interconnection graph of a module only shows the dependencies between global entities. If a global routine was not credited with a dependency on a global entity held by one of its local routines then the existence of dependencies between some entities could be missed and erroneous analysis performed.

8.3 Analysis of the Entity-To-Entity Interconnection Graph

Analysis of the entity-to-entity interconnection graphs is normally performed in conjunction with the analysis of the entity-to-module interconnection graph. By analysing the entity-to-module interconnection graph a maintenance programmer gains information on the client/supplier relationship between modules. In order to fully establish the relationship between a client module and a supplier module, the entity-to-entity interconnection graph has to be analysed. As the entity-to-entity interconnection graph will determine the dependencies between the entities in the client view and establish any dependencies that may exist between these public entities and the private entities of the supplier module.

The entity groups in a entity-to-entity interconnection graph $G(\mathcal{N}, \mathcal{E})$ can be obtained by the following function call,

$$\textit{get-proper-subgraphs}(G(\mathcal{N}, \mathcal{E}))$$

where *get-proper-subgraphs* is the function specified on page 99. The result of this function call is the set of proper subgraphs of $G(\mathcal{N}, \mathcal{E})$. Each proper subgraph represents an entity group. These entity groups can be used to help understand a particular client/supplier relationship.

Let $\mathcal{N}_{cv}$ be the set of nodes associated with the entities in a client view, and let $G_i(\mathcal{N}_i, \mathcal{E}_i)$ be one of the proper subgraphs obtained from the above call of *get-proper-subgraphs*. There are four possible relations between the set of nodes representing the client view and the set of nodes in $G_i(\mathcal{N}_i, \mathcal{E}_i)$:

- $\mathcal{N}_{cv} \cap \mathcal{N}_i = \{\}$
- $\mathcal{N}_{cv} = \mathcal{N}_i$
- $\mathcal{N}_{cv} \subset \mathcal{N}_i$
- $\mathcal{N}_{cv} \supset \mathcal{N}_i$

The relation $\mathcal{N}_{cv} \cap \mathcal{N}_i = \{\}$ says the entity group associated with $G_i(\mathcal{N}_i, \mathcal{E}_i)$ is unrelated to the entities in the client view. Therefore the entities in this group can be disregarded when analysing a module with respect to the client view associated with the set of nodes $\mathcal{N}_{cv}$.

The relations $\mathcal{N}_{cv} = \mathcal{N}_i$ and $\mathcal{N}_{cv} \subset \mathcal{N}_i$ show that the entity group associated with $G_i(\mathcal{N}_i, \mathcal{E}_i)$ contains all the entities that can affect those in the client view. The relation $\mathcal{N}_{cv} = \mathcal{N}_i$ shows that the entities in the client view do not require any of the module's private entities, whereas the relation $\mathcal{N}_{cv} \subset \mathcal{N}_i$ shows that the entities in the client view are dependent on some private entities. If one of these two relationships is true then all the other entity groups can be disregarded, as they cannot relate to the client view because all the entity groups are independent of each other. If a module only has one client view, then the entity groups that are independent of those in the client view consist of redundant entities.

Finally, the relation $\mathcal{N}_{cv} \supset \mathcal{N}_i$ shows that a client view is dependent on more that one entity group, as the entities in the client view are not contained within a single entity group.

As an example, consider the module `IntStack` given in Figures 8.9 and 8.10. Figure 8.9 gives the definition module and Figure 8.10 gives the associated implementation module. The module `IntStack` provides a stack abstract data type

```
DEFINITION MODULE IntStack;

  TYPE StackType;

  PROCEDURE Create(VAR stack: StackType);
  PROCEDURE Pop(VAR stack: StackType);
  PROCEDURE Push(elem: INTEGER; VAR stack: StackType);
  PROCEDURE NumOfStacks(): CARDINAL;
  PROCEDURE NumOfPops(): CARDINAL;
  PROCEDURE NumOfPushes(): CARDINAL;

END IntStack.
```

Figure 8.9: The Definition Module for `IntStack`

for integers. The number of times each of the stack operations `Create`, `Pop` and `Push` is used is recorded. Figure 8.11 gives the entity-to-entity interconnection graph for `IntStack`. The dependency `p-of-type` that appears in Figure 8.11 is an abbreviation for the `parameter-of-type` dependency.

A possible client view for this module is the entire definition module in Figure 8.9. Let $\mathcal{N}_{cv}$ be the set of nodes that are associated with these public entities. The entities groups obtained by the function call

$$\textit{get-proper-subgraphs}(\mathrm{G}_s(\mathcal{N}_s, \mathcal{E}_s))$$

are $\mathrm{G}_1(\mathcal{N}_1, \mathcal{E}_1)$ (given in Figure 8.12) and $\mathrm{G}_2(\mathcal{N}_2, \mathcal{E}_2)$ (given in Figure 8.13).

In this example, $\mathcal{N}_{cv} \subset \mathcal{N}_2$. As a result, the entities associated with $\mathrm{G}_1(\mathcal{N}_1, \mathcal{E}_1)$ are not relevant to the analysis of `IntStack` with respect to the chosen client view. As this client view consists of all of `IntStack`'s public entities, then the entities

```
IMPLEMENTATION MODULE IntStack;
  FROM Storage IMPORT ALLOCATE, DEALLOCATE;
  TYPE StackType = POINTER TO StackElem;
       StackElem = RECORD
                     Data: INTEGER; Link: StackType
                   END; (* RECORD StackElem *)
  VAR CreateCounter, PopCounter, PushCounter: CARDINAL;
      Unused: BOOLEAN;
  PROCEDURE Create(VAR stack: StackType);
  BEGIN stack:= NIL; INC(CreateCounter) END Create;
  PROCEDURE NumOfStacks(): CARDINAL;
  BEGIN RETURN CreateCounter END NumOfStacks;
  PROCEDURE Pop(VAR stack: StackType);
    VAR P: StackType;
  BEGIN
    P:= stack; stack:= stack↑.Link; DISPOSE(P);
    INC(PopCounter)
  END Pop;
  PROCEDURE NumOfPops(): CARDINAL;
  BEGIN RETURN PopCounter END NumOfPops;
  PROCEDURE Push(elem: INTEGER; VAR stack: StackType);
    VAR newStackElem: StackType;
    PROCEDURE CreateStackElem(elem: INTEGER; VAR newElem: StackType);
      VAR P: StackType;
    BEGIN
      NEW(P); P↑.Data:= elem; P↑.Link:= NIL;
      newElem:= P
    END CreateStackElem;
  BEGIN
    CreateStackElem(elem, newStackElem);
    newStackElem↑.Link:= stack; stack:= newStackElem;
    INC(PushCounter)
  END Push;
  PROCEDURE NumOfPushes(): CARDINAL;
  BEGIN RETURN PushCounter END NumOfPushes;
BEGIN
  CreateCounter:= 0; PopCounter:= 0; PushCounter:= 0
END IntStack.
```

Figure 8.10: The Implementation Module for IntStack

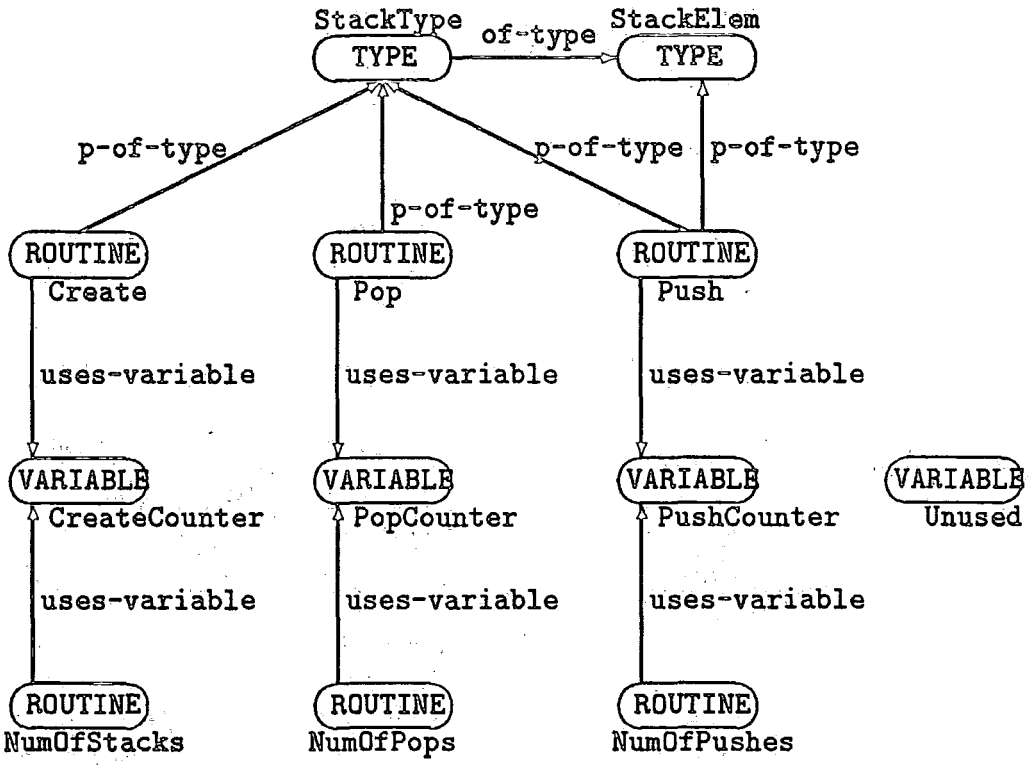


Figure 8.11: The Entity-To-Entity Interconnection Graph for IntStack

Figure 8.12: The Entity Group Associated with $G_1(\mathcal{N}_1, \mathcal{E}_1)$

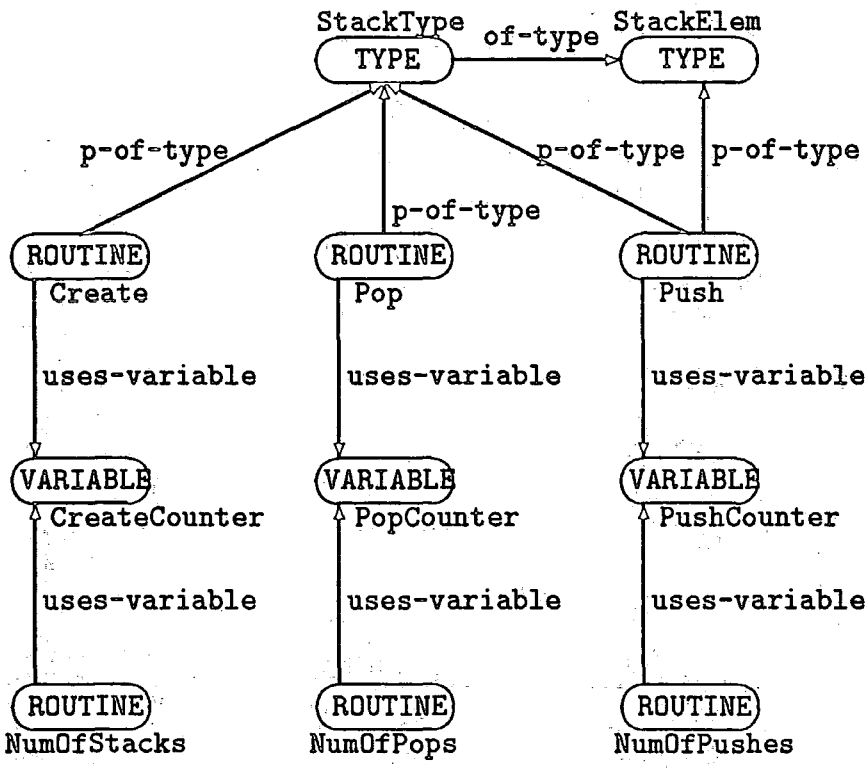


Figure 8.13: The Entity Group Associated with $G_2(\mathcal{N}_2, \mathcal{E}_2)$

associated with the nodes in $\mathcal{N}_1$ must be redundant, i.e., Unused is a redundant entity.

The entity group associated with $G_2(\mathcal{N}_2, \mathcal{E}_2)$ can be classified by examining the classes of entities associated with a particular subgraph. For example, a subgraph associated with routine and constant entities only would represent a collection of routines as a constant entity can be considered as a value function. Similarly a subgraph associated with an entity group consisting of routine, constant and type entities would represent an abstract data type. By examining the dependencies in the subgraph, the abstract data type can be classified according to the classification given by Embley and Woodfield [54, 55]. The entity group associated with $G_2(\mathcal{N}_2, \mathcal{E}_2)$ contains variable entities, therefore the entity group represents an abstract-state machine.

Chapter 9

Module Factoring

9.1 Introduction

Parnas [119] suggests that a module should be considered to be a "responsibility assignment", where related entities are grouped together (see Chapter 3 for a discussion of this definition). By viewing a module in this way, entities that are logically related within a system would be grouped together in a single module. Booch [16] gives a taxonomy of modules in terms of the service that the module provides to the system. With Booch's taxonomy, a module is classed as being one of the following:

- Named Collection of Declarations
- Group of Related Program Units

- Abstract Data Type
- Abstract-State Machine

In Chapter 3 (page 39) we saw that Booch's taxonomy represents an idealised view of the use of modules. In practice, a module is often a combination of these classifications. Such modules are referred to as *potpourri modules.*

The existence of potpourri modules within a system add considerably to the complexity of the system. With potpourri modules, a maintenance programmer must first determine what services a module is providing, and then which of those services, if any, is important to the maintenance activity being performed.

To aid this task, we propose a technique known as *module factoring* [22]. The objective of module factoring, is to determine the different services that a module is providing according to Booch's taxonomy, and to establish all the entities that comprise each of the services. A maintenance programmer can then use this information to either decompose a given module into smaller modules, each of which performs a distinct task, or else the module can be left as it is but the different services and the entities that comprise each of the services can be documented. In this way knowledge gained by a maintenance programmer can be used help in future maintenance work.

Three techniques for factoring a module are presented:

- *Grouping by Type-Families*

  Entities are grouped together because they depend on the same *type-family* (where a type-family is a collection of inter-related types).

- *Grouping by Imports*

  Entities are grouped together because they have the same set of client modules.

- *Grouping by State Variables*

  Entities are grouped together because they use the same state variables.

Each of these techniques makes use of the five specialised forms of the entity-to-entity graph described in the following section.

9.2 The Five Graphs

In order to perform module factoring, the entity-to-entity graph for a module needs to be partitioned into the following subgraphs:

1. A type-connection graph.
2. A call graph.
3. A reference graph.
4. A variable/type association graph.
5. A variable usage graph.

These five graphs fully partition an entity-to-entity graph as they contain all the information that is found in the entity-to-entity graph, i.e., if the graph union operation is applied to the above graphs the original entity-to-entity graph would be obtained.

Each of these graphs will be described in the following subsections, and the appropriate $\alpha\beta$-slicing operation needed to extract such a graph from an entity-to-entity graph is given.

9.2.1 The Type-Connection Graph

The type-connection graph, $G_{tc}(\mathcal{N}_{tc}, \mathcal{E}_{tc})$, is a graph where all the nodes in $\mathcal{N}_{tc}$ are associated with entities that are types. This graph only records three forms of dependencies: `of-type` when a type entity is used to declare another type, and `$ISOLATED$` otherwise. With Oberon it is possible to have the `extension-of` dependency between two type entities. Such a dependency denotes that one type has been constructed as an extension of another.

```
T1 = POINTER TO T2;
T2 = ARRAY [0 .. 9] OF CHAR;
T3 = CHAR;
```

(a)

```
T0 = CHAR;
T1 = POINTER TO T2;
T2 = ARRAY [0 .. 9] OF T0;
T3 = T0;
```

(b)

Figure 9.1: Two Examples of Type-Families

Consider for example the Modula-2 type declarations in Figure 9.1. In Figure 9.1(a) `T1` is dependent on `T2`, and `T3` is independent of both `T1` and `T2`, whereas in Figure 9.1(b) `T1` is dependent on `T2` and both `T2` and `T3` are dependent on `T0`. Figure 9.2 gives the type-connection graphs associated with the two sets of type declarations in Figure 9.1.

The type-connection graph helps to determine the **type-families** contained within a module. A type-family is a collection of types that are related to each

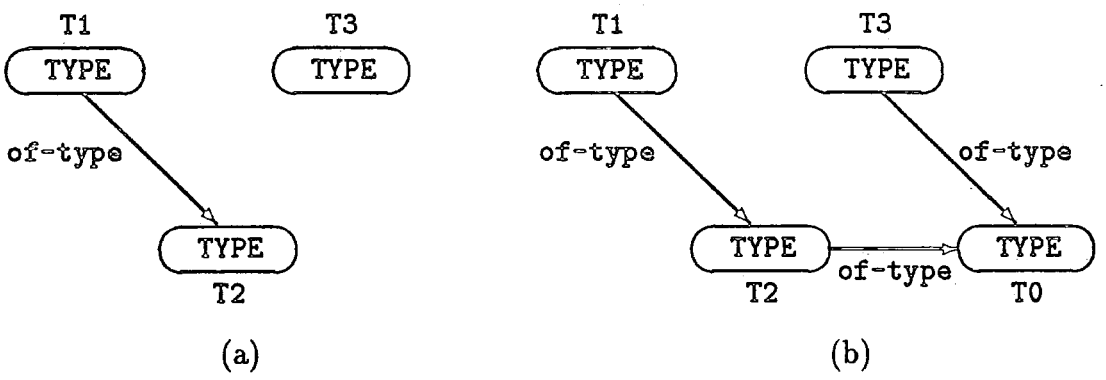


Figure 9.2: Graphical Interpretation of the Dependencies Given in Figure 9.1

other. For example, the Modula-2 type declarations in Figure 9.1(a) introduce two type-families. The types T1 and T2 constitute one type-family and the type T3 constitutes another type-family. T2 and T3 are both dependent on the type CHAR, but as this is a predefined type, T2 and T3 are classed as being independent. For the purpose of creating type-families for module factoring, the only dependencies that are important are those between the types declared in the module that is being factorised.

In comparison, the logically equivalent Modula-2 type declarations in Figure 9.1(b) introduce only one type-family. The type T0 is declared to be synonymous with the predefined type CHAR, and the new declarations of T2 and T3 show these two types to be dependent on T0. Therefore all the types T0, T1, T2 and T3 are now dependent thereby forming only one type-family.

The type-connection graph can be obtained by an $\alpha\beta$-slicing operation on the entity-to-entity graph of a module. Consider for example the type declarations in Figure 9.1(a). If these type declarations occur within the the same module then the

type-connection graph $G^a_{tc}(\mathcal{N}^a_{tc}, \mathcal{E}^a_{tc})$ is the subgraph obtained by the operation:

$$_{class=TYPE}\|_{class=TYPE}(G(\mathcal{N}, \mathcal{E}), \langle \xi, \xi \rangle)$$

where $G(\mathcal{N}, \mathcal{E})$ is the entity-to-entity graph associated with the module.

In order to determine the type-families of a module, the function *get-proper-subgraphs* that is specified on page 99 is applied to the type-connection graph for that module. Each proper subgraph that is returned denotes a type-family. For example, the result of the function call,

$$\textit{get-proper-subgraphs}(G^a_{tc}(\mathcal{N}^a_{tc}, \mathcal{E}^a_{tc}))$$

is the set of graphs,

$$\{G^1_{tf}(\mathcal{N}^1_{tf}, \mathcal{E}^1_{tf}), G^2_{tf}(\mathcal{N}^2_{tf}, \mathcal{E}^2_{tf})\}$$

where,

$$\mathcal{N}^1_{tf} = \{\texttt{T1, T2}\} \quad \mathcal{E}^1_{tf} = \{(\texttt{T1, T2})\}$$
$$\mathcal{N}^2_{tf} = \{\texttt{T3}\} \qquad \mathcal{E}^2_{tf} = \{\}$$

The result of the function call,

$$\textit{get-proper-subgraphs}(G^b_{tc}(\mathcal{N}^b_{tc}, \mathcal{E}^b_{tc}))$$

where $G^b_{tc}(\mathcal{N}^b_{tc}, \mathcal{E}^b_{tc})$ is the graph in Figure 9.2(b), is the graph $G^b_{tc}(\mathcal{N}^b_{tc}, \mathcal{E}^b_{tc})$ itself as this graph has no subgraphs and therefore has only one type-family, which is denoted by the type-connection graph itself.

9.2.2 The Call Graph

The call graph, $G_{cg}(\mathcal{N}_{cg}, \mathcal{E}_{cg})$, is a graph that describes the calling or invoking dependencies between routines. When constructing the call graph for a module the normal dependencies that are recorded are the `invokes` and the `$ISOLATED$` dependencies. With the call graph for an entire system, detecting the existence of `$ISOLATED$` dependencies in a call graph is the same as detecting routines that are never called; however, with the call graph for a module the `$ISOLATED$` dependency is common. In this context, the `$ISOLATED$` dependency does not determine that a routine is never invoked, but instead that the routine is not called by any of the other routines within that module.

Just as with the type-connection graph, the call graph $G_{cg}(\mathcal{N}_{cg}, \mathcal{E}_{cg})$ can be obtained by applying an $\alpha\beta$-slicing operation:

$$_{class=ROUTINE \vee CONSTANT}\|_{class=ROUTINE \vee CONSTANT}(G(\mathcal{N}, \mathcal{E}), \langle \xi, \xi \rangle)$$

where $G(\mathcal{N}, \mathcal{E})$ is the entity-to-entity interconnection graph for the module. For the purpose of this work, it is useful to regard a constant as a value function that always returns the same value. As a result, entities that are constants appear in the call graph.

In the previous subsection, it was shown how the type-connection graph could be used to determine the type-families of a module. In a similar way, the call graph can be used to obtain routine groups. A **routine group** is a group of routines that are dependent because they invoke each other. The routine groups of a module can

be determined by applying the *get-proper-subgraphs* function to a call graph, i.e.,

$$get\text{-}proper\text{-}subgraphs(\mathrm{G}_{cg}(\mathcal{N}_{cg}, \mathcal{E}_{cg}))$$

Each graph contained in the resulting set of graphs denotes a routine group.

9.2.3 The Reference Graph

The reference graph, $\mathrm{G}_{rg}(\mathcal{N}_{rg}, \mathcal{E}_{rg})$, is a graph where all the nodes in $\mathcal{N}_{rg}$ are associated with either type, constant or routine entities. Embley and Woodfield [55] use reference graphs to assess the quality of abstract data types according to the cohesion and coupling measures that they advocate [54]. In module factoring this graph is used to show the dependencies between the type-families and the routine groups of a module. As with the type-connection graph and the call graph described above, the reference graph is analysed in order to detect subgraphs. It can be shown that each subgraph denotes an abstract data type.

A reference graph $\mathrm{G}_{rg}(\mathcal{N}_{rg}, \mathcal{E}_{rg})$ is extracted from an entity-to-entity interconnection graph by applying the following $\alpha\beta$-slicing operations:

$$(_{class=ROUTINE \vee CONSTANT} \| _{class=TYPE}(\mathrm{G}(\mathcal{N}, \mathcal{E}), \langle \xi, \xi \rangle)) \sqcup$$
$$(_{class=TYPE} \| _{class=TYPE}(\mathrm{G}(\mathcal{N}, \mathcal{E}), \langle \xi, \xi \rangle)) \sqcup$$
$$(_{class=TYPE} \| _{class=CONSTANT}(\mathrm{G}(\mathcal{N}, \mathcal{E}), \langle \xi, \xi \rangle))$$

where $\mathrm{G}(\mathcal{N}, \mathcal{E})$ in the entity-to-entity interconnection graph for the module. This is the same operation that is given on page 100 for extracting from a module an abstract data type that satisfies Booch's classification.

Analysis of $G_{rg}(\mathcal{N}_{rg}, \mathcal{E}_{rg})$ has to be done with respect to the entity groups that are produced by analysing $G_{tc}(\mathcal{N}_{tc}, \mathcal{E}_{tc})$ and $G_{cg}(\mathcal{N}_{cg}, \mathcal{E}_{cg})$. If there exists one or more routines that are not connected to any of the type-families, then these routines are grouped together. They will be placed in another module, i.e., a module which is separate from those produced for each of the type families.

Ideally, each routine group should be connected to only one of the type-families. If there exists a connection between two type-families (because one or more routine groups depends on both of them), this does not prohibit factoring with respect to the type-families. However, it does indicate that one of the type-families must be exported to the module that contains the other type-family. The choice of which type-family to export cannot be resolved automatically but instead must depend on the expertise of the programmer who is factoring the module.

Having analysed the type-connection graph, the call graph and the reference graph, the modules which can replace the original module have been determined. The next two graphs are analysed with respect to both the type-families and the routine groups in order to determine how the variables are to be distributed.

9.2.4 The Variable/Type Association Graph

A variable/type association graph, $G_{vt}(\mathcal{N}_{vt}, \mathcal{E}_{vt})$ shows the dependencies between global variables and the type-families. $G_{vt}(\mathcal{N}_{vt}, \mathcal{E}_{vt})$ is normally a disjoint graph with each variable being dependent on only one of the type-families. The dependencies recorded in a variable/type association graph are `of-type`, when a variable is declared to be of a type declared in the same module, and `$ISOLATED$` otherwise.

A variable/type association graph $G_{vt}(\mathcal{N}_{vt}, \mathcal{E}_{vt})$ is extracted from an entity-to-entity graph by applying the following $\alpha\beta$-slicing operation:

$$_{class=VARIABLE}\|_{class=TYPE}(G(\mathcal{N}, \mathcal{E}), \langle \xi, \xi \rangle)$$

where $G(\mathcal{N}, \mathcal{E})$ is the entity-to-entity interconnection graph for the module.

Such a graph has certain characteristics. It will normally be found that the variables can be split into disjoint groups. The variables that are dependent on a particular type-family can be grouped together. They will eventually be placed in the new module created for the type-family. However, a variable will be independent of the type-families if it is declared to be of a type that is not declared in the same module as the variable. Those variables which are independent of all the type-families can temporarily be grouped together.

```
VAR ExampleVar:  RECORD
                    Field1:  TypeFamily1;
                    Field2:  TypeFamily2
                 END;
```

Figure 9.3: Variable Declaration that is Dependent on Two Type-Families

These characteristics of the variable/type association graph are typical for programs written in languages like Ada and Modula-2, where a programmer is encouraged to build new types in terms of previously declared types. It is possible however for a variable to be dependent on more than one type-family. Consider for example, the variable declaration given in Figure 9.3. The variable `ExampleVar` is dependent on the types `TypeFamily1` and `TypeFamily2`, which are representing different type-families. As with the reference graph, this form of connection does not prohibit module factoring, but indicates that some of the new modules will

have to be connected.

9.2.5 The Variable Usage Graph

The variable usage graph, $G_{vu}(\mathcal{N}_{vu}, \mathcal{E}_{vu})$ shows the dependence of the routine groups on variables. Variables that are classed as being dependent on one of the routine groups are assigned to that entity group. For variables that are classed as being independent of all the routine groups, the variable groups derived from $G_{vt}(\mathcal{N}_{vt}, \mathcal{E}_{vt})$ are considered. If a variable is classed as being dependent on one of the type-families, it is assigned to the entity group associated with that type-family. If, on the other hand, the variable is classed as being independent of the type-families then it is assigned to the entity group that is independent of all the type-families. Just as a variable can be dependent on more than one type-family, it can also be dependent on more than one routine group. The resolution of which entity group the variable should be assigned to cannot be resolved automatically; instead the programmer must make this decision.

A variable usage graph $G_{vu}(\mathcal{N}_{vu}, \mathcal{E}_{vu})$ is obtained by applying the $\alpha\beta$-slicing operation:

$$_{class=ROUTINE}\|_{class=VARIABLE}(G(\mathcal{N}, \mathcal{E}), \langle \xi, \xi \rangle)$$

on $G(\mathcal{N}, \mathcal{E})$, the entity-to-entity graph for a module.

9.3 Three Module Factoring Techniques

Three techniques for module factoring will be described in the following subsections. Each of the techniques make use of the five graphs described above. The m2dep program that is provided by Sun Microsystems Inc. with their Modula-2 system will be analysed and factorised.

Figure 6.16 on page 133 gives the module-to-module interconnection graph for the m2dep system. In the analysis of this graph given in section 6.4, the modules `IO` and `UnixSupport` are identified as being fundamental modules. This means that these modules are either providing a single service that is in some way critical to the system, or they are potpourri modules providing several diverse facilities. Module factoring provides a means by which it is possible to determine whether a fundamental module is also a potpourri module. In the following sections, the module `UnixSupport` will be analysed and factorised by means of the "grouping by type-families" and "grouping by imports" techniques, while the module `IO` will be analysed and factorised by means of the "grouping by state variables" technique.

9.3.1 Grouping by Type-Families

The Technique

The "grouping by type-families" technique factors a module by grouping variable, constant and routine entities with the type-family on which they depend and by forming a separate module for each of these groups. For example, if the routines `R1`, `R2` and `R3` just use the types of one type-family, and the routines `R4` and `R5` use

the types of another type-family, then the technique would cause R1, R2 and R3 to be placed in one module, and R4 and R5 to be placed in another. The technique can specifically be used to factor out any abstract data types defined by a module.

In order to perform the "grouping by type-families" technique, it is necessary to know which global entities are dependent on each other. This information can be obtained in a form that facilitates analysis by producing the graphs described above.

The type-connection graph can be used to determine the type-families, while the call graph can be used to determine the routine groups. The reference graph provides a means of determining the dependence between the routine groups and the type-families. Ideally each routine group should be dependent on at most one type-family thereby revealing concealed abstract data types. If there is a routine group that is dependent on more than one type-family then module factoring using the "grouping by type-families" technique is still possible, but greater programmer involvement in the module factoring process is needed.

Consider for example the following segment of code

```
TYPE T1 = [0..9];
     T2 = CHAR;
PROCEDURE P(para1:  T1; para2:  T2);
     :
```

The types T1 and T2 are independent and constitute different type-families. However, the routine P is dependent on both T1 and T2, and so the routine group containing the routine P is dependent on two type-families. In order to overcome

this problem, the programmer can either choose to house T1 and T2 in the same module or else house T1 and T2 in different modules. Housing T1 and T2 in the same module means that the programmer has decided to have one module to represent two type-families; in this way a module represents abstract data types with shared operations as has been proposed by Osterbye [111].

Housing T1 and T2 in different modules means that the programmer does not favour the interpretation of a module representing abstract data types with shared operations. Instead, each module represents a single abstract data type. This then forces the programmer to decide which of the two new modules will house the routine P and therefore have to import the other type-family. This solution always results in some of the modules being dependent on each other because one module has to import some members of type-families housed in other modules.

The routines of any routine group that are independent of the type-families (because the routines do not use any of the type-families) may be placed in a separate module that contains none of the type-families. It is possible for a routine group to be indirectly dependent on a type-family because the routine group is dependent on a state variable which in turn is dependent on a type-family. In order to determine if there are any indirect dependencies between routine groups and type-families, the variable/type association graph and the variable usage graph have to be analysed. Any routine groups that are neither directly or indirectly dependent on any of the type-families are housed in a separate module.

At this stage any concealed abstract data types have been detected. It is now necessary to check whether the new modules need to record any **state information**. State information is any data that is stored in a state variable. In order to do this, the variable/type association graph and the variable usage graph are

analysed. If the module that is being factorised has no state variables then these two graphs will be empty.

Ideally each state variable should be dependent on one type-family and one routine group, but as with the dependency between routine groups and type-families, violation of this ideal does not prohibit module factoring by the "grouping by type-families" technique. The variable/type association graph when analysed in conjunction with the type-connection graph, reveals on which type-families each state variable is dependent. If a state variable is dependent on more than one type-family, then the programmer must decide in which module to house the state variable and establish the appropriate importing links between these new modules.

The variable usage graph when analysed in conjunction with the call graph establishes on which routines each state variable depends. If a state variable is dependent on only one routine grouping then the state variable is housed with the routine group. If a state variable is dependent on more than one routine group then the programmer must decide in which module to house the state variable. This state variable must also be exported to the new module housing the other routine group. As a guideline on which entity groups should become modules it is preferable not to export variables. Therefore entity groups should be arranged to minimise the number of state variables that are exported. This may mean an occasional merging of different entity groups.

An Example

We now look at an example of module factoring using the "grouping by type-families" technique. Consider the definition module for `UnixSupport` given in Fig-

```
DEFINITION FOR C MODULE UnixSupport;

  FROM SYSTEM IMPORT BYTE, ADDRESS;

  TYPE
    Channels = INTEGER;
    StringPointer = POINTER TO ARRAY [0 .. 0] OF CHAR;

  CONST
    maxFileNameLength = 1024;
    stdin = 0;  stdout = 1;  stderr = 2;
    EOL = 12C;
    ReadOnly  = 0;
    WriteOnly = 3001B;

    PROCEDURE read(FileDesc  : Channels;
                   VAR Buffer    : ARRAY OF BYTE;
                       ByteCount : CARDINAL): CARDINAL;

    PROCEDURE write(FileDesc  : Channels;
                    VAR Buffer    : ARRAY OF BYTE;
                        ByteCount : CARDINAL);

    PROCEDURE open(FileName : ARRAY OF CHAR;
                   Mode     : CARDINAL): Channels;

    PROCEDURE close(FileDesc : Channels);

    PROCEDURE strlen(S : ARRAY OF CHAR): CARDINAL;

  TYPE
    Comparator = PROCEDURE(ADDRESS, ADDRESS): INTEGER;

    PROCEDURE strcmp(S1, S2 : ARRAY OF CHAR): INTEGER;

    PROCEDURE qsort(VAR data         : ARRAY OF BYTE;
                        elementCount : CARDINAL;
                        elementSize  : CARDINAL;
                        compProc     : Comparator);

END UnixSupport.
```

Figure 9.4: The Definition Module for UnixSupport

ure 9.4. This is a non-standard Modula-2 definition module as there is no associated implementation module giving the elaboration of the routines. The module `UnixSupport` is providing a Modula-2 interface to predefined C routines. This non-standard module does not adversely affect the module factoring technique: the only effect of this non-standard module is that, as there is no implementation module, all the entity dependencies are derived from the definition module alone.

Figure 9.5 gives the entity-to-entity graph for `UnixSupport`. This graph also serves as the reference graph for `UnixSupport` as all the entities are either constants, types or routines. The type-connection graph for `UnixSupport` is given in Figure 9.6. This shows that there are three type-families each consisting of a single type entity. Similarly, the call graph (Figure 9.7) for `UnixSupport` consists of 14 routine groups each consisting of a single routine (or constant). As `UnixSupport` contains no state variables, the variable/type association graph and the variable usage graph are both empty. Therefore `UnixSupport` is factorised by analysing the type-connection graph, the call graph and the reference graph only.

As the type-connection graph and the call graph show each of the entities to be independent, the entity groups used to factor `UnixSupport` will be those obtained from the reference graph. Figure 9.5 shows that the type `Comparator` and the routine `qsort` are dependent, and the type `Channels` and the routines `open`, `close`, `read` and `write` are dependent. Therefore, the reference graph has revealed two abstract data types which can be removed from `UnixSupport` and housed in their own modules. The remaining entities will be placed in a new module called `UnixSupport2`. The three new definition modules are given in Figure 9.8.

The module-to-module interconnection graph for the new system is given in Figure 9.9. It shows the new modules, `ComparatorADT` and `ChannelsADT` to be

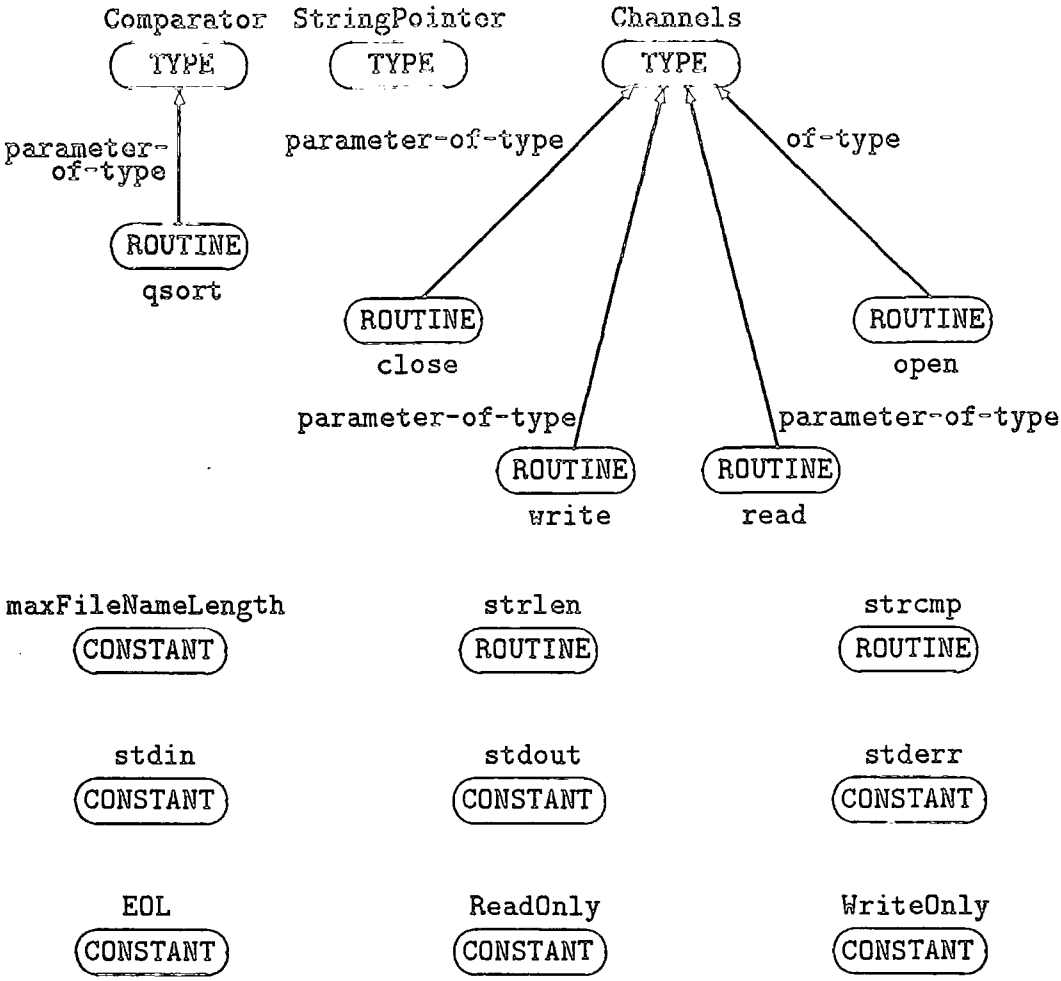


Figure 9.5: The Entity-To-Entity Interconnection Graph for `UnixSupport`

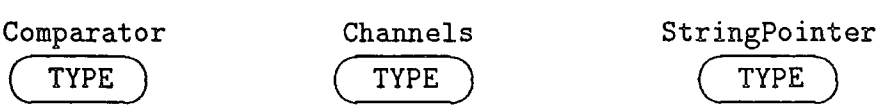


Figure 9.6: The Type-Connection Graph for `UnixSupport`

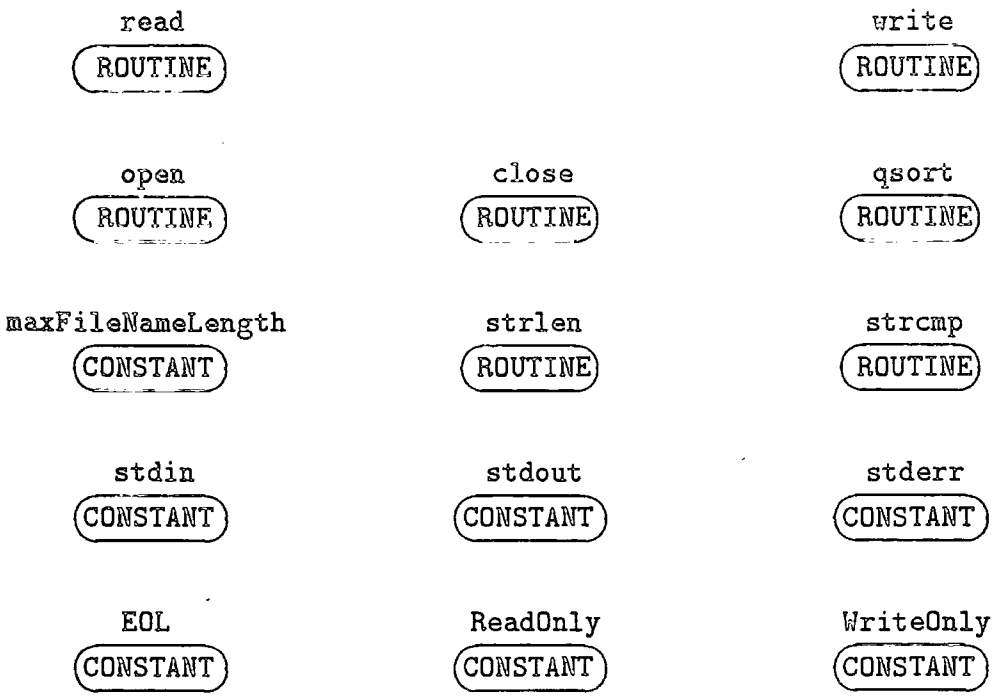


Figure 9.7: The Call Graph for `UnixSupport`

specialised modules. This is consistent with the fact that each of these modules provides an abstract data type to the system.

9.3.2 Grouping by Imports

The Technique

Entities that are connected because they perform similar functions in a system are typically imported into the same set of modules. The "grouping by imports" technique suggests that entities that are imported into the same set of modules should be declared by the same module. As a consequence, entities that are imported into a different set of modules should be declared by different modules.

```
DEFINITION FOR C MODULE ComparatorADT;
  FROM SYSTEM IMPORT BYTE, ADDRESS;
  TYPE
    Comparator = PROCEDURE(ADDRESS, ADDRESS): INTEGER;
    PROCEDURE qsort(VAR data         : ARRAY OF BYTE;
                        elementCount : CARDINAL;
                        elementSize  : CARDINAL;
                        compProc     : Comparator);
END ComparatorADT.


DEFINITION FOR C MODULE ChannelsADT;
  FROM SYSTEM IMPORT BYTE;
  TYPE
    Channels = INTEGER;
    PROCEDURE read(FileDesc  : Channels;
                VAR Buffer   : ARRAY OF BYTE;
                    ByteCount : CARDINAL): CARDINAL;
    PROCEDURE write(FileDesc  : Channels;
                 VAR Buffer   : ARRAY OF BYTE;
                     ByteCount : CARDINAL);
    PROCEDURE open(FileName : ARRAY OF CHAR;
                   Mode     : CARDINAL): Channels;
    PROCEDURE close(FileDesc : Channels);
END ChannelsADT.


DEFINITION FOR C MODULE UnixSupport2;
TYPE
  StringPointer = POINTER TO ARRAY [0 .. 0] OF CHAR;
CONST
  maxFileNameLength = 1024;
  stdin = 0;  stdout = 1;  stderr = 2;
  EOL = 12C;
  ReadOnly = 0;  WriteOnly = 3001B;
  PROCEDURE strlen(S : ARRAY OF CHAR): CARDINAL;
  PROCEDURE strcmp(S1, S2 : ARRAY OF CHAR): INTEGER;
END UnixSupport2.
```

Figure 9.8: The Three New Definition Modules

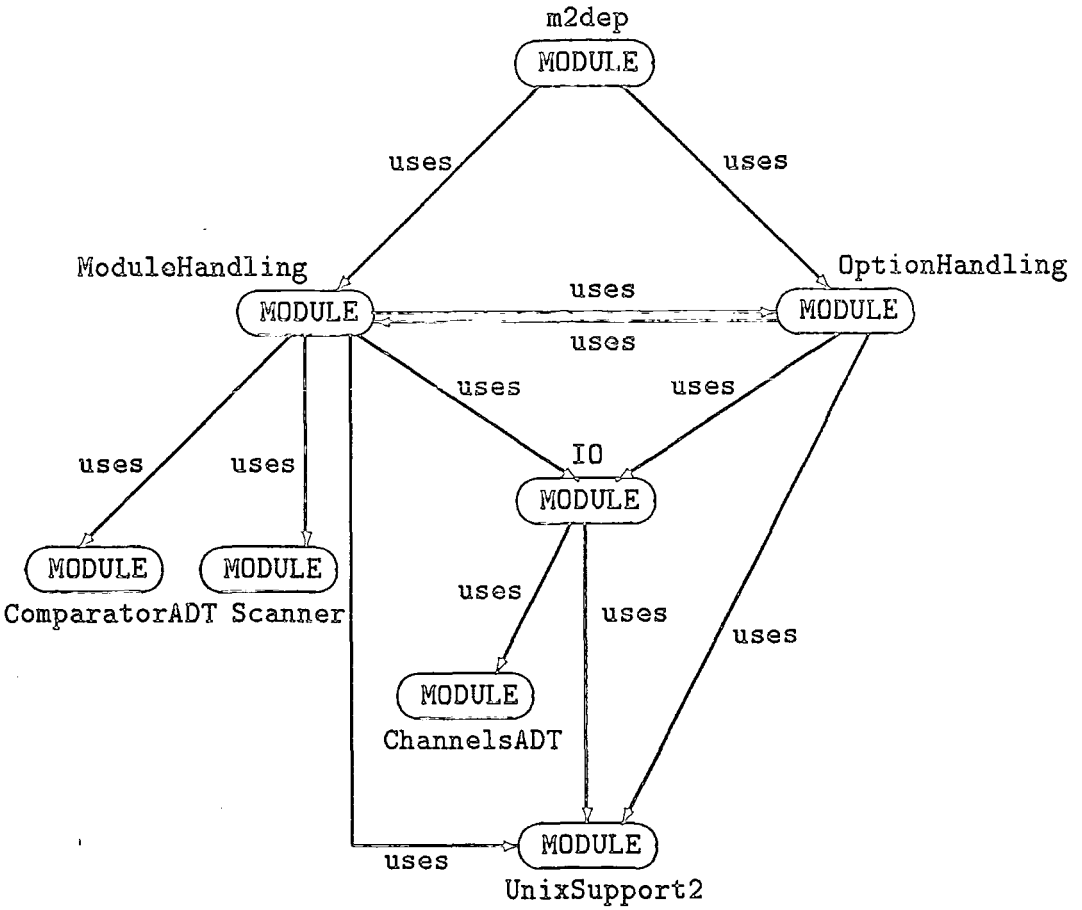


Figure 9.9: The Module-to-Module Graph for the Second Version of `m2dep`

The "grouping by imports" technique has two main phases. In phase 1, entity groups are established; and in phase 2, any inter-group connections are resolved.

Phase 1 of this technique involves determining which modules make use of another module's exported entities. Some languages, like Eiffel, provide selective export capabilities. With these languages it is possible to determine the different entity groups by only analysing the export clause of the module to be factorised. However, in other languages this is not possible and, instead, all the modules that constitute the program have to be examined to determine if they use any of the exported entities of the module to be factorised.

When the different entity groups have been derived, these groups are then used as the basis for creating new modules to replace the given module. Each entity group could be used to form a new module, but there are situations when this would not be desirable. Any two entity groups can either be *independent* of each other (when neither of the groups shares entities) or be *subgrouped* (when one of the groups is regarded as being a subgroup of the other).

Entity groups that are classed as being subgrouped can be merged without violating the principle behind this factoring technique. Since the supergroup is connected to all the modules that the subgroup is connected to, the merging of the two groups can be achieved without adding extra module connections. When the entity groups are classed as being independent, then the two groups should be represented by separate modules.

When the different entity groups that are to form the new modules have been established, any inter-group connections that exist should now be detected. This constitutes phase 2 of the "grouping by imports" module factoring technique. Fur-

thermore, a distribution of the module's private entities must also take place.

Any private entities that are dependent on only one of the entity groups are placed in the same module as that entity group, and those entities remain private. If on the other hand a private entity is dependent on two or more of the entity groups then it is placed in a separate module and imported by the modules that contain the entity groups. This entails making some of the private entities public.

Thus, in order to perform the "grouping by imports" technique it is necessary to determine the dependencies between the private entities. As with the "grouping by type-families" technique, this analysis can easily be performed once the graphs described in section 9.2 have been produced.

An Example

Consider the module `UnixSupport2` given in Figure 9.8. By examining the entity-to-module interconnection graph for the m2dep program, Tables 9.1 and 9.2 are obtained. The tables reveal which entities from `UnixSupport2` are imported by which modules.

Group 1	Group 2	Group 3	Group 4	Group 5

`maxFileNameLength`
`stderr`
`strcmp`
	`StringPointer`
	`strlen`
		`EOL`
		`ReadOnly`
			`stdin`
			`WriteOnly`
				`stdout`

Table 9.1: The Original Entity Groups for `UnixSupport2`

Using programmer expertise, some of the entity groups given in Table 9.1 can be merged. From Table 9.2 the entity in Group 5 is seen to be imported by the module `ModuleHandling` only, while the entities in Group 1 are imported by both

Group	Module Sets
Group 1	ModuleHandling, OptionHandling
Group 2	ModuleHandling, OptionHandling, IO
Group 3	IO
Group 4	$NONE$
Group 5	ModuleHandling

Table 9.2: The Sets of Importing Modules for UnixSupport2

of the modules ModuleHandling and OptionHandling, and the entities in Group 2 are imported by ModuleHandling, OptionHandling and IO. As the set of importing module for Group 5 is a subset of those for Group 1 and Group 2, the entity in Group 5 can be merged with either Group 1 or Group 2 without increasing the number of dependencies between modules. Using knowledge of the Unix operating system we can see that stdout is similar in function to stderr in Group 1, and so Group 5 is merged with Group 1.

Table 9.2 shows that the entities in Group 4 are not imported by any of the modules in the system. This allows the programmer the freedom to remove the entities without affecting the execution of the program. Alternatively the programmer could place these entities into any of the other entity groups. Programmers often implement a module with a view to future needs, so instead of removing the redundant entities they can be reallocated to different entity groups. Group 1 now contains the entity stderr and stdout (as a result of merging Group 5 with Group 1); as these entities are similar in function to stdin in Group 4, stdin is moved to Group 1. Similarly the entity WriteOnly in Group 4 is moved to Group 3 as it is similar in function to ReadOnly in Group 3.

The entities in Group 2 are related to the string data type, this means that the entity strcmp in Group 1 should really be in Group 2 if each of the entity groups

were supposed to represent a different responsibility assignment. Table 9.2 shows that the set of importing modules for Group 1 is a subset of that for Group 2, and so entities from Group 1 can be moved to Group 2 without increasing the number of dependencies. Therefore `strcmp` is moved from Group 1 to Group 2.

Group 1	Group 2	Group 3
maxFileNameLength	StringPointer	EOL
stderr	strlen	ReadOnly
stdin	strcmp	WriteOnly
stdout		

Table 9.3: The Final Entity Groups for `UnixSupport2`

Table 9.3 gives the final entity groups for `UnixSupport2`.

9.3.3 Grouping by State Variables

The Technique

The "grouping by state variables" technique factors a module by grouping constant and routine entities with the state variables that they use. This module factoring technique in effect detects any abstract-state machines within a module and extracts them. This module factoring technique is restricted to grouping variable, constant and routine entities because the "grouping by type-families" technique can be used to group type, variable, constant and routine entities.

The "grouping by state variables" technique uses only the call graph and the variable usage graph. The other graphs contain type entities and therefore are of no relevance to this factoring technique. The call graph is used to obtain the routine groups and the variable usage graph is used to show which routine groups

are dependent on which state variables. Any routine groups that are dependent on the same state variable are merged into a single group. Any routine groups that are dependent on none of the state variables are merged into a single group. In this way the programmer is eventually presented with a set of entity groups such that one group may be a set of routines that are independent of the state variables, and one or more groups consist of variable, constant and routine entities. Each of these entity groups can be made into a separate module.

An Example

Figure 9.10 gives the definition module for the module `IO` from the m2dep system, and Figure 9.11 gives an outline of its implementation module. The bodies of the routines have been omitted, but the entities they use are listed. This module does not have any type entities, and so the "grouping by type-families" technique is not possible.

Figure 9.12 gives the entity-to-entity graph for the module `IO`. The absence of any type entities means that this is also the variable usage graph for `IO`. The call graph for `IO` in Figure 9.13 shows that the routines `WriteChar`, `WriteLn` and `WriteCard` are dependent and so constitute a single routine group. The other routines are independent and so each routine constitutes a routine group. The variable usage graph shows that the routines `ReadChar`, `CloseSource` and `OpenSource` all use the state variables `sourceFile` and `sourceOpen`, and so they are all grouped together, along with the state variables `srcBuff` and `srcBuffIndex`, which are used only by `ReadChar`, and with the state variable `sourceChars` which is used by the routines `ReadChar` and `CloseSource`.

```
DEFINITION MODULE IO;

  FROM UnixSupport IMPORT
    Channels, StringPointer;

  PROCEDURE WriteChar(ch: Channels; C: CHAR);

  PROCEDURE WriteString(ch: Channels; Str: ARRAY OF CHAR);

  PROCEDURE WriteStringIndirect(ch: Channels; sp: StringPointer);

  PROCEDURE WriteLn(ch: Channels);

  PROCEDURE WriteCard(ch: Channels; C: CARDINAL; Width: CARDINAL);

  PROCEDURE OpenSource(namePtr: StringPointer): BOOLEAN;

  PROCEDURE ReadChar(VAR CH: CHAR);

  PROCEDURE CloseSource;

END IO.
```

Figure 9.10: The Definition Module for IO

Group 1	Group 2
OpenSource	WriteChar
CloseSource	WriteString
ReadChar	WriteStringIndirect
sourceFile	WriteLn
sourceOpen	WriteCard
srcBuff	
srcBuffIndex	
sourceChars	

Table 9.4: The Final Entity Groups for IO

```
IMPLEMENTATION MODULE IO;
  FROM UnixSupport IMPORT EOL, StringPointer, Channels, open,
                          close, read, ReadOnly, write, strlen;
  VAR
    sourceFile    : Channels; sourceOpen    : BOOLEAN;
    srcBuff       : ARRAY [0..2047] OF CHAR;
    srcBuffIndex  : CARDINAL; sourceChars   : CARDINAL;
  PROCEDURE WriteChar(ch: Channels; C: CHAR);
  ...
  PROCEDURE WriteString(ch: Channels; Str: ARRAY OF CHAR);
  ...
  PROCEDURE WriteStringIndirect(ch: Channels; sp: StringPointer);
  ...
  PROCEDURE WriteLn(ch: Channels);
  (* WriteChar is invoked   *)
  ...
  PROCEDURE WriteCard(ch: Channels; C: CARDINAL; Width: CARDINAL);
  (* WriteChar is invoked   *)
  ...
  PROCEDURE OpenSource(namePtr: StringPointer): BOOLEAN;
  (* sourceFile is defined and referenced   *)
  (* sourceOpen is defined                  *)
  ...
  PROCEDURE ReadChar(VAR CH: CHAR );
  (* sourceFile is referenced               *)
  (* sourceOpen is referenced               *)
  (* sourceChars is defined and referenced  *)
  (* srcBuff is defined and referenced      *)
  (* srcBuffIndex is defined and referenced *)
  ...
  PROCEDURE CloseSource;
  (* sourceOpen is defined and referenced   *)
  (* sourceChars is defined                 *)
  (* sourceFile is referenced               *)
  ...
BEGIN sourceOpen:= FALSE END IO.
```

Figure 9.11: The Implementation Module for IO

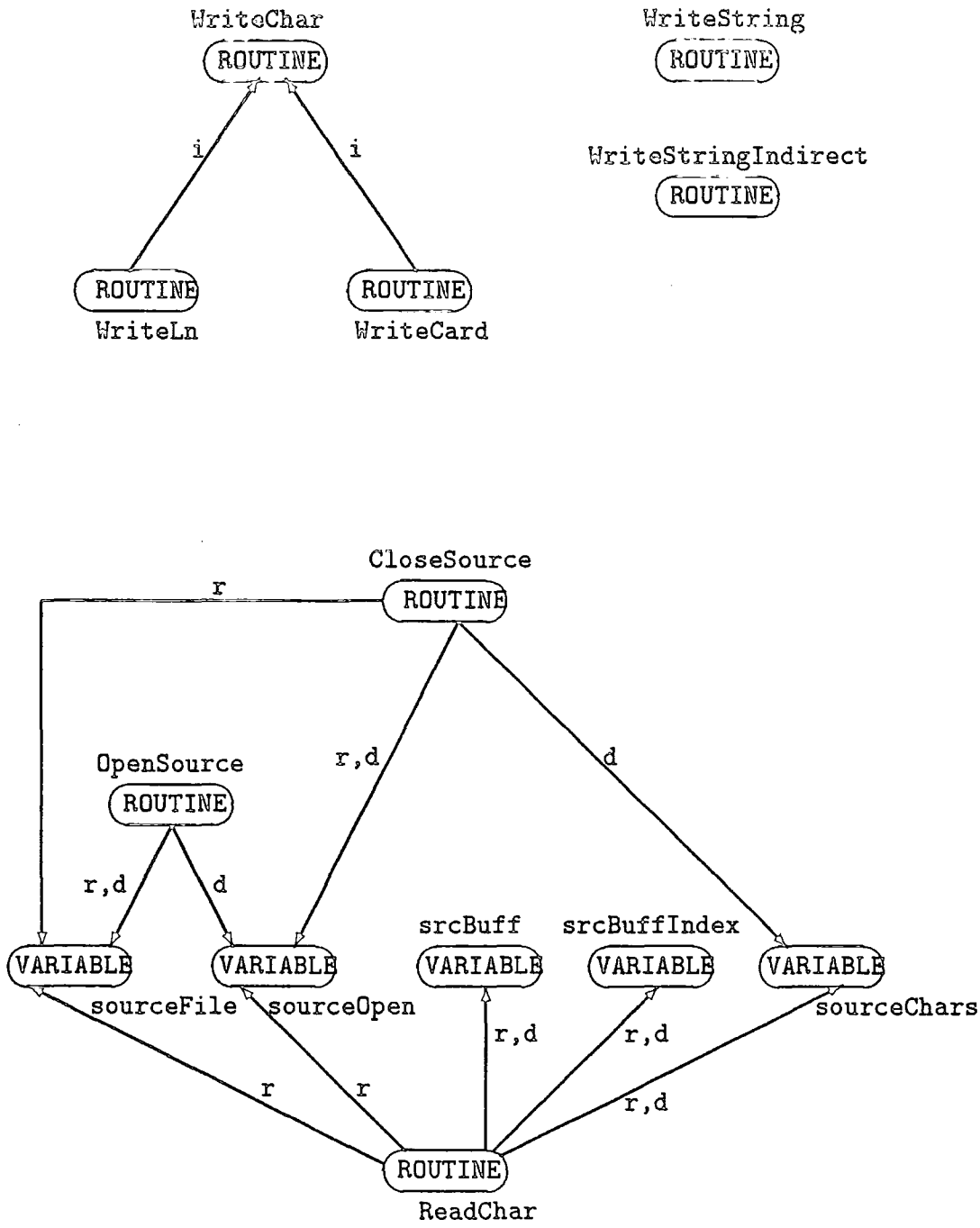


Figure 9.12: The Entity-to-Entity Graph for IO

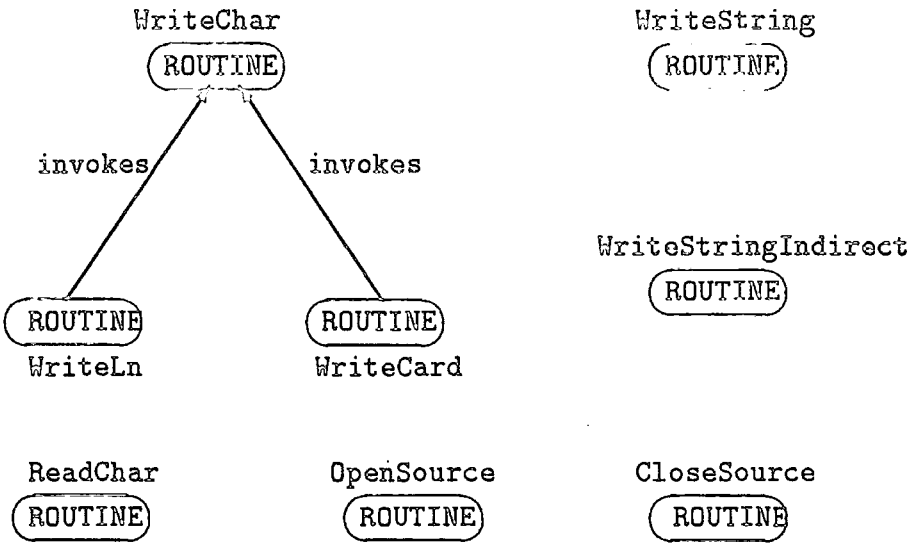


Figure 9.13: The Call Graph for IO

As a result of grouping entities around the five state variables of IO, we obtain the groupings given in Table 9.4. The entities in Group 1 are the state variables and the routines that use them; the entities in Group 2 are the routines that are independent of the state variables.

Thus, the result of the "grouping by state variables" suggests that the entities in these two groups should be put into separate modules.

Here the module factoring process has clearly separated the entities associated with input from those associated with output operations. Such a clear separation between the different entity groups should normally occur within a module. This is because a state variable is a device used to help implement such algorithms. A state variable is used within a module to either: transport data from one routine to another; to transfer status information to other routines which they may use in conditional statements; or the state variable may be a means to record data so that a routine can reuse that data when it is re-executed.

9.4 Summary

In this chapter three module factoring techniques have been described:

1. *Grouping by Type-Families*

   This technique detects abstract data types and extracts them.

2. *Grouping by State Variables*

   This technique groups entities according to the state variables that they use.

3. *Grouping by Imports*

   This technique groups entities because they are used by the same set of importing modules.

Each of these techniques has merit, and generally there is no one technique that should be used before the others in all situations. However, it is recommended that the factoring techniques be used in the order listed above unless the maintenance programmer, from his knowledge of a system, believes that a different ordering would be better suited to the module. This ordering of the module factoring techniques is recommended because it will find the different services of a module in an order that corresponds to the current thinking about the services that modules should provide.

The "grouping by type-families" detects the existence of abstract data types within a module. The use of abstract data types in programming has gained widespread acceptance, e.g., Naphtali and Rich [110] describe experiences with designing a system around abstract data types and Linden [96] describes how the use of abstract data types can make a program easier to maintain. The "grouping by state variables" groups routines around the state variables that they use, thereby

creating abstract-state machines. Creating this form of module can help reduce the problem of a module having to export a variable. The "grouping by imports" can then be used to create the "named collection of declarations" and the "group of related program units" modules.

Chapter 10

Design of a Relational Database

10.1 Introduction

So far, inter-module code analysis has been discussed in terms of operations on the module-to-module, entity-to-module and entity-to-entity interconnection graphs. It is impractical to regenerate the information needed for the graphs for every inter-module code analysis operation. Therefore this information should be stored in a form that allows a maintenance programmer to access the information in an efficient way. Using the existing work on databases, a database schema has been designed that will record the necessary information on the dependencies between the entities of a system.

In section 10.2 several different data storage mediums are discussed and the reasons for choosing a relational database are given. Section 10.3 introduces the

necessary relational database terminology, and finally section 10.4 describes how the relational database schema was derived.

10.2 Reasons for Choosing a Relational Database

The amount of information needed to be recorded for inter-module code analysis is large, especially when dealing with large systems. In order to store this information in a form that facilitates efficient data retrieval it is appropriate to utilise some of the existing work on databases.

Three of the existing database models are:

1. *Hierarchical*

   Where the database has a tree structure.

2. *Network*

   Where the database has a graph structure.

3. *Relational*

   Where the database is unstructured and consists of a collection of tables called **relations**.

For inter-module code analysis it is important that the database be amenable to answering several forms of queries, many of which may not be envisaged at the time the database is designed. The relational model describes a database that is better suited to these needs. The hierarchical and network models are structured databases where the links between the data is designed into the database.

This means that the database is going to be more cumbersome with unanticipated queries than the relational model. With the relational model, the links are established by the data itself, and therefore information from one relation can be used to search another relation. Different queries will involve searching new relations.

Yau and Grabow [173] demonstrate the use of a relational database to represent programs written in Pascal. This paper demonstrated the feasibility of using a relational database to model a program written in a block structured language. Other authors have built on this work to create software tools that assist a programmer in scanning the code of a program, e.g., Glagowski [61] and Linton [97].

Each of these relational databases is used to model a different languages. The work of this thesis involves analysis with respect to a particular program construct — the module. A relational database is therefore needed that will model this construct. the resulting database will not be language dependent, so it can be used to model the inter-module connections in several languages. The relational database designed in this thesis should also be able to model the inter-module connections in a program that is written in several languages.

10.3 Relational Database Terminology

This section introduces the relational database terminology and notation that is needed in order to discuss the relational database schema that is being proposed for inter-module code analysis.

A relational database consists of a collection of *tables*, each of which is assigned

a unique name. With relational databases, these tables are known as relations. Each relation consists of a collection of tuples. A tuple is an ordered collection of values, that denote some relationship between these values. Each of the values in a tuple is known as an attribute value. An attribute value is a particular value from a domain of values which is referred to as the attribute type. For the sake of brevity, the term attribute will be used to refer to both an attribute type and an attribute value where the context can identify the intended meaning. Figure 10.1 gives an example of a relation with the different parts of the relation highlighted.

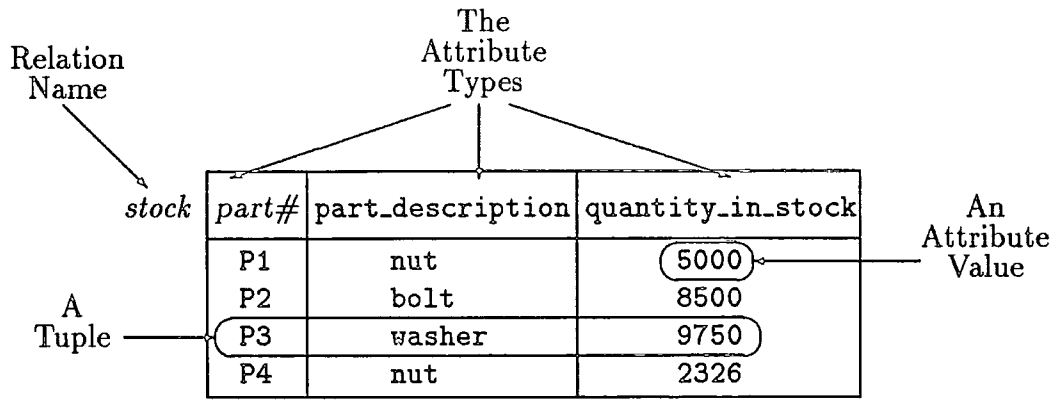


Figure 10.1: An Example of a Relation

A relation scheme (or scheme) names the associated attribute types and corresponds to the programming language notion of a type definition. It has to be instantiated to a relation. (This is analogous to a type being instantiated to a variable.) The relation given in Figure 10.1 has the relation scheme:

`Stock-scheme = (part#, part_description, quantity_in_stock)`

and this is instantiated to a relation by

`stock(Stock-scheme)`

For simplicity, these two will be abbreviated to:

`stock(part#, part_description, quantity_in_stock)`

In order to retrieve selected tuples from a relation, it is important to be able to distinguish between the different tuples. This is done via the key attributes. Candidate keys are the minimum set of attributes needed to identify a tuple uniquely. For a given relation, there can be more than one set of candidate keys. The set of attributes chosen by the database designer as the principle means of identifying tuples within a relation is called the primary key.

10.3.1 The Relational Algebra

When information is stored in a relational database, it is important that a user can easily access this information. In order to facilitate this, several query languages have been developed. The procedural query language *relational algebra* is used in this thesis to interrogate a relational database.

There are five fundamental operations in the relational algebra. These are:

1. select,
2. project,
3. cartesian product,
4. union, and
5. difference.

All of these operations produce a new relation as their result. Each of these operations is described briefly below.

The *select* operation selects all the tuples from a given relation that satisfy a given predicate. The select operation is denoted by the character σ and takes the form,

$$\sigma_{predicate}(relation)$$

Therefore, to select those tuples from `stock` that correspond to a `nut` then the following selection operation can be used,

$$\sigma_{\texttt{part\_description="nut"}}(\texttt{stock})$$

The relation resulting from this operation is given in Figure 10.2.

stock2

part#	part_description	quantity_in_stock
P1	nut	5000
P4	nut	2326

Figure 10.2: The Result of $\sigma_{\texttt{part\_description="nut"}}(\texttt{stock})$

The *project* operation is similar to the select operation in that it operates on only one relation. Whereas the select operation extracts tuples (rows) from a relation, the project operation extracts columns of attribute values from a relation by specifying the attribute types that are to appear in the resulting relation. A project operation is denoted by the character Π, and takes the form,

$$\Pi_{attribute\text{-}names}(relation)$$

As an example, in order to extract the column of *part#* values that are contained

in the relation `stock`, then the following projection operation can be used,

$$\Pi_{\texttt{part\#}}(\texttt{stock})$$

The result of this operation is given in Figure 10.3.

stock3

part#
P1
P2
P3
P4

Figure 10.3: The Result of a *project* Operation

location

part#	shelf
P1	A
P2	B

Figure 10.4: The Relation `location`

The *cartesian product* operation, denoted by $\times$, is used with two relations. A cartesian product operation has the form,

$$relation1 \times relation2$$

The relation scheme for the relation resulting from this operation is the concatenation of the relation scheme for *relation2* to that for *relation1*. As an example, consider the cartesian product operation,

$$\texttt{stock} \times \texttt{location}$$

where the relation `location` is that given in Figure 10.4. The result of this operation is given in Figure 10.5. The original relation name is used as a prefix to the attribute name so as to be able to distinguish between the different attributes.

stock. part#	stock. part_description	stock. quantity_in_stock	location. part#	location. shelf
P1	nut	5000	P1	A
P2	bolt	8500	P1	A
P3	washer	9750	P1	A
P4	nut	2326	P1	A
P1	nut	5000	P2	B
P2	bolt	8500	P2	B
P3	washer	9750	P2	B
P4	nut	2326	P2	B

Figure 10.5: The Relation Resulting from `stock` × `location`

The *union* operation, denoted by ∪, combines two relations that have the same relation scheme. The relation union operation takes the same form as the set union operator, i.e.,

$$relation1 \cup relation2$$

The resulting relation contains the tuples from *relation1* together with any tuples from *relation2* that are not in *relation1*.

The *difference* operation, denoted by −, provides a means of finding tuples that are in one relation but not the other. The result of,

$$relation1 - relation2$$

is the relation containing those tuples in *relation1* but not in *relation2*.

It is possible to combine operations. For example, the answer to the query, "find the part# associated with a nut part", can be obtained by the operation,

$$\Pi_{\texttt{part\#}}(\sigma_{\texttt{part\_description="nut"}}(\texttt{stock}))$$

The result of this compound operation is,

nut-part

part#
P1
P4

The above five operations are sufficient to express any relational algebra query. Some forms of queries are common enough however, to deserve a special notation, thereby simplifying the query.

Relation *intersection* is denoted by the operator $\cap$. The intersection operator can be built from the difference operator, i.e.,

$$r1 \cap r2 \equiv r1 - (r1 - r2)$$

The *theta join* operation is a binary operation that combines the select and cartesian product operation. A theta join operation is denoted by the symbol $\bowtie_\Theta$, where the subscript Θ represents the predicate that is to be used by the select operation. The theta join operation is built up from the cartesian product and select operations as follows,

$$r1 \bowtie_\Theta r2 \equiv \sigma_\Theta(r1 \times r2)$$

The *natural-join* operation, denoted by the symbol $\bowtie$, is a specialisation of the theta join which forces equality on those attributes that appear in both relations. Consider for example, the following natural join operation,

$$\texttt{stock} \bowtie \texttt{location}$$

This is equivalent to

$$\texttt{stock} \bowtie_{\texttt{stock.part\#=location.part\#}} \texttt{location}$$

which in turn is equivalent to,

$$\sigma_{\texttt{stock.part\#=location.part\#}}(\texttt{stock} \times \texttt{location})$$

The result of this operation is,

stock-loc

stock. *part#*	stock. part_description	stock. quantity_in_stock	*location.* *part#*	location. shelf
P1	nut	5000	P1	A
P2	bolt	8500	P2	B

10.4 Rationale for the Relational Database Design

Relational databases were devised by Codd [35] to overcome the problem of application programs being dependent on the representation of data. With the hierarchical and network models, application programs are dependent on the links designed into a database rather than on the data. Techniques for deriving the

relation schemes of a database have been given by Codd [35] and Kent [85]. The process of deriving relations from a description of the data that is to be stored is called *normalisation*. There are five main normal forms, each normalisation step being a refinement of the normalisation. In this thesis only the first four normal forms will be used.

The first normal form involves fixing the length of the schemes. This is necessary because relational database theory does not allow varying length schemes. The second and third normal forms ensure that there is a consistent relationship between key and non-key attributes. Under fourth normal form a scheme cannot contain two or more independent multivalued facts relating to another attribute. Each of these normal forms will be briefly discussed below.

10.4.1 Four Normal Forms

Consider a scheme of the form,

$$\texttt{actress}(\textit{actress-name}, \{\textit{film-title}\})$$

The attribute in curly brackets $\{\ldots\}$ can occur zero or more times for each instance of the attribute *actress-name*. The length of the scheme is fixed by making *film-title* have only one value per tuple. In order to do this, the value of the attribute *actress-name* has to appear several times in the relation. This means that instead of having a relation of the form,

actress

actress-name	*film-title*
Marilyn Monroe	Bus Stop, The Misfits
Elizabeth Taylor	National Velvet

the following relation is used,

actress

actress-name	*film-title*
Marilyn Monroe	Bus Stop
Marilyn Monroe	The Misfits
Elizabeth Taylor	National Velvet

With second normal form, a non-key attribute must relate to all the key attributes and not just a subset. When a non-key attribute relates to only a subset of the key attributes then that relation scheme is decomposed into other relation schemes where the non-key attribute is dependent on the whole key. Consider for example, the scheme,

store(*part#*, *warehouse*, qty, warehouse-addr)

The attributes that form the primary key are in slanted type face. The non-key attribute warehouse-addr relates only to the key attribute *warehouse*. Therefore the scheme store does not conform to second normal form. To do this, store is decomposed into the following two schemes,

store(*part#*, *warehouse*, qty)

warehouse(*warehouse*, warehouse-addr)

With third normal form, a non-key attribute cannot relate to another non-key attribute and not to the key attribute. Consider for example, the relation scheme,

worker(*employee*, department, location)

The attribute location relates to the attribute department and not to *employee*. As a result the relation scheme worker is decomposed into the following two schemes,

staff(*employee*, department)

work-area(*department*, location)

Consider the scheme

employee-rel(*employee*, *skill*, *language*)

This scheme conforms to the first three normal forms, but there is no relation between the attributes *skill* and *language*. As a result, this scheme violates fourth normal form. To comply with forth normal form, the scheme *employee-rel* is decomposed into the following schemes,

employee-skill(*employee*, *skill*)

employee-lang(*employee*, *language*)

10.4.2 Relations for Modules

In order to create a relational database structure for inter-module code analysis, it is necessary to describe the data that the relational database will have to record. In order to do this, the idea of *abstract syntax* from the work on the formal definition

of programming languages will be used. The full abstract syntax being used in this thesis is given in Appendix C. Portions of this description will be introduced in this chapter as they are needed.

With the module languages, a program is composed of a collection of global modules. This is denoted by the description:

Program :: *s-program* : set of *Global-Module*

Global-Module :: *s-global-module* : *Module-Entity*

where

$$inv\text{-}Global\text{-}Module(mk\text{-}Global\text{-}Module(gm)) \quad \triangleq \quad region\#(gm) = 0$$

Module-Entity :: *s-exports* : *Exported-Entities*
s-imports : *Imported-Entities*
s-region# : $\mathbb{N}$
s-region : *Region*

From this description we get the following scheme,

$$\texttt{system1}(sys\text{-}id, \{mod\text{-}id\}, \{\langle Ents \rangle\}\})$$

The attributes delimited by angled brackets $\langle \ldots \rangle$ are attributes whose full elaboration is not yet relevant.

The record structure given above, is a variable length structure. Therefore this record structure must be made to conform to first normal form by making it a fixed length record. In order to do this, the multiple occurrences are replaced by single occurrences. This creates the following relational scheme,

system2(*sys-id*, *mod-id*, ⟨*Ents*⟩)

The attributes enclosed by ⟨...⟩ apply only to the key attribute *mod-id*. To make the scheme system2 conform to second normal form it can be decomposed into the following schemes,

program-components(*sys-id*, *mod-id*) (*1)

modules1(*mod-id*, ⟨*Ents*⟩)

An implicit assumption behind this decomposition of system2 is that a global module can be uniquely identified by its identifier alone. This is a valid assumption if the database is to record information on only one system. If a database is to record information on more than one system, then the attribute *sys-id* is also needed to guarantee that a global module is uniquely selected. In this thesis we will assume that the relational database is to record information on only one system.

The attribute ⟨*Ents*⟩ can be expanded to

{⟨*Exp-Ents*⟩}, {⟨*Imp-Ents*⟩}, ⟨*Region#*⟩, ⟨*Region*⟩

This means that the scheme modules1 now becomes

modules1(*mod-id*, {⟨*Exp-Ents*⟩}, {⟨*Imp-Ents*⟩}, ⟨*Region#*⟩, ⟨*Region*⟩)

This scheme has two variable length attributes, namely: ⟨*Exp-Ents*⟩, ⟨*Imp-Ents*⟩. This scheme is normalised with respect to first normal form giving,

modules2(*mod-id*, ⟨*Exp-Ents*⟩, ⟨*Imp-Ents*⟩, ⟨*Region#*⟩, ⟨*Region*⟩)

This scheme satisfies first and second normal forms, but the non-key attributes are independent of each other (except for ⟨*Region#*⟩, and ⟨*Region*⟩), and so `modules2` can be normalised with respect to third normal form. By normalising the scheme `modules2` in this way, the following schemes are derived:

`mod-exports`(*mod-id*, ⟨*Exp-Ents*⟩)

`mod-imports`(*mod-id*, ⟨*Imp-Ents*⟩)

`mod-region`(*mod-id*, ⟨*Region#*⟩, ⟨*Region*⟩)

Each of these schemes is further refined below.

Consider the first scheme `mod-exports`. The abstract syntax describes the field *s-exports* as being of type *Exported-Entities*, and *Exported-Entities* is described as,

Exported-Entities :: *s-exported-entity* : **map** *Entity* **to** *Module-Set*

Entity :: *s-entity-id* : *Entity-Id*

Module-Set = **set of** *Module-Id*

This can be represented by a scheme of the form,

`mod-exports2`(*mod-id*, {*ent-id*, {*imp-mod-id*}})

This record structure describes a variable length record, so it has to be normalised with respect to first normal form. This creates the scheme,

`exports`(*mod-id*, *ent-id*, *imp-mod-id*) (*2)

If the database is to be used for languages that allows overloading of identifiers, then the abstract syntax for *Entity* would have to be more detailed, e.g.,

Entity :: *s-entity-id* : *Entity-Id*

s-entity-class : *Entity-Class*

Entity-Class = *Constant* | *Type* | *Variable* | *Routine* | *Module*

and the exports scheme resulting from this new description would have to have an attribute for *Entity-Class*. For simplicity the database schema being described will assume that overloading of entities is not being allowed.

Consider now the scheme mod-imports. The record *Module-Entity* describes the field for imported entities as being of type *Imported-Entities*. The VDM description for *Imported-Entities* is

Imported-Entities :: *s-imported-entities* : map *Module-Id* to *Entity-Set*

Entity-Set = set of *Entity*

The *Module-Id* referred to in the abstract syntax for *Imported-Entities* refers to the supplier module that is providing the entities in *Entity-Set*. Therefore we obtain the following scheme

mod-imports2(*mod-id*, {*exp-mod-id*, {*ent-id*}})

By normalising mod-imports2 with respect to first normal form the following scheme is derived

imports(*mod-id*, *exp-mod-id*, *ent-id*) (*3)

10.4.3 The Relations for a Region

The last scheme to be considered as a result of having decomposed the scheme `modules2` is `mod-region`. The attribute, $\langle Region\# \rangle$, is a positive integer that corresponds to the block number. The attribute $\langle Region \rangle$ corresponds to the following abstract syntax,

$$
\begin{aligned}
Region :: \; & s\text{-}constants : \text{map } Constant\text{-}Id \text{ to } Constant\text{-}Set \\
& s\text{-}types : \text{map } Type\text{-}Id \text{ to } Type\text{-}Set \\
& s\text{-}variables : \text{map } Variable\text{-}Id \text{ to } Variable\text{-}Set \\
& s\text{-}routines : \text{map } Routine\text{-}Id \text{ to } Routine\text{-}Set \\
& s\text{-}modules : \text{map } Module\text{-}Id \text{ to } Module\text{-}Set \\
& s\text{-}body : Body
\end{aligned}
$$

$$Constant\text{-}Set = \text{set of } Constant\text{-}Entity$$

$$Type\text{-}Set = \text{set of } Type\text{-}Entity$$

$$Variable\text{-}Set = \text{set of } Variable\text{-}Entity$$

$$Routine\text{-}Set = \text{set of } Routine\text{-}Entity$$

$$Module\text{-}Set = \text{set of } Module\text{-}Entity$$

This means that the scheme `mod-region` expands to

$$\texttt{mod-region}(mod\text{-}id, region\#, \langle Constant \rangle, \langle Type \rangle, \langle Variable \rangle, \langle Routine \rangle, \langle Module \rangle, \langle Body \rangle)$$

This scheme does not satisfies third normal form, because the attributes enclosed by the angled brackets are independent of each other. Normalising `mod-region` with respect to third normal form gives the schemes:

mod-const(*mod-id*, *region#*, ⟨*Constant*⟩)

mod-type(*mod-id*, *region#*, ⟨*Type*⟩)

mod-var(*mod-id*, *region#*, ⟨*Variable*⟩)

mod-rout(*mod-id*, *region#*, ⟨*Routine*⟩)

mod-mod(*mod-id*, *region#*, ⟨*Module*⟩)

mod-body(*mod-id*, *region#*, ⟨*Body*⟩)

With the scheme mod-mod, the same form of decomposition that is described in subsection 10.4.2 can be performed to obtain the schemes:

local-module(*mod-id*, *region#*, *local-mod-id*, c-region#) (*4)

local-module-exports(*mod-id*, *region#*, *local-mod-id*, *ent-id*) (*5)

local-module-imports(*mod-id*, *region#*, *local-mod-id*, *ent-id*) (*6)

The relation scheme local-module-exports does not have to name the modules that the entities are exported to because it has to be the module containing the local module declaration. This is because of the definition of a module given on page 29.

Finally consider the scheme mod-body. The body of a module or routine is a sequence of statements. For the purposes of inter-module code analysis the body is considered in terms of the entities used. The attribute ⟨*Body*⟩ in mod-body corresponds to the abstract syntax

Body :: *s-constants-used* : set of *Constant-Id*

s-type-used : set of *Type-Id*

s-variable-used : set of *Variable-Id*

s-routines-used : set of *Routine-Id*

This means that the scheme mod-body expands to,

mod-body(*mod-id*, *region#*, {*const-id*}, {*type-id*}, {*variable-id*}, {*routine-id*})

Normalising this with respect to first normal form gives,

mod-body2(*mod-id*, *region#*, *const-id*, *type-id*, *variable-id*, *routine-id*)

The attributes for each of the different classes of entities are independent of each other. Therefore this scheme can be normalised with respect to fourth normal form to give the following schemes,

constants-used(*mod-id*, *region#*, *const-id*)	(*7)
types-used(*mod-id*, *region#*, *type-id*)	(*8)
variables-used(*mod-id*, *region#*, *variable-id*)	(*9)
routines-used(*mod-id*, *region#*, *routine-id*)	(*10)

10.4.4 The Relations for Type Entities

A type entity is described in the abstract syntax as follows,

Type-Entity :: *s-type* : *Type-Constructor*

In many programming languages, it is possible for a type to be constructed from other types. For example, the following Modula-2 fragment introduces a record type with a field that is an array of set elements, and the indices of the array are specified as a subrange.

```
RECORD
    Field:  ARRAY [1..10] OF BITSET
END
```

In order to deal with this, each type is given a type number that is unique within a given module. In this way, combining a *mod-id* attribute with a *type#* attribute will uniquely determine a type. In the above example, the record could have the type number *1*, the array *2*, the subrange *3* and the set *4*. Each form of type requires different information to be recorded, e.g., a record needs to store information on each of the fields, an array needs to store information on each of the index types and information on the element type, etc. This requires that different relations be created for each of the types. (These relations are given in Appendix D.) To aid the searching of these relations, a special relation called `type` has been created. The `type` relation has the scheme,

`type`(*mod-id*, *type#*, `type-form`) (*11)

The value for `type-form` indicates which relation to interrogate for more information on the type. For the Modula-2 record type given above, the following instance of the `type` relation is created.

type

mod-id	*type#*	type-form
M	*1*	RECORD
M	*2*	ARRAY
M	*3*	SUBRANGE
M	*4*	SET

10.4.5 The Relation for Entities Declared in a Region

Consider now the scheme mod-type that was obtained after decomposing the scheme mod-region. The field in *Region* that corresponds to types has the form

map *Type-Id* to *Type-Set*

where

Type-Set = set of *Type-Entity*

Type-Entity :: *s-type* : *Type-Constructor*

The *Type-Constructor* information is recorded in the database by giving a type declaration number and using the type relation in association with the relations needed for the different forms of types. Therefore the scheme for a type declaration becomes,

type-dec(*mod-id*, *region#*, *ent-id*, type#) (*12)

The schemes mod-const and mod-variable can be expanded in a similar way to create the schemes,

constant-dec(*mod-id*, *region#*, *ent-id*, type#) (*13)

variable-dec(*mod-id*, *region#*, *ent-id*, type#) (*14)

Finally, the scheme mod-rout will be refined. The unexpanded attribute ⟨*Routine*⟩ is associated with the abstract syntax for *Routine-Entity*.

Routine-Entity :: *s-formal-parameters* : *Parameters*

s-result-type : *Type-Constructor*

s-region# : $\mathbb{N}$

s-region : *Region*

Parameters = map *Para-Id* to *Type-Constructor*

This creates a variable length record structure of the form,

mod-rout2(*mod-id*, *region#*, *ent-id*, {*para-id*, p-type#}, r-type#, c-region#, ⟨*Region*⟩)

Normalising this structure with respect to first normal form gives,

mod-rout3(*mod-id*, *region#*, *ent-id*, *para-id*, p-type#, r-type#, c-region#, ⟨*Region*⟩)

This record structure does not conform to second normal form because the attributes r-type#, c-region#, ⟨*Region*⟩ to not relate to the key attribute *para-id*. Normalising mod-rout3 with respect to second normal form gives the schemes,

para-id-rel(*mod-id*, *region#*, *ent-id*, *para-id*, p-type#) (*15)

mod-rout4(*mod-id*, *region#*, *ent-id*, r-type#, c-region#, ⟨*Region*⟩)

Normalising the scheme mod-rout4 with respect to third normal form we get the schemes,

type-of-routine(*mod-id*, *region#*, *ent-id*, r-type#) (*16)

mod-rout5(*mod-id*, *region#*, *ent-id*, c-region#, ⟨*Region*⟩)

The scheme mod-rout5 can be expanded and decomposed in the same way that mod-region is on page 220 but this would result in unnecessary relations being created. The primary key for the schemes dealing with entity declarations consists of a declaration of: *mod-id*, the identifier of the global module in which the entity is declared, *region#*, the number of the actual region in which the entity is declared, and *ent-id*, the identifier of the entity being declared. Any entities that are declared within a routine can therefore be represented by the relations already derived. As a result the scheme mod-rout5 become,

region-of-routine(*mod-id*, *region#*, *ent-id*, c-region#) (*17)

10.4.6 An Example

```
MODULE EntityDeclarations;

  VAR v1: INTEGER;

  PROCEDURE P1;
    VAR v1: CHAR;
  BEGIN
    ...
  END P1;
BEGIN
  ...
END EntityDeclarations.
```

Figure 10.6: A Modula-2 Program Module to Demonstrate Entity Declarations

To demonstrate the validity of this database schema, consider the Modula-2 program module given in Figure 10.6. The relations for this module declaration

are given in Figure 10.7. The variable v1 declared in routine P1 is distinguished from the state variable v1 by the region number used as the primary key.

program-components (*1)

sys-id	*mod-id*
EntityDeclarations	EntityDeclarations

type-of-routine (*16)

mod-id	*region#*	*ent-id*	r-type#
EntityDeclarations	*0*	P1	*2*

variable-dec (*14)

mod-id	*region#*	*ent-id*	type#
EntityDeclarations	*0*	v1	*1*
EntityDeclarations	*1*	v1	*3*

type (*11)

mod-id	*type#*	type-form
EntityDeclarations	*1*	QUALIDENT
EntityDeclarations	*2*	QUALIDENT
EntityDeclarations	*3*	QUALIDENT

qualident

mod-id	*type#*	ent-id
EntityDeclarations	*1*	INTEGER
EntityDeclarations	*2*	$VOID$
EntityDeclarations	*3*	CHAR

Figure 10.7: The Relations for the Entity Declarations in Figure 10.6

10.5 The Full Relational Database Scheme

Appendix D gives the full relational database schema that is proposed for inter-module code analysis. It does not cater for inheritance or instantiation. This is because this thesis has concentrated on analysing systems with respect to the uses and local-to dependencies.

The relations that are used to record the information about the different forms of types are given in this appendix. One of these relations is subrange_delimiters which records the names of the constants and routines (function calls) that are used in the expressions that are the bounds of a subrange. This relation is needed because no relations have been created to cater for an expression, and it is possible to have declarations of the form,

```
C: constant INTEGER:= 10;
type S is range C..2*C;
```

All of the information that is to be stored in the database can be generated by a suitably tailored compiler front end.

10.6 A Prototype Database

A prototype database was implemented in C-Prolog on a Sun 3–50 so that the relational database schema described in this chapter could be validated. With the prototype database Prolog facts are used to represent the tuples of a relation, and a collection of same named Prolog facts constitute a relation. To demonstrate how this database can be interrogated the entity groups given in Table 9.1 (on page 193) Table 9.2 (on page 194) will be derived.

In order to find the client views of UnixSupport2 the following select operations need to be performed,

$$\sigma_{\texttt{mod-id='UnixSupport2'}}(\texttt{exports})$$

```
exports('UnixSupport2', 'EOL', '$All$').
exports('UnixSupport2', 'ReadOnly', '$All$').
exports('UnixSupport2', 'StringPointer', '$All$').
exports('UnixSupport2', 'WriteOnly', '$All$').
exports('UnixSupport2', 'maxFileNameLength', '$All$').
exports('UnixSupport2', stderr, '$All$').
exports('UnixSupport2', stdin, '$All$').
exports('UnixSupport2', stdout, '$All$').
exports('UnixSupport2', strcmp, '$All$').
exports('UnixSupport2', strlen, '$All$').

imports('IO', 'UnixSupport2', 'EOL').
imports('IO', 'UnixSupport2', 'ReadOnly').
imports('IO', 'UnixSupport2', 'StringPointer').
imports('IO', 'UnixSupport2', strlen).
imports('ModuleHandling', 'UnixSupport2', 'StringPointer').
imports('ModuleHandling', 'UnixSupport2', 'maxFileNameLength').
imports('ModuleHandling', 'UnixSupport2', qsort).
imports('ModuleHandling', 'UnixSupport2', stderr).
imports('ModuleHandling', 'UnixSupport2', stdout).
imports('ModuleHandling', 'UnixSupport2', strcmp).
imports('ModuleHandling', 'UnixSupport2', strlen).
imports('OptionHandling', 'UnixSupport2', 'StringPointer').
imports('OptionHandling', 'UnixSupport2', 'maxFileNameLength').
imports('OptionHandling', 'UnixSupport2', stderr).
imports('OptionHandling', 'UnixSupport2', strcmp).
imports('OptionHandling', 'UnixSupport2', strlen).
```

Figure 10.8: A Collection of Prolog Facts Constituting the `imports` and `exports` Relations

```
create_import_set(Mod_Id, Ent_Id, Import_Set):-
   findall(Importing_Mod_Id,
           imports(Importing_Mod_Id, Mod_Id, Ent_Id),
           Import_Set).

create_raw_import_list(Module_Id, [Entity_Id | Import_List]):-
   exports(Module_Id, Entity_Id, _),
   findall(Import_Set,
           create_import_set(Module_Id, Entity_Id, Import_Set),
           Import_List).

create_raw_entity_groups(Module_Id, [Entity_List | Import_List]):-
   create_raw_import_list(Module_Id, [_ | Import_List]),
   findall(Entity_Id,
           create_raw_import_list(Module_Id, [Entity_Id | Import_List]),
           Entity_List).

get_entity_groups(Module_Id, Entity_Groups):-
   findall(Entity_List,
           create_raw_entity_groups(Module_Id, Entity_List),
           Tmp_List),
   strip_list(Tmp_List, Entity_Groups).

get_and_display_entity_groups(Module_Id):-
   get_entity_groups(Module_Id, Entity_Groups),
   display_list_of_lists(Entity_Groups).
```

Figure 10.9: Prolog Database Interrogation Programs

```
Script started on Tue Nov 28 20:57:04 1989
ws_eddy 1>prolog
C-Prolog version 1.5
| ?- [startup].
startup consulted 7644 bytes 0.833333 sec.

yes
| ?- load.
IO consulted 204 bytes 0.0500009 sec.
ModuleHandling consulted 280 bytes 0.0666667 sec.
OptionHandling consulted 200 bytes 0.0333334 sec.
UnixSupport2 consulted 444 bytes 0.0833341 sec.
prog_components consulted 256 bytes 0.0500004 sec.
grouping1 consulted 688 bytes 0.1 sec.

yes
| ?- get_and_display_entity_groups('UnixSupport2').
[[EOL,ReadOnly],[IO]]
[[stdout],[ModuleHandling]]
[[stdin,WriteOnly],[]]
[[maxFileNameLength,stderr,strcmp],[ModuleHandling,OptionHandling]]
[[StringPointer,strlen],[IO,ModuleHandling,OptionHandling]]

yes
| ?- halt.

[ Prolog execution halted ]
ws_eddy 2>^D
script done on Tue Nov 28 20:57:58 1989
```

Figure 10.10: A Unix Script Recording a Database Query

$$\sigma_{\texttt{exp-mod-id='UnixSupport2'}}(\texttt{imports})$$

The result of these select operations is given in Figure 10.8. The relations in Figure 10.8 are then used in a normal join operation of the form,

$$\sigma_{\texttt{mod-id='UnixSupport2'}}(\texttt{exports}) \bowtie \sigma_{\texttt{exp-mod-id='UnixSupport2'}}(\texttt{imports})$$

This normal join operation forces equality on the `ent-id` attributes in the `exports` and `imports` relations. Projecting the resulting relation with respect to the `ent-id` and `imp-mod-id` attributes creates a relation that records the client views of the module `UnixSupport2`. This information can then be used to derive the entity groups and module sets in Tables 9.1 and 9.2.

Figure 10.9 gives the Prolog goals that interrogate the given relations. A call to the goal `get_and_display_entity_groups`, with the name of a module as an argument will produce the entity groups and the associated importing module set for the named module. This is demonstrated in Figure 10.10 which contains a Unix script file that records the execution of,

```
get_and_display_entity_groups('UnixSupport2').
```

The script file shows that the first Prolog command is to consult the file `startup`. This file contains the declaration of several Prolog goals that are used to interrogate Prolog facts. These goals are based on those given by Clocksin and Mellish [34]. A list data structure is used to simulate a set. Therefore the result of the goal `get_and_display_entity_groups` is shown as a list.

The `findall` goal finds all the facts (tuples) that satisfy some constraint. As a result this goal is used extensively in the goals given in Figure 10.9. The goal

create_import_set finds the sets of modules that import a named entity, while the goal create_raw_import_list ensures that the set of importing modules for each of the exported entities is derived. The goal create_raw_entity_groups derives the set of entities that a set of modules import, e.g., the module IO imports the set of entities EOL and ReadOnly. Finally the goal get_entity_groups ensures that there is only one occurrence of each (entity-group, module-group) tuple by stripping out any tuples that contain the same elements but where the ordering is different.

The database structure has been shown to be capable of generating the information needed for inter-module code analysis. All the relation manipulations described in Chapter 11 have been implemented in the prototype database.

Chapter 11

The Use of a Relational Database

11.1 Introduction

Chapter 10 describes the process by which the relational database given in Appendix D was derived. This chapter will show how this database schema can be used to implement the graph manipulation operations that are described in Chapters 5–9.

Section 11.2 shows how the relational database can be used to derive the information that is recorded in the module-to-module, entity-to-module and entity-to-entity interconnection graphs. Section 11.3 shows how the graph operations that are described in Chapter 5. can be implemented as operations on the relational database.

11.2 The Graph Structure

Figure 4.6 on page 64 gives the description of a graph structure using VDM. This description, can be used to develop a relation scheme for an interconnection graph in the same way that the abstract syntax in Appendix C is used to derive the relational database schema.

A graph is described as being a collection of related nodes and edges, i.e.,

Graph :: *nodes* : set of *Node*
edges : set of *Edge*
where

$$inv\text{-}Graph(mk\text{-}Graph(nodes, edges)) \triangleq$$
$$\forall e \in edges \cdot (start\text{-}node(e) \in nodes \wedge stop\text{-}node(e) \in nodes)$$

This description of a graph gives rise to the scheme,

$$\texttt{graph-scheme1}(\{\langle Node\rangle\}, \{\langle Edge\rangle\})$$

Normalising this scheme with respect to first normal form gives the scheme,

$$\texttt{graph-scheme2}(\langle Node\rangle, \langle Edge\rangle)$$

This scheme does not conform to fourth normal form as there is no relation between the attributes $\langle Node\rangle$, and $\langle Edge\rangle$. Normalising this scheme with respect to fourth normal form gives,

$$\texttt{node-scheme1}(\langle Node\rangle)$$
$$\texttt{edge-scheme1}(\langle Edge\rangle)$$

Consider first the scheme edge-scheme1. The attribute ⟨*Edge*⟩ is associated with the edges in a graph. In Figure 4.6 an edge is described as,

Edge :: *start-node* : *Node*
stop-node : *Node*
dependency : *Dependency*

as a result, the scheme edge-scheme1 expands to,

edge-scheme1(⟨*Start-Node*⟩, ⟨*Stop-Node*⟩, *dependency*)

Each of the attributes ⟨*Start-Node*⟩ and ⟨*Stop-Node*⟩ correspond to a node in the graph. Therefore these attributes conform to the VDM description,

Node :: *node-name* : *Name*
node-label : *Label*

Label :: *entity-class* : *Class*
entity-source : *Source*
entity-declaration-block : *Block-Number*

and so edge-scheme1 expands to the scheme,

edge(*ent-id1*, *ent-class1*, *mod-id1*, *region1#*,
ent-id2, *ent-class2*, *mod-id2*, *region2#*,
dependency)

The attributes *ent-id1*, *ent-class1*, *mod-id1* and *region1#* are needed to uniquely identify the entity associated with the start-node of an edge, while the attributes *ent-id2*, *ent-class2*, *mod-id2* and *region2#* uniquely identify the stop-node. This information is obtainable directly from the relations that record information on the

declaration of entities, i.e., constant-dec, type-of-routine, region-of-routine type-dec, variable-dec and local-module. The global modules can be identified by the modules listed in the relation program-components. The value of the attribute *dependency* can be derived by examining the entities named as being start-nodes and stop-nodes.

Similarly, the scheme node-scheme expands to the scheme,

node(*ent-id*, *ent-class*, *mod-id*, *region#*)

Therefore a graph is represented by two relations, one an instance of the scheme node, and the other an instance of the scheme edge.

Some examples are given below to show how to derive the dependencies between the named modules.

11.2.1 Dependency Derivation

Deriving the dependencies that can appear in the module-to-module, entity-to-module or the entity-to-entity interconnection graphs initially appears to be a costly operation, because a large system can contain thousands of entities and the dependencies that each of these entities is involved with has to be determined. Each of these dependencies has properties which can help eliminate whole classes of entities as is demonstrated below. Some of the different dependencies that can appear in the module-to-module, entity-to-module or the entity-to-entity interconnection graphs will be derived by analysing the relational database schema given in Appendix D.

Firstly consider the module-to-module interconnection graph. In Chapter 6 this graph is shown to record the `instantiates-to`, `inherits-from`, `uses` and `local-to`, The relational database schema in Appendix D docs not cater for instantiation or inheritance, therefore only the `local-to` and `uses` dependencies will be considered here.

The `uses` dependency exists between global modules only. Therefore the only relations that need to be used are those that show the dependence between global modules. This is done by the relations `exports` and `imports` and shows the global modules that aquire entities provided by other global modules. To determine which global modules are connected by a `uses` dependency, projection operations are performed on the `exports` and `imports` relations. Consider first the `imports` relation. This relation shows which modules explicitly import an entity from another global module. The `imports` relation has the scheme,

$$\texttt{imports}(\textit{dest\_mod\_id}, \textit{src\_mod\_id}, \textit{ent\_id})$$

In order to determine the modules involved in a uses dependency, the following projection operation can be used,

$$\Pi_{\texttt{dest\_mod\_id,src\_mod\_id}}(\texttt{imports})$$

Each tuple in the resulting relation denotes an edge that represents a `uses` dependency, with the module named in the attribute *dest_mod_id* being the start-node and the the module named in the attribute *src_mod_id* being the stop-node.

When selective export has been employed, the `exports` relation can be used to determine the existence of a `uses` dependency. The `exports` relation shows which

entities are exported to a module. If selective export is not being represented then the special value All is recorded as the importing module. The exports relation can only be used to determine the existence of a uses dependency if selective export is used. The scheme for the exports relation is,

$$\texttt{exports}(\textit{exp\_mod\_id}, \textit{ent\_id}, \textit{imp\_mod\_id})$$

When the entity named in the attribute *ent_id* is not selectively exported, the value All is stored in the attribute *imp_mod_id*. Therefore, only those tuples that have a module named in *imp_mod_id* denote a uses dependency. To extract the tuples from exports that show a uses dependency, the following combination of a project and select operation can be used,

$$\sigma_{\texttt{imp\_mod\_id} \neq \texttt{\$All\$}}(\Pi_{\texttt{exp\_mod\_id,imp\_mod\_id}}(\texttt{exports}))$$

In order to find the global modules that are isolated within a system, the relations exports, imports and program_components are used. The following four operations find all global modules that are subject to a uses dependency,

$$\Pi_{\texttt{dest\_mod\_id}}(\texttt{imports})$$

$$\Pi_{\texttt{src\_mod\_id}}(\texttt{imports})$$

$$\Pi_{\texttt{exp\_mod\_id}}(\texttt{exports})$$

$$\sigma_{\texttt{imp\_mod\_id} \neq \texttt{\$All\$}}(\Pi_{\texttt{imp\_mod\_id}}(\texttt{exports}))$$

By performing a union of these four relations, a single relation is created containing the names of all the global modules subject to a uses dependency. Call this relation used_mods. The result of the following project operation is the name of all the

global modules in a system,

$$\Pi_{\texttt{mod\_id}}(\texttt{program\_components})$$

If this relation is called `mods` then the names of the isolated modules is obtained by the operation,

$$\texttt{mods} - \texttt{used\_mods}$$

The `local-to` dependency exists between a local module and another local module, or a local module and a global module. In order to find the local and global modules that are involved in a `local-to` dependency, the following operation can be used,

$$\Pi_{\texttt{loc-mod-id,mod-id}}(\sigma_{\texttt{region\#=0}}(\texttt{local-module}))$$

Each tuple in the relation resulting from this operation represents an edge denoting a `local-to` dependency. The select operation extracts thoses tuples that relate to a local module declared in the outermost block of a global module, while the project operation extracts the module names.

In order to find the local-modules connected by a `local-to` dependency the `local-module` relation is again used, but this time it is interrogated differently. Identifying the local modules that are connected by a `local-to` dependency takes two steps.

1. Find the region associated with each local module.

2. Find any local modules declared in this region.

The result of a project operation of the form,

$$\Pi_{\texttt{mod-id,loc-mod-id,c-region\#}}(\texttt{local-module})$$

will result in a relation `lm-region` which shows the region associated with each local module. A theta join operation of the form,

$$\texttt{lm-region} \bowtie_{\Theta} \texttt{local-module}$$

where Θ is the predicate,

$$\texttt{lm-region.mod-id} = \texttt{local-module.mod-id} \wedge$$
$$\texttt{lm-region.c-region\#} = \texttt{local-module.region\#}$$

will result in a relation showing which local modules are declared within the region of another local module. A project operation of the form,

$$\Pi_{\substack{\texttt{local-module.loc-mod-id, local-module.mod-id, local-module.region\#,} \\ \texttt{lm-region.loc-mod-id, lm-region.mod-id, lm-region.region\#}}} (\ldots)$$

results in a relation where each tuple shows modules that are connected by a `local-to` dependency.

As an example, consider the Modula-2 module declaration in Figure 11.1. The module-to-module interconnection graph for these module declarations is given in Figure 11.2. The `local-module` relation for these declarations is given in Figure 11.3.

```
IMPLEMENTATION MODULE GM1;
  MODULE LM1;
    MODULE LM2;
      ...
    END LM2;
    MODULE LM3;
      ...
    END LM3;
  END LM1;
  MODULE LM4;
    ...
  END LM4;
END GM1.

IMPLEMENTATION MODULE GM2;
  MODULE LM1;
    MODULE LM2;
      ...
    END LM2;
  END LM1;
END GM2.
```

Figure 11.1: Modula-2 Module Declarations for `local-to` Dependency Derivation

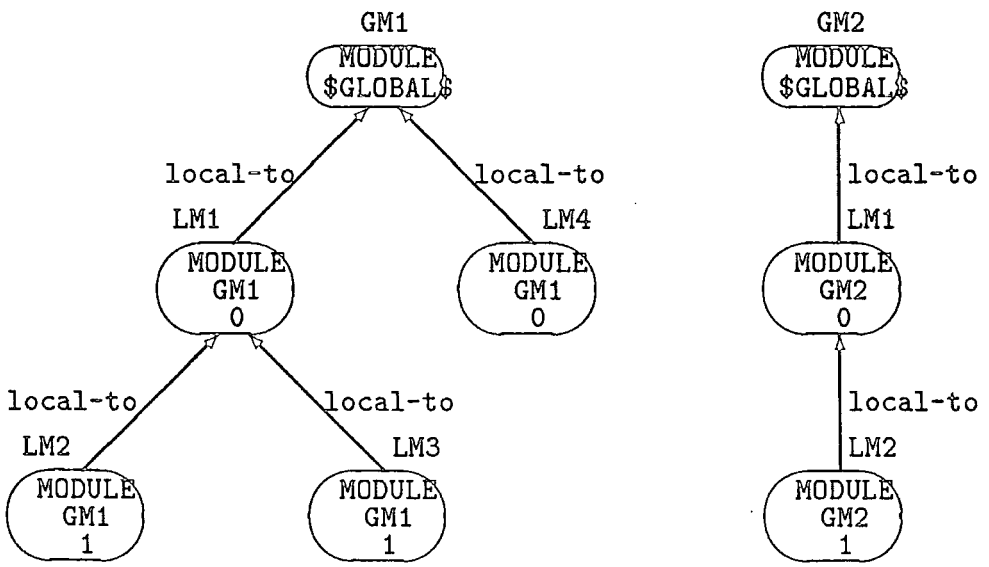


Figure 11.2: The Module-to-Module Interconnection Graph for Figure 11.1

local-module

mod-id	*region#*	*loc-mod-id*	c-region#
GM1	*0*	LM1	*1*
GM1	*1*	LM2	*2*
GM1	*1*	LM3	*3*
GM1	*0*	LM4	*4*
GM2	*0*	LM1	*1*
GM2	*1*	LM2	*2*

Figure 11.3: The `local-to` Relation for the Module in Figure 11.1

The result of the operation,

$$\Pi_{\texttt{mod-id,loc-mod-id,c-region\#}}(\texttt{local-module})$$

is the relation,

lm-region

mod-id	*loc-mod-id*	c-region#
GM1	LM1	*1*
GM1	LM2	*2*
GM1	LM3	*3*
GM1	LM4	*4*
GM2	LM1	*1*
GM2	LM2	*2*

Performing the theta join operation gives,

lm-reg mod-id	lm-reg region#	lm-reg c-region#	loc-mod mod-id	loc-mod region#	loc-mod loc-mod-id	loc-mod c-region#
GM1	LM1	*1*	GM1	*1*	LM2	*2*
GM1	LM1	*1*	GM1	*1*	LM3	*3*
GM2	LM1	*1*	GM2	*1*	LM2	*2*

Therefore the edge relation will have three tuples of the form,

ent id1	ent class1	mod id1	region1#	ent id2	ent class2	mod id2	region2#	dependency
LM2	MODULE	GM1	*1*	LM1	MODULE	GM1	*0*	local-to
LM3	MODULE	GM1	*1*	LM1	MODULE	GM1	*0*	local-to
LM2	MODULE	GM2	*1*	LM1	MODULE	GM2	*0*	local-to

Consider now the entity-to-module interconnection graph. In Chapter 7 this graph is shown to record the `imported`, `exported`, `inherited` and `injected` dependencies. These dependencies all show a dependency between an entity declared within a module and some other module. These dependencies are characterised by an edge of the form,

(*entity*, *module*)

As the relational database schema being used here does not record information on inheritance, the `inherited` dependency will not be discussed here.

The dependencies `exported` and `imported` are dependencies that exist between global modules. All the information needed to record if an entity is exported to, or imported by, a module can be obtained from the `exports` and `imports` relations. For example, to determine the entities and modules involved in an `imported` dependency, the following projection operation can be used.

$$\Pi_{\texttt{mod-id,ent-id}}(\texttt{imports})$$

In order to determine the class of the imported entities, the following steps can be taken. Step 1, obtain the source of the entity. This can be done by the following projection operation

$$\Pi_{\texttt{exp-mod-id,ent-id}}(\texttt{imports})$$

Call the relation resulting from this operation source-entity. Step 2, determine the class of the entity by a theta join operation of the form,

$$\texttt{source-entity} \bowtie_{\Theta} \texttt{dec-rel}$$

where dec-rel is one of the relations, constant-dec, type-dec, variable-dec, type-of-rout or region-of-rout, and Θ is the predicate,

$$\begin{aligned} \texttt{source-entity.exp-mod-id} &= \texttt{dec-rel.mod-id} \wedge \\ \texttt{dec-rel.region\#} &= 0 \end{aligned}$$

To find the entities and modules involved in an exported dependency, the exports relation is interrogated in a similar way to the imports relation. When analysing the exports relation, it is necessary to extract the tuples that do not have the attribute value 'All' in the attribute imp-mod-id. This can be done by a select operation of the form,

$$\Pi_{\texttt{ent-id,imp-mod-id}}(\sigma_{\texttt{imp-mod-id} \neq \texttt{\$All\$}}(\texttt{exports}))$$

The *injected* dependency exists between an entity from a local module and the module containing the local module. The procedure used to derive the local-to dependency shows how to identify the encapsulating module. The relation that resulted from the theta join of lm-region and local-module on page 241 can be used to derive which entities are injected into modules. If linked-lms is the relation resulting from this theta join operation, the entities and modules involved

in an `injected` dependency can be obtained by the following operation

$$\texttt{linked-lms} \bowtie_{\Theta} \texttt{local-module-exports}$$

where Θ is the predicate,

$$\begin{aligned}
\texttt{linked-lms.local-module.mod-id} &= \texttt{local-module-exports.mod-id} \wedge \\
\texttt{linked-lms.local-module.region\#} &= \texttt{local-module-exports.region\#} \wedge \\
\texttt{linked-lms.local-module.loc-mod-id} &= \texttt{local-module-exports.loc-mod-id}
\end{aligned}$$

Finally consider the entity-to-entity interconnection graph. The dependencies recorded in this graph are those that exist between the global entities of a module. It is a characteristic of these dependencies that they are associated with an edge of the form,

$$(\textit{entity}, \textit{entity})$$

Some of the dependencies that can be recorded are those between routines and entities that are constants, types, variables and routines. These are the `uses-constant`, `uses-type`, `uses-variable` and `invoked` dependencies. The entities involved in these dependencies are the relations `constants-used`, `types-used`, `variables-used` and `routines-used`. The analysis of the entity-to-entity interconnection graph described in this thesis has confined itself to determining which global entities are used by other global entities, and so these relations are all that is needed. The existing relations can be used to determine how a global type or constant is being used within a routine, if this form of analysis is required.

Three other forms of dependencies that can appear in the entity-to-entity interconnection graph are `delimited-by`, `para-of-type` and `of-type`.

The `delimited-by` dependency exists between a type and a constant, where the constant marks a bound of a subrange type. This form of dependency is characterised by an edge of the form,

(*type*, *constant*)

The relation `subrange-delimiters` lists the constants that are used to delimit a type, and so the entities involved in a `delimited-by` dependency can be derived by interrogating this relation.

The `para-of-type` dependency exists between a routine and a type, where the type is used to declare one or more parameters of the routine. The information needed to determine which routines and which types are connected by a `para-of-type` dependency can be obtained by interrogating the `para-id-rel` and the `type-dec` relations. The `para-id-rel` relation records the `type#` value for each of the parameters of a routine, and the `type-dec` relation records the `type#` value for every type entity. Therefore, the following operation can be used to associate routines and types involved in a `para-of-type` dependency,

$$\Pi_{\texttt{para-id-rel.mod-id,para-id-rel.ent-id,type-dec.ent-id}}(\texttt{para-id-rel} \bowtie_{\Theta} \texttt{type-dec})$$

where Θ is the predicate,

$$\begin{aligned} \texttt{para-id-rel.mod-id} &= \texttt{type-dec.mod-id} \wedge \\ \texttt{para-id-rel.type\#} &= \texttt{type-dec.type\#} \wedge \end{aligned}$$

$$\texttt{para-id-rel.region\#} = 0 \wedge$$

$$\texttt{type-dec.region\#} = 0$$

The `of-type` dependency exists between entities and type entities. This dependency is characterised by an edge of the form,

$$(\textit{entity}, \textit{type})$$

Consider the case when it is required to find the constants and types involved in an `of-type` dependency. This pairing of entities can be obtained by the following operation,

$$\Pi_{\texttt{constant-dec.ent-id,type-dec.ent-id}}(\texttt{constant-dec} \bowtie_{\Theta} \texttt{type-dec})$$

where Θ is the predicate,

$$\begin{aligned}
\texttt{constant-dec.mod-id} &= \texttt{type-dec.mod-id} \wedge \\
\texttt{constant-dec.type\#} &= \texttt{type-dec.type\#} \wedge \\
\texttt{constant-dec.region\#} &= 0 \wedge \\
\texttt{type-dec.region\#} &= 0
\end{aligned}$$

Similar combinations of a theta join and a project operations can be used with the `type-dec`, `variable-dec` and `type-of-routine` to find the type, variable and routine entities involved in an `of-type` dependency.

11.3 The Graph Operations

Having derived the relations needed to represent a graph, and shown how the dependencies represented by an edge can be determined by interrogating the database, now consider how to implement the graph operations in terms of operations on these relations.

Chapter 5 describes some graph operations that are used to manipulate and reason about graphs in Chapters 6–8. Each of these operations will be considered below.

11.3.1 Subgraphs

There are two subgraph operations, each of which is a boolean operation. Consider the subgraph operation,

$$G_1(\mathcal{N}_1, \mathcal{E}_1) \sqsubseteq G_2(\mathcal{N}_2, \mathcal{E}_2) \tag{11.1}$$

This operation is true if all the nodes and edges in the graph $G_1(\mathcal{N}_1, \mathcal{E}_1)$ are also in the graph $G_2(\mathcal{N}_2, \mathcal{E}_2)$. If the relations `graph1-nodes` and `graph1-edges` represent $G_1(\mathcal{N}_1, \mathcal{E}_1)$, and the relations `graph2-nodes` and `graph2-edges` represent $G_2(\mathcal{N}_2, \mathcal{E}_2)$, then the graph operation (11.1) can be determined by the following relation operations,

$$\begin{aligned} \texttt{graph1-nodes} - \texttt{graph2-nodes} &= \textit{empty-relation} \wedge \\ \texttt{graph1-edges} - \texttt{graph2-edges} &= \textit{empty-relation} \end{aligned}$$

If all the nodes and edges in $G_1(\mathcal{N}_1, \mathcal{E}_1)$ are in $G_2(\mathcal{N}_2, \mathcal{E}_2)$, then all the tuples in the relations representing $G_1(\mathcal{N}_1, \mathcal{E}_1)$ should also exist in the relations representing $G_2(\mathcal{N}_2, \mathcal{E}_2)$. Therefore the result of the two relation difference operations should be empty relations.

For the strict subgraph operation,

$$G_1(\mathcal{N}_1, \mathcal{E}_1) \sqsubset G_2(\mathcal{N}_2, \mathcal{E}_2)$$

to be true, it is necessary for all the nodes and edges in $G_1(\mathcal{N}_1, \mathcal{E}_1)$ to be in $G_2(\mathcal{N}_2, \mathcal{E}_2)$, and $G_2(\mathcal{N}_2, \mathcal{E}_2)$ has some nodes or edges that are not in $G_1(\mathcal{N}_1, \mathcal{E}_1)$. This form of subgraph relationship can be determined by the following relation operations,

$$\begin{aligned}
(&\texttt{graph1-nodes} - \texttt{graph2-nodes} = \textit{empty-relation} \wedge \\
&\texttt{graph1-edges} - \texttt{graph2-edges} = \textit{empty-relation}) \wedge \\
(&\texttt{graph2-nodes} - \texttt{graph1-nodes} \neq \textit{empty-relation} \vee \\
&\texttt{graph2-edges} - \texttt{graph1-edges} \neq \textit{empty-relation})
\end{aligned}$$

11.3.2 Graph Union

A simple graph union operation of the form,

$$G_1(\mathcal{N}_1, \mathcal{E}_1) \sqcup G_2(\mathcal{N}_2, \mathcal{E}_2)$$

results in a new graph containing all the nodes and edges of $G_1(\mathcal{N}_1, \mathcal{E}_1)$ and $G_2(\mathcal{N}_2, \mathcal{E}_2)$. In terms of the relations representing $G_1(\mathcal{N}_1, \mathcal{E}_1)$ and $G_2(\mathcal{N}_2, \mathcal{E}_2)$

a simple graph union is implemented as the following relation operations,

$$\texttt{graph1-nodes} \cup \texttt{graph2-nodes}$$

$$\texttt{graph1-edges} \cup \texttt{graph2-edges}$$

The distributed graph union operation,

$$\bigsqcup\{G_1(\mathcal{N}_1, \mathcal{E}_1), \ldots, G_n(\mathcal{N}_n, \mathcal{E}_n)\}$$

is n − 1 applications of the simple graph union operation. So this operation can be implemented as,

$$\texttt{graph1-nodes} \cup \ldots \cup \texttt{graphn-nodes}$$

$$\texttt{graph1-edges} \cup \ldots \cup \texttt{graphn-edges}$$

11.3.3 Graph Intersection

Strict Graph Intersection

With a strict graph intersection operation,

$$G_1(\mathcal{N}_1, \mathcal{E}_1) \sqcap G_2(\mathcal{N}_2, \mathcal{E}_2)$$

a graph is created that contains all the nodes and edges that are common to both graphs. In terms of the relations representing $G_1(\mathcal{N}_1, \mathcal{E}_1)$ and $G_2(\mathcal{N}_2, \mathcal{E}_2)$, a

strict graph intersection operation is implemented as,

$$\texttt{graph1-nodes} \cap \texttt{graph2-nodes}$$

$$\texttt{graph1-edges} \cap \texttt{graph2-edges}$$

Just as with the graph union operation, the distributed strict graph intersection operation,

$$\sqcap\{G_1(\mathcal{N}_1, \mathcal{E}_1), \ldots, G_n(\mathcal{N}_n, \mathcal{E}_n)\}$$

is implemented as n − 1 applications of the simple form of strict graph intersection, i.e.,

$$\texttt{graph1-nodes} \cap \ldots \cap \texttt{graphn-nodes}$$

$$\texttt{graph1-edges} \cap \ldots \cap \texttt{graphn-edges}$$

Full Graph Intersection

A full graph intersection operation of the form,

$$G_1(\mathcal{N}_1, \mathcal{E}_1) \sqcap_+ G_2(\mathcal{N}_2, \mathcal{E}_2)$$

creates a graph containing the nodes that are common to both graphs together with the edges from both graphs whose start-node and stop-node in this set of common nodes. In terms of the relations representing the graphs, the full graph intersection operation can be implemented as follows.

Create the relation `common-nodes` that records the nodes common to both

graphs by the operations,

$$\texttt{graph-nodes1} \cap \texttt{graph-nodes2}$$

and create the relation all-edges that records all the edges from both graphs

$$\texttt{graph-edges1} \cup \texttt{graph-edges2}$$

Then derive the relations ok-start-node and ok-stop-node which records the edges with a valid start-node and stop-node respectively. These relations are created by the following operations,

$$\Pi_{\Psi}(\texttt{common-nodes} \bowtie_{\Theta 1} \texttt{all-edges})$$

$$\Pi_{\Psi}(\texttt{common-nodes} \bowtie_{\Theta 2} \texttt{all-edges})$$

where Ψ is the list of attributes,

ent-id1, ent-class1, mod-id, region#1,
ent-id2, ent-class2, mod-id, region#2,
dependency

$\Theta 1$ is the predicate,

$$\begin{aligned}
\texttt{common-nodes.ent-id} &= \texttt{all-edges.ent-id1} \\
\texttt{common-nodes.ent-class} &= \texttt{all-edges.ent-class1} \\
\texttt{common-nodes.mod-id} &= \texttt{all-edges.mod-id1} \\
\texttt{common-nodes.region\#} &= \texttt{all-edges.region\#1}
\end{aligned}$$

and $\Theta 2$ is the predicate,

$$\begin{aligned}
\texttt{common-nodes.ent-id} &= \texttt{all-edges.ent-id2} \\
\texttt{common-nodes.ent-class} &= \texttt{all-edges.ent-class2} \\
\texttt{common-nodes.mod-id} &= \texttt{all-edges.mod-id2} \\
\texttt{common-nodes.region\#} &= \texttt{all-edges.region\#2}
\end{aligned}$$

The distributed form of strict graph intersection is n − 1 applications of the simple form.

11.3.4 δ-Slicing

A δ-slicing operation,

$$\delta(\mathrm{G}(\mathcal{N}, \mathcal{E}), \mathcal{C})$$

extracts the subgraph from $\mathrm{G}(\mathcal{N}, \mathcal{E})$ in which all the edges denote the dependencies listed in the argument $\mathcal{C}$. Consider the δ-slicing operation,

$$\delta(\mathrm{G}(\mathcal{N}, \mathcal{E}), \{\texttt{dep1}\})$$

If `dep1` is not associated with the dependency `$ISOLATED$`, then the following relation operations can be performed to derive the relations that represent the desired subgraph.

The select operation,

$$\sigma_{\texttt{dependency=dep1}}(\texttt{graph-edges})$$

will create the relation that records the edges that exist in the subgraph. This relation will be called subgraph-edges. All the nodes that exist in this subgraph are start-nodes and stop-nodes of the edges. Therefore the relation recording the nodes of the subgraph can be obtained by the operation,

$$\Pi_{\texttt{ent-id1,ent-class1mod-id1region\#1}}(\texttt{subgraph-edges}) \cup$$
$$\Pi_{\texttt{ent-id2,ent-class2mod-id2region\#2}}(\texttt{subgraph-edges})$$

If dep1 represents the dependency $ISOLATED$, then we have to find the nodes that are neither a start-node or stop-node for any of the edges in a given graph. This can be done by the following operations,

$$\Pi_{\Psi}(\textit{graph-nodes} \bowtie_{\Theta} \textit{graph-edges})$$

where Ψ is the list of attributes,

$$\texttt{graph-nodes.ent-id, graph-nodes.ent-class,}$$
$$\texttt{graph-nodes.mod-id, graph-nodes.region\#}$$

and Θ is the predicate,

$$\begin{aligned}
\texttt{graph-nodes.ent-id} &\neq \texttt{graph-edges.ent-id1} \wedge \\
\texttt{graph-nodes.ent-class} &\neq \texttt{graph-edges.ent-class1} \wedge \\
\texttt{graph-nodes.mod-id} &\neq \texttt{graph-edges.mod-id1} \wedge \\
\texttt{graph-nodes.region\#} &\neq \texttt{graph-edges.region\#1} \wedge \\
\texttt{graph-nodes.ent-id} &\neq \texttt{graph-edges.ent-id2} \wedge \\
\texttt{graph-nodes.ent-class} &\neq \texttt{graph-edges.ent-class2} \wedge
\end{aligned}$$

$$\texttt{graph-nodes.mod-id} \neq \texttt{graph-edges.mod-id2} \wedge$$

$$\texttt{graph-nodes.region\#} \neq \texttt{graph-edges.region\#2}$$

A δ-slicing operation,

$$\delta(\mathrm{G}(\mathcal{N}, \mathcal{E}), \{\texttt{dep1}, \ldots, \texttt{depn}\})$$

is equivalent to,

$$\bigsqcup\{\delta(\mathrm{G}(\mathcal{N}, \mathcal{E}), \{\texttt{dep1}\}), \ldots, \delta(\mathrm{G}(\mathcal{N}, \mathcal{E}), \{\texttt{depn}\})\}$$

Therefore a δ-slicing operation consisting of more than one dependency can be performed in terms of relational operations by combining the δ-slicing operations for a single dependency given above with the implementation of distributed graph union in subsection 11.3.2.

11.3.5 $\alpha\beta$-Slicing

An $\alpha\beta$-slicing operation,

$$_{\alpha}\|_{\beta}(\mathrm{G}(\mathcal{N}, \mathcal{E}), \mathcal{C})$$

employs node-based slicing criteria for extracting a subgraph from the given graph. The argument $\mathcal{C}$ and the α and β constraints will be considered separately.

The Argument $\mathcal{C}$

The argument $\mathcal{C}$, lists the nodes that can be a start-node or stop-node in the resulting graph. If one of the named nodes is isolated in the given graph, then it also appears in the resulting graph. The δ-slicing operation,

$$\delta(\mathrm{G}(\mathcal{N}, \mathcal{E}), \{\$\texttt{ISOLATED}\$\})$$

can be used to find the isolated nodes in a graph, and the technique for implementing this operation is given in the previous subsection on page 255. The relation representing the subgraph of isolated nodes will be called `isolated-nodes`.

Let `start-nodes` be the relation recording the set of nodes that can be a start-node in the resulting graph, and `stop-nodes` the relation recording the set of nodes that can be a stop-node in the resulting graph. The relation `ok-start-nodes`, which records the edges with a valid start-node can be obtained by,

$$\texttt{start-nodes} \bowtie_{\Theta} \texttt{graph-edges}$$

where Θ is the predicate,

$$\begin{aligned}
\texttt{start-nodes.ent-id} &= \texttt{graph-edges.ent-id1} \wedge \\
\texttt{start-nodes.ent-class} &= \texttt{graph-edges.ent-class1} \wedge \\
\texttt{start-nodes.mod-id} &= \texttt{graph-edges.mod-id1} \wedge \\
\texttt{start-nodes.region} &= \texttt{graph-edges.region1}
\end{aligned}$$

Similarly the relation `ok-stop-nodes`, which records the edges with a valid stop-

node can be obtained by,

$$\texttt{stop-nodes} \bowtie_{\Theta} \texttt{graph-edges}$$

where Θ is the predicate,

$$\begin{aligned}
\texttt{stop-nodes.ent-id} &= \texttt{graph-edges.ent-id2} \wedge \\
\texttt{stop-nodes.ent-class} &= \texttt{graph-edges.ent-class2} \wedge \\
\texttt{stop-nodes.mod-id} &= \texttt{graph-edges.mod-id2} \wedge \\
\texttt{stop-nodes.region} &= \texttt{graph-edges.region2}
\end{aligned}$$

The relation **subgraph-edges**, which records the edges that are to appear in the subgraph can be obtained by the operation,

$$\Pi_{\Psi}(\texttt{ok-start-nodes} \cup \texttt{ok-stop-nodes})$$

where Ψ is the list of attributes,

ent-id1, ent-class1, mod-id, region#1,
ent-id2, ent-class2, mod-id, region#2,
dependency

The set of nodes that can appear in the resulting graph can be obtained by finding nodes that are a start-node or a stop-node for the edges in **subgraph-edges** and then combining this set of nodes with the set of valid isolated nodes. The combined set of start-nodes and stop-nodes from **subgraph-edges** (called **end-nodes**

say) can be obtained by,

$$\Pi_{\texttt{ent-id1,ent-class1,mod-id1,region\#1}}(\texttt{subgraph-edges}) \cup$$
$$\Pi_{\texttt{ent-id2,ent-class2,mod-id2,region\#2}}(\texttt{subgraph-edges}) \qquad (11.2)$$

and the set of valid isolated nodes (called `valid-isolated`) can be obtained by

$$(\texttt{start-nodes} \bowtie \texttt{isolated-nodes}) \cup$$
$$(\texttt{stop-nodes} \bowtie \texttt{isolated-nodes})$$

The relation `subgraph-nodes` is then the result of

$$\texttt{end-nodes} \cup \texttt{valid-isolated}$$

The α and β Constraints

The α and β constraints slice a graph with respect to the label of a node. Consider the following $\alpha\beta$-slicing operation of the form,

$$\|_{\xi}^{\texttt{class=}\textit{class-name}}(\mathrm{G}(\mathcal{N}, \mathcal{E}), \langle \xi, \xi \rangle)$$

In order to find the edges that satisfy this constraint and hence appear in the resulting graph, the following select operation can be used,

$$\sigma_{\texttt{graph-edges.ent-class1=}\textit{class-name}}(\texttt{graph-edges}) \qquad (11.3)$$

If the relation resulting from this select operation is called `subgraph-edges`, then

the operation (11.2) can be used to obtain the relation end-nodes which contain the nodes that are either a start-node or stop-node for an edge in the subgraph.

To find the isolated nodes that satisfy a given α constraint, the following select operation can be used,

$$\sigma_{\texttt{isolated-nodes.ent-class=}\textit{class-name}}(\texttt{isolated-nodes})$$

The relation that results from this operation will be called `valid-isolated`. The set of nodes in the resulting graph is therefore the result of,

$$\texttt{end-nodes} \cup \texttt{valid-isolated}$$

For the other α constraints, the same procedure can be followed, but different predicates given to the select operations.

Consider now the $\alpha\beta$-slicing operation,

$$\xi \| _{\texttt{class=}\textit{class-name}}(\mathrm{G}(\mathcal{N}, \mathcal{E}), \langle \xi, \xi \rangle)$$

The procedure to obtain the subgraph of $\mathrm{G}(\mathcal{N}, \mathcal{E})$ that satisfies this slicing operation is the same as for the α constraint, but with different attributes named in the predicate of the select operation (11.3). In operation (11.3), the predicate refers to the start-node of an edge. The new predicate should be,

$$\texttt{graph-nodes.ent-class2} = \textit{class-name}$$

which refers to the stop-node of an edge. The other β constraints can be handled in a similar way.

On page 97 the $\alpha\beta$-slicing operation is defined as,

$$_{\alpha}\|_{\beta}(\mathrm{G}(\mathcal{N}, \mathcal{E}), \mathcal{C}) \equiv \mathit{remove\text{-}false\text{-}isolated\text{-}nodes}(\mathrm{G}(\mathcal{N}, \mathcal{E}), \mathrm{G}_i(\mathcal{N}_i, \mathcal{E}_i))$$

where $\mathrm{G}_i(\mathcal{N}_i, \mathcal{E}_i)$ is the graph resulting from,

$$\bigsqcap\{_{\xi}\|_{\xi}(\mathrm{G}(\mathcal{N}, \mathcal{E}), \mathcal{C}),\ _{\alpha}\|_{\xi}(\mathrm{G}(\mathcal{N}, \mathcal{E}), \langle \xi, \xi \rangle),\ _{\xi}\|_{\beta}(\mathrm{G}(\mathcal{N}, \mathcal{E}), \langle \xi, \xi \rangle)\}$$

The graph $\mathrm{G}_i(\mathcal{N}_i, \mathcal{E}_i)$ can be obtained by following the procedures given above. The function *remove-false-isolated-nodes* removes the nodes that are isolated in $\mathrm{G}_i(\mathcal{N}_i, \mathcal{E}_i)$ but which are not isolated in $\mathrm{G}(\mathcal{N}, \mathcal{E})$. To perform this operation, derive the set of isolated nodes within $\mathrm{G}(\mathcal{N}, \mathcal{E})$. The procedure for doing this is given on page 255. If this relation is called `isolated-nodes` and the relation `graphi-nodes` represents the nodes that are isolated in $\mathrm{G}_i(\mathcal{N}_i, \mathcal{E}_i)$, then the set of false isolated nodes can be removed by the operation,

`isolated-nodes` $\cap$ `graphi-nodes`

Chapter 12

Conclusions

12.1 Have the Goals been Achieved?

The work described in this thesis has addressed itself to the maintenance phase of the software lifecycle, with particular emphasis placed on finding techniques for reading and understanding large programs. In order to reason about the code of a program, it is necessary to abstract away from the code so that only the relevant information is considered. To accomplish this the module-to-module, entity-to-module and entity-to-entity interconnection graphs are used. A set of graph operations are developed which help a maintenance programmer to manipulate these graphs.

The module-to-module interconnection graph can be used to analyse the structure of a system. Two common hierarchies that are used by different software

design strategies are the virtual machine and abstract-data type hierarchies.

For the module languages, the module-to-module interconnection graph can be analysed in terms of both of these hierarchies. By analysing the module-to-module interconnection graph, modules can be classified in terms of the role they are playing within a system. This allows a maintenance programmer to identify modules that may require a closer examination. Analysing the module-to-module interconnection graph can also help detect modules that are redundant within a system, because they are never used.

The entity-to-module interconnection graph provides a less abstract view of a program, revealing the client and supplier relationships between the modules. This graph can help detect inconsistent importing and exporting of entities. This in turn reveals an inconsistency in interpreting a design decision.

The entity-to-entity interconnection graph records information on the dependencies between the level 0 entities of a module. Combining the analysis of this graph with the analysis of the entity-to-module interconnection graph allows a module to be classified with respect to the taxonomies of modules given by Booch and Ross. Having classified modules according to Booch's taxonomy, potpourri modules can be identified.

Three techniques are given for breaking up a potpourri module into smaller modules which provide only a single service to the system. The first technique is "grouping by type-families", which is the process of detecting abstract data types that exist within a module. The second technique is "grouping by state variables", which is the process of detecting the abstract-state machines within a module. The last technique is "grouping by imports", which uses the different client views of a

module as the basis for breaking up a module.

The graph operations allow a maintenance programmer to combine two or more graphs so as to create a single graph, or to create a graph that is the subgraph common to two or more given graphs. Graph slicing operations are developed that extract a subgraph from a given graph that satisfies particular node and edge constraints. These graphs operations can be used to extract special forms of subgraphs. In particular, the three module factoring techniques given involve the use of these graph operations.

When working with a large system, it is impractical to regenerate the information needed for inter-module code analysis for every code analysis related activity. A relational database schema has therefore been designed which can record information on the inter-module connections within a system. Techniques for using this database schema to perform the graph operations are given.

12.2 Future Directions

The relational database schema has to be extended to cater for inheritance and instantiation. This will then allow the database to record all the different inter-module connections within a system written in one of the module languages. Furthermore, the existing database schema handles the use of a variables in a very simple way. The database schema should be modified so that the actions that are performed on a variable within a region are recorded, i.e., record if the variable is `defined` or `referenced`. Extending the database schema in this way will then allow inter-module code analysis to be performed with respect to *how* a state variable is

being used. This will then help in the detection of ripple effects across a module's boundary.

The work on inter-module code analysis can be extended to consider code based software metrics. With the current work it is possible to detect the different client views of a module, and it is possible to determine the module categories for each client view. The simplest form of module is one that provides only one client view, and that client view corresponds to only one module category. The most complex module would be one that has many client views, each of which corresponds to a potpourri module, i.e., it corresponds to several module categories. In order to determine the complexity of the module interconnections, it would be necessary to combine these relative complexity measures for modules, with complexity measures for the architectural structure of a system. This work should also help in assessing the reusability of a module.

Guidelines for determining how well an abstract-state machine is coded need to be developed. This will involve redefining the existing cohesion and coupling measures, so that they accommodate the module construct. This will then also help in developing software metrics on module interconnections.

The relational database schema was designed so that its use was not restricted to the inter-module code analysis work of this thesis. With some extensions, the database could be used by software tools such as interactive cross referencers and documentation generator. Developing the appropriate database front ends, and integrating them, could form the basis for a software maintenance support environment.

Appendix A

Glossary of Terminology

Architectural Structure	The structure of a system with respect to the unit of modularity. With the module languages this is the structure of the system with respect to the module construct.
Attribute	See **Attribute Type** and **Attribute Value.**
Attribute Type	This is the name that identifies a particular domain of values.
Attribute Value	An attribute value is an instance of a value from the domain denoted by the associated attribute type.

Bequeathing Module	A module that provides entities to another module via an inheritance mechanism.
Call Graph	A call graph is a directed graph that represents the dynamic relations between routines.
Candidate Key	A combination of attributes that can uniquely identify a tuple from a given relation.
Class	This is a data type module used to implement abstract data types.
Client Module	A module that imports an entity from another module.
Client View	The view of a module that is given to users of a module. This view normally consists of a lists of the modules public entities, together with enough information to know how to use them.
Code Analysis	The process of examining a program in order to gain some knowledge of its structure or execution behaviour.
Cohesion	A measure of the strength of functional association of processing activities.

Common Coupling	A type of coupling characterised by two or more routines referring to the same shared variable.
Concrete Module	A concrete module is a fully elaborated version of a generic module, where the entities of the concrete module can be used within the program.
Coupling	The degree of dependence of one routine upon another; specifically, a measure of the chance that a defect in one routine will appear as a defect in another, or the chance that a change to one routine will necessitate a change to the other.
Edge	An ordered pair of nodes denoting a dependency between the start node and the stop node.
Entity	An entity is anything that can be named or denoted in a program. Objects, types, values, modules are all entities.
Entity Attributes	Details about the characteristics of an entity, e.g., its type, the number of parameters, the type of the parameters, etc.
Functional Role	The functional role of a module is it's classification according to Booch's taxonomy [16, pages 228–9].
Fundamental Module	A module that plays a critical role in a system.

Generic Module	A generic module is a template module that describes the characteristics of its entities in a general way. The description of these entities is not complete enough to allow them to be used in the program.
Global Entity	Entities declared in the outermost block of a module.
Global Module	A module that is not contained within a block.
Heir Module	A module that aquires entities by means of an inheritance mechanism.
Imported Entity	An entity that is explicitly imported by a module.
Information Hiding	Implementation information is not revealed to client modules.
Inheritance	The process of creating a module as an extension or specialisation of another.
Inheritance Graph	A graph showing the modules that are created as extension or specialisations of each other. The dependency recorded in this graph is the `inherits-from`.
Injected Entity	An entity that is exported by a local module and implicitly imported by the module that encapsulates the local module.

Instantiation Tree	An instantiation tree is a tree structure that shows which modules are instantiations of other modules.
Interconnection Graph	An interconnection graph is a directed labelled graph that is used to represent the dependencies between entities in a program.
Local Module	A module declared within a block.
Metaclass	A class module whose instances are themselves class modules.
Module	A module is a named collection of entities, where the programmer has precise control over the entities that are imported from and exported to the surrounding environment.
Module Factoring	Module factoring is the process of taking a module that provides several disparate services, and creating smaller modules where each module provides only one service.
Module Languages	Languages that provide a module construct as part of the language.

Multiple Inheritance	A form of inheritance where a module can be created as an extension or specialisation of several modules.
Node	The *objects* that comprise a tree or graph.
Object	An object has a set of operations and a state that remembers the effect of the operations.
Overriding	Overriding is the redeclaration of an inherited entity.
Potpourri Module	A module conforming to several of the classifications given by Booch [16, pages 228–9].
Primary Key	The set of attributes chosen by the database designer as the principle means of identifying tuples within a relation.
Private Variable	A variable that is local to module, but which is considered non-local by the routines of that module.
Program Comprehension	This is the process of understanding what a program does with respect to the real world, or with respect to a particular problem domain.
Public Entities	Entities exported by a module.

Relation	A named set of tuples.
Relation Scheme	A description of the structure of a relation.
Responsibility Assignment	The service which a module is charged with providing to a system.
Ripple Effect	The manifestation of a defect in one part of a system as a defect in other parts of the system; the effect of a change in one part of a system causing defects in other parts of the system and/or necessitating further changes to other parts of the system.
Root Module	A module that appears to represent a system, or part of the system.
Routine	A subprogram unit that could be either a procedure or function.
Routine Group	A group of routines that are dependent because they invoke each other.
Scope	The scope of a declaration is the region of text over which the declaration has an effect.

Shared Variable	A variable that is declared as a global variable in a module and whose scope of visibility extends over several modules.
Solitary Module	A module that is independant of the other modules in a system with respect to the uses dependency.
Spanning Tree	The spanning tree is a graph with all backward edges removed.
Specialised Module	A module that provides a specialised services within a system.
Start-node	The node that marks the starting point for an edge.
State Information	Data that is stored in a state variable.
State Variable	A variable that is declared at level 0 in a module.
Stop-node	The node that marks the terminating point for an edge.
Subclassing	A form of inheritance where a module can be constructed as an extension or specialisation of only one module.
Supplier Module	A module that provides an entity to other modules.

Supplier View — The view of a module that the implementor has. It contains all the information given to the client modules, together with knowledge about how a module implements the service it is providing.

Terminal Module — A low level module within a system that requires non of the facilities offered by the other modules in the system.

Tuple — A set of related attribute values.

Type-Family — A group of type entities that are dependent on each other.

Virtual Machine — A virtual machine ia a software extension to the underlying hardware, and also possibly to other virtual machines.

Visible — At a given point in a program text, the declaration of an entity with a certain identifier is said to be visible if, the entity is an acceptable meaning for an occurrence at that point of the identifier.

Appendix B

Glossary of Notation

$\mathbb{G}_1(\mathcal{N}_1, \mathcal{E}_1) \sqsubseteq \mathbb{G}_2(\mathcal{N}_2, \mathcal{E}_2)$	Graph containment. $G_1(\mathcal{N}_1, \mathcal{E}_1)$ is a subgraph of $G_2(\mathcal{N}_2, \mathcal{E}_2)$.
$\mathbb{G}_1(\mathcal{N}_1, \mathcal{E}_1) \sqsubset \mathbb{G}_2(\mathcal{N}_2, \mathcal{E}_2)$	Strict graph containment.
$\mathbb{G}_1(\mathcal{N}_1, \mathcal{E}_1) \sqsupseteq \mathbb{G}_2(\mathcal{N}_2, \mathcal{E}_2)$	Graph containment. $G_1(\mathcal{N}_1, \mathcal{E}_1)$ is a supergraph of $G_2(\mathcal{N}_2, \mathcal{E}_2)$.
$\mathbb{G}_1(\mathcal{N}_1, \mathcal{E}_1) \sqsupset \mathbb{G}_2(\mathcal{N}_2, \mathcal{E}_2)$	Strict graph containment.
$\mathbb{G}_1(\mathcal{N}_1, \mathcal{E}_1) \sqcup \mathbb{G}_2(\mathcal{N}_2, \mathcal{E}_2)$	Simple graph union.
$\bigsqcup\{\mathbb{G}_1(\mathcal{N}_1, \mathcal{E}_1) \ldots \mathbb{G}_n(\mathcal{N}_n, \mathcal{E}_n)\}$	Distributed graph union.

$\mathbb{G}_1(\mathcal{N}_1, \mathcal{E}_1) \sqcap \mathbb{G}_2(\mathcal{N}_2, \mathcal{E}_2)$	Strict simple graph intersection.
$\mathbb{G}_1(\mathcal{N}_1, \mathcal{E}_1) \sqcap_+ \mathbb{G}_2(\mathcal{N}_2, \mathcal{E}_2)$	Full simple graph intersection.
$\bigsqcap\{\mathbb{G}_1(\mathcal{N}_1, \mathcal{E}_1) \ldots \mathbb{G}_n(\mathcal{N}_n, \mathcal{E}_n)\}$	Distributed strict graph intersection.
$\bigsqcap_+\{\mathbb{G}_1(\mathcal{N}_1, \mathcal{E}_1) \ldots \mathbb{G}_n(\mathcal{N}_n, \mathcal{E}_n)\}$	Distributed full graph intersection.
$\delta(\mathbb{G}(\mathcal{N}, \mathcal{E}), \mathcal{C})$	δ-slicing.
${}_\alpha\|_\beta(\mathbb{G}(\mathcal{N}, \mathcal{E}), \mathcal{C})$	$\alpha\beta$-slicing.
ξ	A symbol used to denote that no restriction is being employed.

Appendix C

Abstract Syntax

Program :: *s-program* : set of *Global-Module*

Global-Module :: *s-global-module* : *Module-Entity*
where

inv-Global-Module(*mk-Global-Module*(*gm*)) $\triangleq$ *region#*(*gm*) = *0*

Module-Entity :: *s-exports* : *Exported-Entities*
s-imports : *Imported-Entities*
s-region# : $\mathbb{N}$
s-region : *Region*

Exported-Entities :: *s-exported-entity* : map *Entity* to *Module-Set*

Imported-Entities :: *s-imported-entity* : map *Module-Id* to *Entity-Set*

Module-Set = set of *Module_Id*

Entity-Set = set of *Entity*

Entity :: *s-entity-id* : *Entity-Id*

Entity-Id = *Constant-Id* | *Type-Id* | *Variable-Id* | *Routine-Id* | *Module-Id*

Region :: *s-constant* : map *Constant-Id* to *Constant-Set*
s-type : map *Type-Id* to *Type-Set*
s-variable : map *Variable-Id* to *Variable-Set*
s-routine : map *Routine-Id* to *Routine-Set*
s-module : map *Module-Id* to *Module-Set*
s-body : *Body*

Constant-Set = set of *Constant-Entity*

Type-Set = set of *Type-Entity*

Variable-Set = set of *Variable-Entity*

Routine-Set = set of *Routine-Entity*

Module-Set = set of *Module-Entity*

Constant-Entity :: *s-type* : *Type-Constructor*

Type-Entity :: *s-type* : *Type-Constructor*

Variable-Entity :: *s-type* : *Type-Constructor*

Routine-Entity :: *s-formal-parameters* : *Parameters*
s-result-type : *Type-Constructor*
s-region# : $\mathbb{N}$
s-region : *Region*

Parameters = map *Parameter-Id* into set of *Type-Constructor*

Type-Constructor = *Qualident* | *Enumerated* | *Subrange* | *Array* | *Record* | *Set* | *Pointer* | *Routine*

Qualident = *Type-Id*

Enumerated = set of *Enum-Id*

Body :: *s-constants-used* : set of *Constant-Id*
s-types-used : set of *Type-Id*
s-variables-used : set of *Variable-Id*
s-routine-used : set of *Routine-Id*

System-Id = ...
Module-Id = ...
Constant-Id = ...
Type-Id = ...
Variable-Id = ...
Routine-Id = ...
Enum-Id = ...

} Some appropriate set of entity identifiers

Appendix D

The Database Relations

1. The relation *program-components* is used to record which external modules a named system consists of. The relation *program-components* has two attributes, and they both form the primary key.

   sys-id — This is the key attribute. It is used to denote the name of the system to which the modules belong.

   mod-id — This attribute is the name of a module that belongs to the named system.

   program-components

sys-id	*mod-id*

2. The relation *exports* is used to record which entities are exported by a named module. The relation *exports* has three attributes all of which combine to form the primary key.

mod-id — This is the first key attribute. It denotes the name of the module that exports the entity.

ent-id — This is the second key attribute. This attribute denotes the name of an entity exported by the named module.

imp-mod-id — This is the third key attribute. This attribute denotes the name of the module that can import the named entity, if any external module can import the entity then the special value 'All' is used.

exports

mod-id	*ent-id*	*imp-mod-id*

3. The relation *imports* records information on a named entity exported by a named module and imported into another named module. The *imports* relation has five attributes, four of which are key attributes.

mod-id — This is the first key attribute. It denotes the name of the importing module.

exp-mod-id — This is the second key attribute. It denotes the name of the exporting module.

ent-id — This is the third key attribute. It denotes the name of an entity exported by the module named in the exp-mod-id attribute field and imported by the module named in the dest-mod-id attribute field.

imports

mod-id	*exp-mod-id*	*ent-id*

In languages like CLU and Eiffel, whose module does not have separate definition and implementation parts, the imported entities are always classed as

being visible in both the definition and implementation parts.

4. The relation *constant-dec* records information on the type of a named constant, declared in a named module. The relation *constant-dec* has five attribute three of which form the primary key.

   mod-id — This is the first key attribute. Its value is the name of a module

   region# — This is the second key attribute. This is the region number in which the constant is declared. Region number zero is the outermost level and is associated with the scope of the declaring module.

   ent-id — This is the third key attribute. Its value is the name of a constant declared in the module named in the mod-id attribute.

   type# — A "type" is either a qualified identifier or a type-constructor. A unique identifier called the "type#" is allocated for each qualified identifier that is used — the same one is re-used for each occurrence. A unique identifier is allocated for each occurrence of a type-constructor.

constant-dec

mod-id	*region#*	*ent-id*	type#

5. The relation *type-dec* records information on the type of a named type declared in a named module. The relation *type-dec* has four attribute three of which form the primary key attributes.

mod-id — This is the first key attribute. Its value is the name of a module

region# — This is the second key attribute. This is the region in which the type is declared.

ent-id — This is the third key attribute. Its value is the name of a type declared in the module named in the mod-id attribute.

type# — A "type" is either a qualified identifier or a type-constructor. A unique identifier called the "type#" is allocated for each qualified identifier that is used — the same one is re-used for each occurrence. A unique identifier is allocated for each occurrence of a type-constructor.

type-dec

mod-id	*region#*	*ent-id*	type#

6. The relation *variable-dec* records information on the type of a named variable declared in a named module. The relation *variable-dec* has four attributes, three of which form the primary key attributes.

mod-id — This is the first key attribute. Its value is the name of a module

region# — This is the second key attribute. This is the region in which the variable is declared.

ent-id — This is the third key attribute. Its value is the name of a variable declared in the module named in the mod-id attribute.

type# — A "type" is either a qualified identifier or a type-constructor. A unique identifier called the "type#" is allocated for each qualified identifier that is used — the same one is re-used for each occurrence. A unique identifier is allocated for each occurrence of a type-constructor.

variable-dec

mod-id	*region#*	*ent-id*	type#

7. The relation *type-of-routine* records information on the type of a routine. The relation *type-of-routine* has four attributes, three of which form the primary key relations.

mod-id — This is the first key attribute. Its value is the name of a module

region# — This is the second key attribute. This is the region in which the routine is declared.

ent-id — This is the third key attribute. Its value is the name of a routine declared in the module named in the mod-id attribute.

r_type# — This is the attribute that records the result type of the routine. If the routine is a procedure the result type is the special value '$Void$.

type-of-routine

mod-id	*region#*	*ent-id*	r_type#

8. The relation *region-of-routine* records information on the region number associated with a routine. The relation *region-of-routine* has four attributes, three of which form the primary key relations.

mod-id — This is the first key attribute. Its value is the name of a module

region# — This is the second key attribute. This is the region in which the routine is declared.

ent-id — This is the third key attribute. Its value is the name of a routine declared in the module named in the mod-id attribute.

c-region# — This is the region that belongs to the routine identified by the primary key.

region-of-routine

mod-id	*region#*	*ent-id*	c-region#

9. The relation *para-id-rel* records information about the type and class of a named formal parameter from a named particular list. The relation *para-id-rel* has five attributes, four of them being key attribute.

mod-id — This the first key attribute. It denotes the name of the module in which the symbol recorded in the `para-list-sym` attribute exist.

region# — This is the second key attribute. This is the region in which the routine is declared.

ent-id — This is the third key attribute. Its value is the name of an routine declared in the module named in the mod-id attribute.

para-id — This is the fourth key attribute, it identifies a particular formal parameter.

p_type# — A "type" is either a qualified identifier or a type-constructor. A unique identifier called the "type#" is allocated for each qualified identifier that is used — the same one is re-used for each occurrence. A unique identifier is allocated for each occurrence of a type-constructor.

para-id-rel

mod-id	*region#*	*ent-id*	*para-id*	p_type#

10. The relation *local-module* is used to record the region number associated with a particular local module. The relation *local-module* has four attributes, three of which are key attributes.

mod-id — This is the first key attribute. Its value is the name of an external module

region# — This is the second key attribute. This is the region in which the local module is declared.

loc-mod-id — This is the third key attribute. Its value is the name of a local module declared in the external module named in the mod-id attribute.

c-region# — This is the region that belongs to the local module identified by the primary key.

local-module

mod-id	*region#*	*loc-mod-id*	*c-region#*

11. The relation *local-module-exports* is used to record the name of the entities exported by a named local module. The relation has four attributes, all of which combine to form the primary key.

mod-id — This is the first key attribute. Its value is the name of an external module

region# — This is the second key attribute. This is the region in which the local module is declared.

loc-mod-id — This is the third key attribute. Its value is the name of a local module declared in the external module named in the mod-id attribute.

ent-id — This is the fourth key attribute. Its value, is the identifier of an entity exported by the named local module.

local-module-exports

mod-id	*region#*	*loc-mod-id*	*ent-id*

12. The relation *local-module-imports* is used to record information on a named entity imported by the named local module.

    mod-id — This is the first key attribute. Its value is the name of an external module

    region# — This is the second key attribute. This is the region in which the local module is declared.

    loc-mod-id — This is the third key attribute. Its value is the name of a local module declared in the external module named in the mod-id attribute.

    ent-id — This is the fourth key attribute. Its value, is the identifier of an entity imported by the named local module.

local-module-exports

mod-id	*region#*	*loc-mod-id*	*ent-id*

13. The relation *type* records information on the class of each of the types in a module. The term "type" is used to apply to all types anonymous types in variable and routine declarations, as well as type declarations. This relation is used to determine which of the different type relations contains the description of the named type declaration. The relation *type* has three attributes, two of which are key attributes.

mod-id — This is the first key field. It denotes the name of the module in which the type elaboration exists.

type# — This is the second key field. It denotes the type declaration whose class is being sought.

type-class — This attribute records the class of a named type elaboration. Its values can be one of the following: "QUALIDENT", "ENUMERATED", "SUBRANGE", "ARRAY", "RECORD", "SET", "POINTER" or "ROUTINE".

type

mod-id	*type#*	type-class

14. The relation *qualident* records the identifier of a type entity. The value can be either a predefined type like INTEGER or CARDINAL, the name of a type entity declared in the named module, or a type module imported into the named module. The relation *qualident* has three attributes, two of which are key attributes.

mod-id — This is the first key field. It denotes the name of the module in which the type elaboration exists.

type# — This is the second key field. It denotes the type declaration whose identifier is being sought.

ent-id — This attribute records the identifier of the type named in the type# attribute.

qualident

mod-id	*type#*	ent-id

15. The relation *enumerated-type* record the names of each of the elements of a named enumerated type. The relation *enumerated-type* has three attributes,

two of which are key attributes.

mod-id — This is the first key field. It denotes the name of the module in which the type elaboration exists.

type# — This is the second key field. It denotes the enumerated type whose element identifiers are being sought.

ent-id — This attribute records the identifier of each of the elements of a named enumerated type.

enumerated-type

mod-id	*type#*	ent-id

16. The relation *subrange-type* records information on the range type of a subrange type. The relation *subrange-type* has five attributes, two of them are key attributes.

mod-id — This is the first key field. It denotes the name of the module in which the type elaboration exists.

type# — This is the second key field. It denotes the type declaration whose subrange elaboration is being sought.

range-type# — This attribute records the type declaration number of the base type, i.e. the type of the subrange elements.

subrange-type

mod-id	*type#*	range-type#

17. The relation *subrange-delimiters* records information on the range type of a subrange type. The relation *subrange-type* has five attributes, two of them are key attributes.

mod-id — This is the first key field. It denotes the name of the module in which the subrange type elaboration exists.

type# — This is the second key field. It denotes the type declaration number of the subrange for which the delimiters are being sought.

ent-id — This attribute records identifier of the routine or constant used to delimit one of the bounds of the subrange.

subrange-delimiters

mod-id	*type#*	ent-id

18. The relation *array-type* records information on a named array type. The relation *array-type* has four attributes, three of them being key attributes.

mod-id — This is the first key field. It denotes the name of the module in which the type elaboration exists.

type# — This is the second key field. It denotes the type declaration whose array elaboration is being sought.

indices-list-sym — This attribute records a value that is used as a key attribute with the *array-indices* relation. It is used to find the type of each of the indices of a given array.

elem-type# — This attribute records the type declaration number for the array elements.

array-type

mod-id	*type#*	elem-type#

19. The relation *array-indices* record the type declaration number for each array index in a named array indices list.

mod-id — This is the first key attribute. Its value is the name of a module

indices-list-sym — This is the second key attribute. Its value is the symbol that denotes an array index list.

index-type# — This attribute records the type declaration number for an array index.

array-indices

mod-id	*indices-list-sym*	elem-type#

20. The relation *record-field-type* records information about the type of a named field in a named record. The relation *record-field-type* has four attributes, three of which are key attributes.

mod-id — This is the first key field. It denotes the name of the module in which the type elaboration exists.

type# — This is the second key field. It denotes the record type whose field identifiers are being sought.

field-id — This is the third key field. It denotes the name of a field in a named record.

field-type# — This attribute records the type declaration number of the named record field.

record-field-type

mod-id	*type#*	*field-id*	field-type#

21. The relation *set-type* records information on the type of the elements of a named set. The relation *set-type* has three attributes, two of them being key attributes.

mod-id — This is the first key field. It denotes the name of the module in which the type elaboration exists.

type# — This is the second key field. It denotes the enumerated type whose element identifiers are being sought.

base-type# — This attribute records the type declaration number of the base type, i.e. the type of the set elements.

set-type

mod-id	*type#*	base-type#

22. The relation *pointer-type* records information on the type that is bound by the pointer. The relation *pointer-type* has three attributes, two of them being key attributes.

mod-id — This is the first key field. It denotes the name of the module in which the type elaboration exists.

type# — This is the second key field. It denotes the type declaration whose pointer elaboration is being sought.

bound-type# — This attribute records the type declaration number of the base type, i.e. the type of the referenced elements.

pointer-type

mod-id	*type#*	bound-type#

23. The relation *proc-type* records information on a named routine type. The

relation *proc-type* has four attributes, two of them being key attributes.

mod-id — This is the first key field. It denotes the name of the module in which the type elaboration exists.

type# — This is the second key field. It denotes the routine type to which the information pertains.

para-sym — This attribute denotes the value that is used as the key attribute for the relations *proc-type-para-list* and *proc-type-pos-type*

proc-type# — This attribute records the type declaration number of the routine type. The special value, '$Void$', is used when the routine has no type, i.e. it is a PROCEDURE.

proc-type

mod-id	*type#*	*para-sym*	proc-type#

24. The relation *proc-type-para-type* records information on the type and class of a named parameter, denoted by its position within the parameter list.

mod-id — This is the first key attribute. Its value is the name of a module

para-sym — This is the first key attribute. It identifies a parameter list.

para-type# — This attribute denotes the type declaration number that describes the formal parameter's type

proc-type-para-type

mod-id	*para-sym*	para-type#

25. The relation *constants-used* is used to record the global constants used by a global routine. The relation *constants-used* has three attributes, and they are

all used to form a primary key.

mod-id — This is the first key attribute. Its value is the name of the external module.

routine-id — This is the second key attribute. Its value is the name of the name of the routine that uses the named constant.

constant-id — This is the third key attribute. Its value is the name of the name of the constant that is used.

constants-used

mod-id	*routine-id*	*constant-id*

26. The relation *types-used* is used to record the global types used by a global routine. The relation *types-used* has three attributes, and they are all used to form a primary key.

mod-id — This is the first key attribute. Its value is the name of the external module.

routine-id — This is the second key attribute. Its value is the name of the name of the routine that uses the named type.

type-id — This is the third key attribute. Its value is the name of the name of the type that is used.

types-used

mod-id	*routine-id*	*type-id*

27. The relation *variables-used* is used to record the global variables used by a global routine. The relation *variables-used* has three attributes, and they are all used to form a primary key.

mod-id — This is the first key attribute. Its value is the name of the external module.

routine-id — This is the second key attribute. Its value is the name of the name of the routine that uses the named variable.

variable-id — This is the third key attribute. Its value is the name of the name of the variable that is used.

variables-used

mod-id	*routine-id*	*variables-id*

28. The relation *routines-used* is used to record the global routines called by another global routine. The relation *routines-called* has three attributes, and they are all used to form a primary key.

mod-id — This is the first key attribute. Its value is the name of the external module.

calling-routine-id — This is the second key attribute. Its value is the name of the name of the routine that calls the named routine.

called-routine-id — This is the third key attribute. Its value is the name of the name of the routine that is called.

routines-used

mod-id	*routine-id*	*variables-id*

Bibliography

[1] *Reference Manual for the Ada Programming Language.* 1983. ANSI/MIL-STD 1815A.

[2] *Working Draft: Programming Language Extended Pascal.* October 1986. ISO/TC97/SC22/WG2 N100.

[3] *Standard FORTRAN Programming Manual.* Manchester, second edition ed., 1972.

[4] *American National Standard Programming Language PL/1.* New York, 1976. ANSI X3.53-1976.

[5] *Data Processing — Programming Languages — SIMULA.* May 1987. Svensk Standard SS 63 61 14.

[6] Abbott, R.J., "Program Design by Informal English Description," *Communications of the ACM*, vol. 26, pp. 882–894, November 1983.

[7] Aho, A.V., Sethi, R., and Ullman, J.D., *Compilers Principles, Techniques and Tools.* Reading, Massachusetts: Addison-Wesley Publishing Company, 1986.

[8] Ambras, J. and O'Day, V., "MicroScope: A Program Analysis System," in *Proceedings of the Twentieth Hawaii International Conference on System Sci-*

ences 1987, (Shriver, B.D., ed.), (California), pp. 71–81, Western Periodicals Company, January 1987.

[9] Ashcroft, E. and Manna, Z., "The Translation of 'GOTO' Programs to 'WHILE' Programs," in *Proceedings of IFIP Congress 71*, pp. 250–255, North-Holland Publishing Co., 1972.

[10] Baker, B.B., "An Algorithm for Structuring Flowgraphs," *Journal of the ACM*, vol. 24, pp. 98–120, January 1977.

[11] Bartussek, W. and Parnas, D.L., "Using Assertions about Traces to Write Abstract Specifications for Software Modules," in *Proceedings of Second Conference of European Cooperation in Informatics*, pp. 211–236, Sprnger-Verlag, 1978. Lecture Notes in Computer Science, 65. Also in "Software Specification Techniques".

[12] Bergeretti, J.F. and Carré, B.A., "Information-Flow and Data-Flow Analysis of WHILE-Programs," *ACM Transactions on Programming Languages and Systems*, vol. 7, pp. 37–61, January 1985.

[13] Bobrow, D.G., DeMichiel, L.G., Gabriel, R.P., Keene, S.E., Kiczales, G., and Moon, D.A., *Common Lisp Object System Specification X3JI3 Document 88–002R.* 1988. Appeared as a special issue of Sigplan Notices, vol 23, September 1988.

[14] Böhm, C. and Jacopini, G., "Flow Diagrams, Turing Machines and Languages with only Two Formation Rules," *Communications of the ACM*, vol. 9, pp. 366–371, May 1966.

[15] Booch, G., "Object-Oriented Development," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 2, pp. 211–221, 1986.

[16] Booch, G., *Software Engineering with Ada.* Reading, Massachusetts: Benjamin/Cummings Publishing Company Inc., second ed., 1987.

[17] Britton, K.H. and Parnas, D.L., "A-7E Software Module Guide," Tech. Rep., Naval Research Laboratory, Washington, D.C., December 1981. NRL Memorandum Report 4702.

[18] Buhr, R.J.A., *System Design with Ada.* Englewood Cliffs, NJ.: Prentice-Hall International, 1984.

[19] Burstall, R.M. and Darlington, J.A., "A Transformational System for Developing Recursive Procedures," *Journal of the ACM*, vol. 24, pp. 44–67, January 1977.

[20] Calliss, F.W., "Problems with Automatic Restructurers," *SIGPLAN Notices*, vol. 23, pp. 13–21, March 1988.

[21] Calliss, F.W. and Cornelius, B.J., "Dynamic Data Flow Analysis of C Programs," in *Proceedings of the Twenty-First Annual Hawaii International Conference on System Sciences*, (Washington, D.C.), pp. 518–523, IEEE Computer Society Press, January 1988.

[22] Calliss, F.W. and Cornelius, B.J., "Two Module Factoring Techniques," *The Journal of Software Maintenance — Research and Practice*, 1989.

[23] Calliss, F.W., Khalil, M.M., Munro, M., and Ward, M., "Reengineering vs Restructuring," *Software Maintenance News*, vol. 5, p. 8, October 1987.

[24] Calliss, F.W., Khalil, M.M., Munro, M., and Ward, M., "A Knowledge-Based System for Software Maintenance," in *Proceedings of the Conference on Software Maintenance — 1988*, pp. 319–324, 1988.

[25] Cardelli, L., Donahue, J., Glassman, L., Jordan, M., Kalsow, B., and Nelson, G., "Modula-3 Report," Tech. Rep., Olivetti Research Center, Menlo Park, California, 1988.

[26] Cardelli, L. and Wegner, P., "On Understanding Types, Data Abstraction, and Polymorphism," *Computing Surveys*, vol. 17, pp. 471–522, December 1985.

[27] Carré, B.A., "Validation Techniques," in *Software Engineering for Microprocessor Systems*, (Depledge, P., ed.), pp. 173–197, London: Peter Peregrinus Ltd., 1984.

[28] Chen, T.Y. and Poole, P.C., "Dynamic Dataflow Analysis," *Information and Software Technology*, vol. 30, pp. 497–505, October 1988.

[29] Clements, P.C., "Function Specifications for the A-7E Function Driver Module," Tech. Rep., Naval Research Laboratory, Washington, D.C., November 1981. NRL Memorandum Report 4658.

[30] Clements, P.C., "Interface Specifications for the A-7E Shared Services Module," Tech. Rep., Naval Research Laboratory, Washington, D.C., September 1982. NRL Memorandum Report 4863.

[31] Clements, P.C., Faulk, S.R., and Parnas, D.L., "Interface Specification for the SCR (A-7E) Application Data Types Module," Tech. Rep., Naval Research Laboratory, Washington, D.C., August 1983. Technical Report NRL Report 8734.

[32] Clements, P.C., Parker, R.A., Parnas, D.L., Shore, J., and Britton, K.H., "A Standard Organization for Specifying Abstract Interfaces," Tech. Rep., Naval Research Laboratory, Washington, D.C., June 1984. Technical Report NRL Report 8815.

[33] Cleveland, L., "An Environment for Understanding Programs," in *Proceedings of the Twenty-First Annual Hawaii International Conference on System Sciences*, (Shriver, B.D., ed.), pp. 500–509, January 1988.

[34] Clocksin, W.F. and Mellish, C.S., *Programming in Prolog.* Berlin: Springer-Verlag, second edition ed., 1984.

[35] Codd, E.F., "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM*, vol. 13, pp. 377–387, June 1970.

[36] Cooper, S.D., "Pascal Program Call Graph Generator," Tech. Rep., School of Engineering and Applied Science(Computer Science), University of Durham, 1987. Final Year Project.

[37] Cooprider, L.W., *The Representation of Families of Software Systems.* PhD thesis, Carnegie-Mellon University, Computer Science Department, April 1979.

[38] Cornelius, B.J., "Problems with the Language Modula-2," *Software — Practice and Experience*, vol. 18, pp. 529–543, June 1988.

[39] Cornelius, B.J. and Kirby, G.H., "Depth of Recursion and the Ackermann Function," *Bit*, vol. 15, pp. 144–150, 1975.

[40] Cowell, D.F., Gillies, D.F., and Kaposi, A.A., "Synthesis and Structural Analysis of Abstract Programs," *The Computer Journal*, vol. 23, pp. 243–247, August 1980.

[41] Cox, B.J., *Object Oriented Programming — An Evolutionary Approach.* Reading, Massachusetts: Addison-Wesley, 1986.

[42] Cunningham, W. and Beck, K., "A Diagram for Object Oriented Programs," in *OOPSLA '86 Conference Proceedings*, pp. 361–367, ACM, 1986. Appeared as *Sigplan Notices*, vol. 21, no.11.

[43] Dahl, O.-J. and Hoare, C.A.R., "Hierarchical Program Structures," in *Structured Programming*, London: Academic Press Inc., 1972.

[44] Danforth, S. and Tomlinson, C., "Type Theories and Object-Oriented Programming," *ACM Computing Surveys*, vol. 20, pp. 29–72, March 1988.

[45] Debnath, N.C. and Bieman, J.M., "A Representation and Analysis of Interprocedural Structure," in *Proceedings of the Twentieth Hawaii International Conference on System Sciences 1987*, (Shriver, B.D., ed.), (California), pp. 92–100, Western Periodicals Company, January 1987.

[46] DeMarco, T., *Structured Analysis and System Specification.* New York: Yourdon Inc., 1978.

[47] DeRemer, F. and Kron, H.H., "Programming-in-the-Large versus Programming-in-the-Small," *IEEE Transactions on Software Engineering*, vol. SE-2, pp. 80–86, June 1976.

[48] Dijkstra, E., "GOTO Statements Considered Harmful," *Communications of the ACM*, vol. 11, pp. 147–148, 1968.

[49] Dijkstra, E.W., "The Structure of the "THE"-Multiprogramming System," *Communications of the ACM*, vol. 11, pp. 341–346, May 1968. Reprinted in Communications of the ACM, vol. 26, no. 1, July 1975, pp. 49-52.

[50] Dijkstra, E.W., "The Humble Programmer," *Communications of the ACM*, vol. 15, pp. 859–866, October 1972. ACM Turing Award.

[51] Dijkstra, E.W., "Notes on Structured Programming," in *Structured Programming*, London: Academic Press Inc., 1972.

[52] Dijkstra, E.W., "Guarded Commands, Nondeterminancy and Formal Derivation of Programs," *Communications of the ACM*, vol. 18, pp. 453–457, August 1975.

[53] Dijkstra, E.W., *A Discipline of Programming.* Englewood Cliffs, NJ.: Prentice-Hall, 1976.

[54] Embley, D.W. and Woodfield, S.N., "Cohesion and Coupling for Abstract Data Types," in *Sixth Annual International Phoenix Conference on Computer Communications*, (Washington, D.C.), pp. 229–234, IEEE Computer Society Press, February 1987.

[55] Embley, D.W. and Woodfield, S.N., "Assessing the Quality of Abstract Data Types Written in Ada," in *Proceedings: 10^{th} International Conference on Software Engineering*, (Washington, D.C.), pp. 144–153, IEEE Computer Society Press, March 1988.

[56] Fairfield, P. and Hennell, M.A., "Data Flow Analysis of Recursive Procedures," *SIGPLAN Notices*, vol. 23, pp. 48–57, January 1988.

[57] Fosdick, L.D. and Osterweil, L.J., "The Detection of Anomalous Interprocedural Data Flow," in *Proceedings — Second International Conference on Software Engineering*, (Washington, D.C.), pp. 624–628, IEEE Computer Society Press, October 1976.

[58] Fosdick, L.D. and Osterweil, L.J., "Data Flow Analysis in Software Reliability," *Computing Surveys*, vol. 8, pp. 305–330, September 1976.

[59] Gardner, M.R., "Successes and Limitations of Object-Oriented Design," *Journal of Pascal, Ada & Modula-2*, vol. 7, pp. 30–41, November-December 1988.

[60] Gehani, N., *Ada An Advanced Introduction Including Reference Manual for the Ada Programming Language.* New Jersey: Prentice-Hall Inc., 1984.

[61] Glagowski, T.G., "Using a Relational Query Language as a Software Maintenance Tool," in *Conference on Software Maintenance-1985*, (Washington, D.C.), pp. 211–220, IEEE Computer Society Press, November 1985.

[62] Goldberg, A. and Robson, D., *Smalltalk-80: The Language and its Implementation.* Reading, Massachusetts: Addison-Wesley Publishing Company, 1983.

[63] Green, T.R.G., "Conditional Program Statements and their Comprehensibility to Professional Programmers," *Journal of Occupational Psychology*, vol. 50, pp. 93–109, June 1977.

[64] Gries, D., "An Illustration of Current Ideas on the Derivation of Correctness Proofs and Correct Programs," *IEEE Transactions on Software Engineering*, vol. SE-2, pp. 238–244, December 1976.

[65] Gries, D., *The Science of Programming.* New York: Springer-Verlag, 1981.

[66] Hartmann, J. and Robson, D.J., "Approaches to Regression Testing," in *Proceedings Conference on Software Maintenance – 1988*, (Washington, D.C.), pp. 368–372, IEEE Computer Society Press, 1988.

[67] Hartmann, J. and Robson, D.J., "Revalidation During the Software Maintenance Phase," Tech. Rep., School of Engineering and Applied Science, University of Durham, 1989. Computer Science Technical Report TR 1/89.

[68] Heninger, K.L., "Specifying Software Requirements for Complex Systems: New Techniques and their Application," *IEEE Transaction on Software Engineering*, vol. SE-6, pp. 2–13, January 1980.

[69] Heninger, K.L., Kallander, J.W., Shore, J.E., and Parnas, D.L., "Software Requirements for the A-7E Aircraft," Tech. Rep., Naval Research Laboratory, Washington, D.C., November 1978. NRL Memorandum Report 3876.

[70] Hennell, M.A., Woodward, M.R., and Hedley, D., "On Program Analysis," *Information Processing Letters*, vol. 5, pp. 136–140, November 1976.

[71] Hoare, C.A.R., "An Axiomatic Basis for Computer Programming," *Communications of the ACM*, vol. 12, pp. 576–583, October 1969.

[72] Hoare, C.A.R., "Notes on Data Structuring," in *Structured Programming*, London: Academic Press Inc., 1972.

[73] Hoare, C.A.R., "The Emperor's Old Clothes," *Communications of the ACM*, vol. 24, pp. 75–83, February 1981. The 1980 Turing Award Lecture.

[74] Hoffman, D., "Practical Interface Specification," *Software — Practice and Experience*, vol. 19, pp. 127–148, February 1989.

[75] Howden, W. E., "A Functional Approach to Program Testing and Analysis," *IEEE Transactions on Software Engineering*, vol. SE-12, pp. 997–1005, October 1986.

[76] Huang, J.C., "Detection of Data Flow Anomaly Through Program Instrumentation," *IEEE Transactions on Software Engineering*, vol. SE-5, pp. 226–236, May 1979.

[77] Ince, D.C., "A Program Design Language Based Software Maintenance Tool," *Software — Practice and Experience*, vol. 15, pp. 583–594, June 1985.

[78] Ince, D.C. and Woodman, M., "The Rapid Generation of a Class of Software Tools," *The Computer Journal*, vol. 29, pp. 151–160, April 1986.

[79] Jachner, J. and Agarwal, V.K., "Data Flow Anomaly Detection," *IEEE Transactions on Software Engineering*, vol. SE-10, pp. 432–437, July 1984.

[80] Jensen, K. and Wirth, N., *PASCAL: User Manual and Report.* New York: Springer-Verlag, third ed., 1985. Revised by Mickel, A.B. and Miner, J.F.

[81] Johnson, W.L. and Soloway, E., "PROUST," *Byte*, vol. 10, pp. 179–190, April 1985.

[82] Johnson, W.L. and Soloway, E., "PROUST: Knowledge-Based Program Understanding," in *Conference on Software Maintenance-1985*, (Washington, D.C.), pp. 369–380, IEEE Computer Society Press, November 1985. Also in 'Readings in Artificial Intelligence and Software Engineering', eds. Rich, C. and Waters, R.C.

[83] Jones, C.B., *Systematic Software Development Using VDM.* Englewoods Cliffs, New Jersey: Prentice-Hall International, 1986.

[84] Karp, C.R., *Languages with Expressions of Infinite Length.* North-Holland, 1964.

[85] Kent, W., "A Simple Guide to Five Normal Forms in Relational Database Theory," *Communications of the ACM*, vol. 26, pp. 120–125, February 1983.

[86] Kernighan, B.W. and Ritchie, D.M., *The C Programming Language.* New Jersey: Prentice Hall, 1978.

[87] Knuth, D.E., *The Art of Computer Programming — Fundamental Algorithms.* Addison-Wesley, second ed., 1973.

[88] Korson, T.D. and Vaishnavi, V.K., "An Empirical Study of the Effects of Modularity on Program Modifiability," in *Empirical Study of Programmers*, (Soloway, E. and Iyengar, S., eds.), pp. 168–186, Norwood, N.J.: Ablex, 1986.

[89] Kuhn, D.R. and Hollis, C.G., "Simple Tools to Automate Documentation," in *Conference on Software Maintenance-1985*, (Washington, D.C.), pp. 203–210, IEEE Computer Society Press, November 1985.

[90] Leintz, B.P. and Swanson, E.B., "Software Maintenance: A User/Management Tug-of-War," *Data Management*, pp. 26–30, April 1979.

[91] Letovsky, S., "Cognitive Processes in Program Comprehension," in *Proceedings of the Conference on Empirical Studies of Programmers*, 1986.

[92] Letovsky, S., "The Lambda Calculus as a Representation Language for Program Plans," in *Proceedings of the IJCAI-87*, 1987.

[93] Letovsky, S. and Soloway, E., "Strategies for Documenting Delocalized Plans," in *Proceedings of the Conference on Software Maintenance — 1985*, (Washington, D.C.), pp. 144–151, IEEE Computer Society Press, November 1985.

[94] Letovsky, S. and Soloway, E., "Delocalized Plans and Program Comprehension," *IEEE Software*, vol. 3, pp. 41–49, May 1986.

[95] Leung, H.K.N. and Reghbati, H.K., "Comments on Program Slicing," *IEEE Transactions on Software Engineering*, vol. SE-13, pp. 1370–1371, December 1987.

[96] Linden, T.A., "The Use of Abstract Data Types to Simplify Program Modifications," *SIGPLAN Notices*, vol. 11, pp. 12–23, 1976.

[97] Linton, M.A., "Implementing Relational Views of Programs," in *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pp. 132–140, ACM SIGSOFT/SIGPLAN, May 1984. Appears as *Software Engineering Notes*, vol. 9, *SIGPLAN Notices* vol. 19, no. 5.

[98] Liskov, B., Atkinson, R., Bloom, T., Moss, E., Schaffert, J.C., Scheifler, R., and Snyder, A., *CLU Reference Manual.* New York: Springer-Verlag, 1980.

[99] Liskov, B. and Guttag, J., *Abstraction and Specification in Program Development.* The MIT Press, 1986.

[100] Lyle, J.R. and Gallagher, K.B., "Using Program Decomposition to Guide Modifications," in *Proceedings Conference on Software Maintenance - 1988*, (Washington, D.C.), pp. 265–269, IEEE Computer Society Press, October 1988.

[101] Manna, Z. and Waldinger, R., "Synthesis: Dreams $\Rightarrow$ Programs," *IEEE Transactions on Software Engineering*, vol. SE-5, no. 4, pp. 294–328, 1979.

[102] Metcalf, M., "FORTRAN 8X — The Emerging Standard," *ACM Fortran Forum*, vol. 6, pp. 28–47, April 1987.

[103] Metcalf, M. and Reid, J., *FORTRAN 8X Explained.* Oxford University Press, 1987.

[104] Meyer, B., "Eiffel: A Language and Environment for Software Engineering," *The Journal of Systems and Software*, vol. 8, pp. 199–246, June 1988.

[105] Meyer, B., *Object-Oriented Software Construction.* Prentice Hall International, 1988.

[106] Middelburg, C.A., "VVSL: A Language for Structured VDM Specifications," *Formal Aspects of Computing — The International Jornal of Formal Methods*, vol. 1, pp. 115–135, January-March 1989.

[107] Moon, D., "Object-Oriented Programming with Flavours," in *OOPSLA '86 Conference Proceedings*, ACM, 1986. Appeared as *Sigplan Notices*, vol. 21, no.11.

[108] Morissey, J.H. and Wu, L.S.Y., "Software Engineering: An Economic Perspective," in *Proceedings of the 4^{th} International Conference on Software Engineering*, pp. 17–19, September 1979.

[109] Munro, M. and Ward, M., "Intelligent Program Analysis Tools for Maintaining Software," in *UK IT 88 Conference Publication*, pp. 7–10, 1988.

[110] Naphtali, E. and Rich, M., "Some Practical Considerations Regarding an ADT-Obsessed Design," *Software Engineering Journal*, vol. 3, pp. 57–63, March 1988.

[111] Osterbye, K., "Abstract Data Types with Shared Operations," *SIGPLAN Notices*, vol. 23, pp. 91–96, June 1988.

[112] Osterweil, L.J. and Fosdick, L.D., "DAVE — A Validation Error Detection and Documentation System for FORTRAN Programs," *Software — Practice and Experience*, vol. 6, pp. 473–486, October-December 1976. Also in Tutorial: Software Testing and Validation Techniques (2nd edition) eds. Miller, E. and Howden, W.E.

[113] Osterweil, L.J., Fosdick, L.D., and Taylor, R.N., "Error and Anomaly Diagnosis Through Data Flow Analysis," in *Computer Program Testing — Proceedings of the Summer School on Computer Program Testing held at SO-*

GESTA, Urbino, Italy, (Chandrasekaran, B. and Radicchi, S., eds.), pp. 35–63, June 29 - July 3 1981.

[114] Ottenstein, K.J. and Ottenstein, L.M., "The Program Dependence Graph in a Software Development Environment," in *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pp. 193–196, ACM SIGSOFT/SIGPLAN, 1984. ACM SIGSOFT Software Engineering Notes vol. 9 ACM SIGPLAN Notices vol. 19, no. 5.

[115] Oulsnam, G., "Unravelling Unstructured Programs," *The Computer Journal*, vol. 25, pp. 379–387, August 1982.

[116] Page-Jones, M., *The Practical Guide to Structured Systems Design.* New York: Yourdon Press, 1980.

[117] Parker, R.A., Heninger, K.L., Parnas, D.L., and Shore, J.E., "Abstract Interface Specifications for the A-7E Device Interface Module," Tech. Rep., Naval Research Laboratory, Washington, D.C., November 1980. NRL Memorandum Report 4385 (Revised).

[118] Parnas, D.L., "Information Distribution Aspects of Design Methodology," in *Proceedings of the IFIP Congress — 1971*, pp. 339–344, North-Holland, 1972.

[119] Parnas, D.L., "On the Criteria to be used in Decomposing Systems into Modules," *Communications of the ACM*, vol. 15, pp. 1053–1058, December 1972.

[120] Parnas, D.L., "A Technique for Software Module Specification with Examples," *Communications of the ACM*, vol. 15, pp. 330–336, May 1972.

[121] Parnas, D.L., "On a "Buzzword": Hierarchical Structure," in *Proceedings IFIP Congress*, (Amsterdam, The Netherlands), North-Holland, 1974. Also

in 'Programming Methodology: A Collection of Articles by Members of the IFIP WG2.3', Gries, D., editor.

[122] Parnas, D.L., "On the Design and Development of Program Families," *IEEE Transactions on Software Engineering*, vol. SE-2, pp. 1–9, March 1976.

[123] Parnas, D.L., "Use of Abstract Interfaces in the Development of Software for Embedded Computer Systems," Tech. Rep., Naval Research Laboratory, Washington, D.C., June 1977. NRL Report 8047.

[124] Parnas, D.L., "Designing Software for Ease of Extension and Contraction," *IEEE Transactions on Software Engineering*, vol. SE-5, pp. 128–137, March 1979.

[125] Parnas, D.L. and Clements, P.C., "A Rational Design Process: How and Why to Fake It," *IEEE Transactions on Software Engineering*, vol. SE-12, pp. 251–257, February 1986.

[126] Parnas, D.L., Clements, P.C., and Weiss, D.M., "Enhancing Reusability with Information Hiding," in *Proceedings Workshop on Reusability in Programming*, pp. 240–247, 1983.

[127] Parnas, D.L., Clements, P.C., and Weiss, D.M., "The Modular Structure of Complex Systems," *IEEE Transactions on Software Engineering*, vol. SE-11, pp. 259–266, March 1985.

[128] Parnas, D.L. and Siewiorek, D.P., "Use of the Concept of Transparency in the Design of Hierarchically Structured Systems," *Communications of the ACM*, vol. 18, pp. 401–408, July 1975.

[129] Parnas, D.L., Weiss, D.M., Clements, P.C., and Britton, K.H., "Interface Specifications for the SCR (A-7E) Extended Computer Module," Tech. Rep.,

Naval Research Laboratory, Washington, D.C., December 1984. NRL Report 5502.

[130] Prather, R.E. and Guilieri, S.G., "Decomposition of Flowchart Schemata," *The Computer Journal*, vol. 24, pp. 258–262, August 1981.

[131] Raither, B. and Osterweil, L., "TRICS: A Testing tool for C," in *Proceedings of the European Conference on Software Engineering — 1987*, 1987.

[132] Reddy, U.S., "Transformational Derivation of Programs Using the Focus System," in *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, (Baltimore, Maryland), pp. 163–172, ACM Press, November 1988. Appeared as *Software Engineering Notes*, vol. 13, no. 5, November 1988 and as *Sigplan Notices*, vol. 24, no. 2, February 1989.

[133] Ross, D.L., "Classifying Ada Packages," *ACM SIGAda Ada Letters*, vol. VI, no. 4, pp. 53–65, 1986.

[134] Ryder, B.G., "Constructing the Call Graph of a Program," *IEEE Transactions on Software Engineering*, vol. SE-5, pp. 216–226, May 1979.

[135] Ryder, B.G., "An Application of Static Program Analysis to Software Maintenance," in *Proceedings of the Twentieth Hawaii International Conference on System Sciences 1987*, (Shriver, B.D., ed.), (California), pp. 82–91, Western Periodicals Company, January 1987.

[136] Ryder, B.G. and Carroll, M.D., "An Incremental Algorithm for Software Analysis," *SIGPLAN Notices*, vol. 22, pp. 171–179, January 1987. Proceedings of ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, December 1986.

[137] Sarraga, R.F., "Static Data Flow Analysis of PL/1 Programs with the PROBE System," *IEEE Transactions on Software Engineering*, vol. SE-10, pp. 451–459, July 1984.

[138] Scherlis, W.L., "Program Improvement by Internal Specialization," in *ACM Symposium on Principles of Programming Languages*, pp. 41–49, 1981.

[139] Sengler, H.E., "A Model of the Understanding of a Program and its Impact on the Design of the Program Language GRADE," in *The Psychology of Computer Use*, (Green, T.R.G., Payne, S.J., and van der Veer, G., eds.), pp. 91–106, London: Academic Press, 1983.

[140] Shneiderman, B., *Software Psychology: Human Factors in Computer and Information Systems.* Cambridge, Massachusetts: Winthrop Publishers Inc., 1980.

[141] Shneiderman, B., "Control Flow and Data Structure Documentation: Two Experiments," *Communications of the ACM*, vol. 25, pp. 55–63, January 1982.

[142] Shneiderman, B., Shafer, P., Simon, R., and Weldon, L., "Display Strategies for Program Browsing," in *Proceedings of the Conference on Software Maintenance — 1985*, (Washington, D.C.), pp. 136–143, IEEE Computer Society Press, November 1985.

[143] Shneiderman, B., Shafer, P., Simon, R., and Weldon, L., "Display Strategies for Program Browsing: Concepts and Experment," *IEEE Software*, vol. 3, pp. 7–15, May 1986.

[144] Sneed, H.M. and Jandrasics, G., "Inverse Transformation of Software from Code to Specification," in *Proceedings of the Conference on Software Main-*

tenance — 1988, (Washington, D.C.), pp. 102–109, IEEE Computer Society Press, 1988.

[145] Soloway, E. and Ehrlich, K., "Empirical Studies of Programming Knowledge," *IEEE Transactions on Software Engineering*, vol. SE-10, pp. 595–609, September 1984. Also in 'Readings in Artificial Intelligence and Software Engineering', eds. Rich, C. and Waters, R.C.

[146] Stroustrup, B., *The C++ Programming Language*. Reading, Massachusetts: Addison-Wesley, 1986.

[147] Sundblad, Y., "The Ackermann Function. A Theoretical, Computational, and Formula Manipulative Study," *Bit*, vol. 11, pp. 107–119, 1971.

[148] Taylor, R.N. and Osterweil, L.J., "A Facility for Verification, Testing, and Documentation of Concurrent Process Software," in *Proceedings COMPSAC*, pp. 36–41, November 1978.

[149] Taylor, R.N. and Osterweil, L.J., "Anomaly Detection in Concurrent Software by Static Data Flow Analysis," *IEEE Transactions on Software Engineering*, vol. SE-6, pp. 265–278, May 1980. Also in 'Tutorial: Software Testing and Validation Techniques' (2nd edition) eds. Miller, E. and Howden, W.E.

[150] Thomas, J.W., *Module Interconnection in Programming Systems Supporting Abstraction*. PhD thesis, Brown University, 1976.

[151] van Delft, A.J.E., "Comments on Oberon," *Sigplan Notices*, vol. 24, pp. 23–30, March 1989.

[152] van Kiet, L., *The Module: A Tool for Structured Programming*. PhD thesis, Swiss Federal Institute of Technology, Zurich, 1978. ETH Diss. Nr. 6153.

[153] Ward, M., "Transforming a Program into a Specification," Tech. Rep., University of Durham, Durham, England, 1988. Computer Science Technical Report 88/1.

[154] Ward, M., *Proving Program Refinements and Transformations.* PhD thesis, Oxford University, Computer Science, 1989.

[155] Ward, M., Calliss, F.W., and Munro, M., "The Use of Transformation in The Maintainer's Assistant," Tech. Rep., University of Durham, Durham, England, 1988. Computer Science Technical Report 88/9.

[156] Ward, M., Calliss, F.W., and Munro, M., "The Maintainer's Assistant," in *Proceedings of the Conference on Software Maintenance — 1989*, (Washington, D.C.), pp. 307–315, IEEE Computer Society Press, 1989.

[157] Wegner, P., "Dimensions of Object-Based Language Design," in *OOPSLA '87 Conference Proceedings*, pp. 168–182, ACM, 1987. Appeared as *Sigplan Notices*, vol. 22, no.11.

[158] Weiser, M., "Programmers use Slices when Debugging," *Communications of the ACM*, vol. 25, pp. 446–452, July 1982.

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

[160] Weiser, M.D., *Program Slices: Formal, Psychological, and Practical Investigation of an Automatic Program Abstraction Method.* PhD thesis, University of Michigan, 1979.

[161] Williams, M.H., "Generating Structured Flow Diagrams: the Nature of Unstructuredness," *The Computer Journal*, vol. 20, pp. 45–50, February 1977.

[162] Williams, M.H., "A Comment on the Decomposition of Flowchart Schemata," *The Computer Journal*, vol. 25, pp. 393–396, August 1982.

[163] Williams, M.H. and Ossher, H.L., "Conversion of Unstructured Flow Diagrams to Structured Form," *The Computer Journal*, vol. 21, pp. 161–167, May 1978.

[164] Wilson, C. and Osterweil, L.J., "OMEGA-A Data Flow Analysis Tool for the C Programming Language," *IEEE Transactions on Software Engineering*, vol. SE-11, pp. 832–838, September 1985.

[165] Wirth, N., "Program Development by Stepwise Refinement," *Communications of the ACM*, vol. 14, pp. 221–227, April 1971. Also in 'PROGRAMMING METHODOLOGY A Collection of articles by members of the IFIP G2.3,' (ed. Gries, D.).

[166] Wirth, N., *Systematic Programming: An Introduction.* Englewood Cliffs, New Jebsey: Prentice-Hall Inc., 1973.

[167] Wirth, N., "On the Composition of Well-Structured Programs," *Computing Surveys*, vol. 6, pp. 247–259, December 1974.

[168] Wirth, N., "Modula: A Language for Modular Multiprogramming," *Software — Practice and Experience*, vol. 7, pp. 3–35, January-February 1977.

[169] Wirth, N., *Programming in Modula-2.* New York: Springer-Verlag, third corrected ed., 1985.

[170] Wirth, N., "Type Extensions," *ACM Transactions on Programming Languages and Systems*, vol. 10, pp. 204–215, April 1988.

[171] Wirth, N., "From Modula to Oberon," *Software — Practice and Experience*, vol. 18, pp. 661–670, July 1988. Also appears as a technical report from ETH in September 1987.

[172] Wirth, N., "The Programming Language Oberon," *Software — Practice and Experience*, vol. 18, pp. 671–690, July 1988. Also appears as a technical report from ETH in September 1987.

[173] Yau, S.S. and Grabow, P.C., "A Model for Representing Programs using Hierarchical Graphs," *IEEE Transactions on Software Engineering*, vol. SE-7, pp. 556–574, November 1981.

[174] Yourdon, E., *Techniques of Program Structure and Design.* New Jersey: Prentice-Hall Inc., 1975.

[175] Yourdon, E. and Constantine, L.L., *Structured Design Fundamentals of a Discipline of Computer Program and Systems Design (2nd Edition).* New York: Yourdon Press, 1978.

Index