Safe Data Structure Visualisation

Richard A. Eyre-Todd

Ph.D.
University of Edinburgh
1993
Abstract

A simple three layer scheme is presented which broadly categorises the types of support that a computing system might provide for program monitoring and debugging, namely hardware, language and external software support. Considered as a whole, the scheme forms a model for an integrated debugging-oriented system architecture. This thesis describes work which spans the upper levels of this architecture.

A programming language may support debugging by preventing or detecting the use of objects that have no value. Techniques to help with this task such as formal verification, static analysis, required initialisation and default initialisation are considered. Strategies for tracking variable status at run-time are discussed. Novel methods are presented for adding run-time pointer variable checking to a language that does not normally support this facility. Language constructs that allow the selective control of run-time unassigned variable checking for scalar and composite objects are also described.

Debugging at a higher level often involves the extensive examination of a program's data structures. The problem of visualising a particular kind of data structure, the hierarchic graph, is discussed using the previously described language level techniques to ensure data validity. The elementary theory of a class of two-level graphs is presented, together with several algorithms to perform a clustering technique that can improve graph layout and aid understanding.
Acknowledgements

I would like to sincerely thank my supervisor, Rob Pooley, for the guidance, support, patience and friendship which he gave me during the production of this thesis. I would also like to thank Mark Jerrum and Alistair Sinclair for the many discussions we have had on graph theoretical topics over the years. Finally, many thanks must go to both Mark Jerrum and Jane Hillston for their constructive and incisive comments on earlier drafts of Chapters 3 to 8.

Lindsay and Sophie gave me their love and inspired me to finish. I will be forever grateful.

This work was funded by a Research Studentship from the Science and Engineering Research Council.

This is the revised version of my thesis. The comments and suggestions put forward by my examiners, Brian Wichmann and Frank Stacey, have been most helpful.

Declaration

I declare that this thesis was composed by myself, and that the work it describes is my own, except where stated in the text. Some of the work described in Chapter 4 will appear in [ET94].
Table of Contents

1. Introduction 1

2. Support for Debugging 5
   2.1 Hardware support .......................... 5
       2.1.1 General Requirements .................. 6
       2.1.2 Control of Execution .................. 7
       2.1.3 State Manipulation ................... 8
       2.1.4 Non-processor Hardware ............... 8
   2.2 Language Support for Debugging .......... 11
   2.3 Software System Support ................. 14
       2.3.1 Program Animation .................... 15
       2.3.2 Data Display .......................... 16
       2.3.3 Graph Drawing ......................... 17
       2.3.4 Graphical Abstraction ................. 24

3. Unassigned Variables 27
   3.1 Introduction ................................ 28
       3.1.1 Required Initialisation ................. 29
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2</td>
<td>Eliminating Unassigned Variable Errors using Formal Verification</td>
<td>31</td>
</tr>
<tr>
<td>3.3</td>
<td>Eliminating Unassigned Variable Errors using Static Analysis</td>
<td>33</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Flow Analysis</td>
<td>33</td>
</tr>
<tr>
<td>3.3.2</td>
<td>SPADE</td>
<td>37</td>
</tr>
<tr>
<td>3.3.3</td>
<td>MALPAS</td>
<td>38</td>
</tr>
<tr>
<td>3.3.4</td>
<td>Conclusions and Problems</td>
<td>39</td>
</tr>
<tr>
<td>3.4</td>
<td>Eliminating Unassigned Variable Errors using Required or Default Initialisation</td>
<td>42</td>
</tr>
<tr>
<td>3.5</td>
<td>Detecting Unassigned Variable Errors in Languages with No Initialisation</td>
<td>47</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Physical representation of Unassigned</td>
<td>50</td>
</tr>
<tr>
<td>3.5.2</td>
<td>The Run-time Functions of Unassigned Variables</td>
<td>53</td>
</tr>
<tr>
<td>3.5.3</td>
<td>Variables that are Never Unassigned</td>
<td>54</td>
</tr>
<tr>
<td>3.5.4</td>
<td>The Dynamic Valuation and Propagation of Unassigned Variables</td>
<td>55</td>
</tr>
<tr>
<td>3.5.5</td>
<td>Adding Language Support for the Detection of Unassigned Variables</td>
<td>56</td>
</tr>
<tr>
<td>4.1</td>
<td>Dangling Reference Detection</td>
<td>63</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Never Deallocate</td>
<td>64</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Double Indirection</td>
<td>65</td>
</tr>
</tbody>
</table>
# Table of Contents

4.1.3 The Key Method .......................... 67  
4.1.4 Summary .................................. 68  
4.2 Adding Run-time Checking to C++ ............... 69  
4.2.1 C++ and Smart Pointers .................... 69  
4.2.2 Extending the Key Method to C++ ............. 70  
4.2.3 Extending the Double Indirection Method to C++ 76  
4.2.4 Comparison of the Two Implementation Methods .. 80  
4.2.5 Summary .................................. 82  
4.3 Objects and Pointer Arithmetic .................. 82  

5. Composite Objects ............................ 90  
5.1 Unassigned status ................................ 90  
5.2 State Tracking ............................... 92  
5.3 Examples .................................... 95  
5.4 Simplifications ................................ 97  
5.5 Composite State Tracking revisited ................ 102  

6. Graph Display ................................. 105  
6.1 Introduction .................................. 105  
6.2 Graph Theory .................................. 106  
6.2.1 Definitions and Notation .................... 106  
6.2.2 Theory ................................... 107  
6.2.3 Calculating $T^z_{x,y}$ ......................... 112  
6.2.4 Graph Planarity and Edge Crossings .......... 113
6.3 Bipartite Graph Information Content ........................................... 123
6.3.1 Motivation for edge reduction algorithms ................................. 123
6.3.2 Detecting complete bipartite subgraphs .................................. 125

7. Graph Theoretical Methods .......................................................... 128

7.1 Determining the presence of a $K_{x,y}$ subgraph ............................ 129
7.1.1 Network flow Theory .............................................................. 130
7.1.2 The Ford-Fulkerson method ..................................................... 135
7.2 Finding a $K_{x,y}$ subgraph ......................................................... 140
7.2.1 Extended algorithm and observations ..................................... 140
7.2.2 The Unlabelled Vertex Subgraph Problem ................................ 145

8. Cluster Detection and Pattern Matching ....................................... 158

8.1 Approximate solutions ............................................................... 159
8.1.1 Newberry's algorithm ......................................................... 159
8.1.2 Another approximate algorithm .......................................... 164
8.2 Structure-oriented Algorithm ..................................................... 174
8.2.1 Implementation ................................................................. 196
8.2.2 Conclusions ................................................................. 197

9. Conclusions ................................................................................. 203

Bibliography .................................................................................. 210
To Mum, Dad and Gran, and in loving memory of my Grandma, Catherine B. Eyre-Todd and my Grandad, Squadron Leader P. Frank Varey MBE.
Chapter 1

Introduction

The process of development of a new piece of software always includes a phase of removing errors or bugs from the code which cause the system behaviour to deviate from its specification. This debugging process is often referred to as an art rather than a science due to the absence of development of a single, formal, precise debugging methodology. Instead, various techniques and tools have been devised that simply provide help to perform the task [FB63,ED66,Sat72,LG76,Lau79,Tra79,Gla80,Ham83,ST83].

Modern research has shown the benefit of creating an integrated set of tools to aid the development of production quality software [Gol84,Rei85,SZBH86]. This thesis advances the related proposition that equivalent benefits would be gained from an integrated approach to program monitoring and debugging. Figure 1–1 presents a simple framework that broadly categorises the types of support for debugging that a modern computing system might provide. Taken as a whole, this framework is a model for a Fully Integrated Debugging-Oriented System Architecture. As general-purpose computing systems become more powerful and integrated circuit densities increase, it will become increasingly difficult to accurately monitor program behaviour. Many system functions, which were previously monitorable by external hardware attached to bus and interrupt lines, will become internal features of custom silicon chips. Therefore it is very likely that monitoring and
Figure 1-1: *Fully Integrated Debugging-Oriented System Architecture*

documenting facilities will have to be considered *from the start* as part of a system's *design requirements*, and will *demand* the integration of hardware, programming language and external software support.

**Motivation**

This work presents new results which span the upper levels of the integrated architecture model. However the original motivation for the research lies in the problems encountered during an earlier project. An attempt was made to implement a simple user interface for a program that illustrated hierarchic graph algorithms, using the InterViews package [Lin89] for the X Windows system [Sch86]. Two major sources of difficulty were encountered:

1. The system did not support unassigned variable checking. This led to fundamental problems in tracking algorithmic bugs in both the application and the graphics package. The application code was complex and the run-time system was unable to detect dangling pointers. This meant that it was extremely hard to determine whether a graph that had been drawn was "real", or whether it was a structure that had been deallocated from memory. The memory occupied by a deallocated structure was typically never reallocated,
and so the pictures that were drawn of real and deallocated structures were usually indistinguishable. This led to the work presented in Chapters 3 to 5, where the problems of unassigned variables and the management of variable status at run-time are described.

2. Once drawn on a high resolution display and proven to be "real", the sheer complexity of some types of graph was found to make their presentation very difficult to understand. Chapters 6 to 8 consider the underlying theory of one class of complex graphs and describe some novel algorithms that can be used to improve their readability when drawn.

Thesis Outline

Chapter 2 considers each of the levels of the debugging-oriented architecture in detail, presenting examples of techniques, methodologies, tools and systems which have been developed to provide each type of debugging support. The benefits of the integrated approach adopted by some research is shown where appropriate.

Chapters 3 to 5 describe the methods that language designers have used to cope with the problems of unassigned scalar and composite objects. Techniques are presented for tracking variable status at run-time. Novel methods are described for adding run-time pointer variable checking to a language that does not normally support this facility, in such a way that the program itself can detect an error rather than the run-time system. In addition, language constructs are presented for the selective control of run-time unassigned variable checks.

Chapters 6 to 8 consider the problems of displaying very complex hierarchical graphs which are known to be "real". An elementary theory of simple two-level hierarchies, or bipartite graphs, is presented. Several algorithms to perform a type of bipartite graph clustering operation that improves the layout of this type of
graph, and conveys the information contained within it more clearly, are described. Conclusions on the effectiveness of the clustering operation are drawn.

Chapter 9 presents a summary of the major results and contributions presented in the thesis. The possible directions of future research to extend the work within the framework of the integrated debugging-oriented architecture are outlined.
Chapter 2

Support for Debugging

2.1 Hardware support

Hardware support for debugging is highly specialised in general, and usually only provides a small functionality. Most modern hardware designs are governed by a desire for high performance and so the impact on cost, chip area and processing speed that hardware debugging support can impose means that most common processors have little or no built in facilities [CL87]. However, as the cost of producing reliable software increases while the cost of hardware generally decreases, hardware support for debugging should become more important. This is especially true for debugging on parallel architectures where there is a desire to avoid the probe effect [Gai86], the distortion of program execution caused by interactive analysis.

The requirements for the architectural support of debugging are comprehensively surveyed in [Joh82b] and [MST82], and will be briefly summarised here.
2.1.1 General Requirements

A hardware debugging facility must be efficient i.e. provide a fast response, and so execution overhead should only be incurred when specific debugging operations are being used. In the SPAM system [Joh81b] for example, a bit in the program status word can be used to disable breakpoint detection, and so if breakpoints are not being used the overhead required to support the facility is very small. Another method for improving efficiency is to perform debugging operations in parallel using carefully constructed sequences of microcode.

Instruction pipelining and optimisation should not distort the user’s high level view of the execution and structure of the source program. Therefore the "semantic gap" between the source and object programs should be minimised before debugging software is activated (for example by flushing the pipeline or “unravelling” optimisations).

Software reliability considerations lead to the desire for hardware detection of semantic and logical errors [Mye73]. These types of errors include:

- unassigned variables
- out of range errors
- access and type violations and
- using data as instructions or instructions as data.

In a tagged architecture [Feu73] each storage item is associated with a description of the type of the item. A descriptor [BB81] is a generalised tag that contains more elaborate information about the properties of the data item e.g. type, assigned status, size etc. Storage tags are used in the Burroughs B1700 [Mye73], the Burroughs B6700 [Org73], the Intel iAPX 432 [Tyn81], SPAM, SWARD [Mye73] and X-Tree [Dit80].
Some types of error can be detected using a less expensive mechanism. For example, a special unit in the MU5 [Ibb82] performed bounds checking for one dimensional vectors in parallel with the calculation of the address of an element. Hill [Hil81] showed that this type of error can be trapped by adding one new instruction, two registers and an ALU flag bit to a processor design.

### 2.1.2 Control of Execution

The control of program execution is of fundamental importance in debugging. A hardware mechanism such as the instruction trap [Joh81a] can be used to provide execution breakpoints. The IBM system 370 [CP78] used a bit in the status word and two bounds registers to define a set of instructions that are to be trapped, while the NCR Criterion [Sha78] reserved a ‘breakpoint’ bit in each instruction word.

Different types of trap can be used to provide different functions. A trap after the execution of a specified instruction, for example ‘GOTO A’ where ‘A’ is a variable, is sometimes useful for branch traces, as it does not require a trap to be set at all the possible values of ‘A’. A trap before a successful branch supports branch traces, retrospective traces and reversible execution [Zel73,TR81,WF81, ACS84]. A facility to trap on the execution of the next instruction can be used to provide stepwise execution. Suspension of execution when a data value is changed can provide assertion and data driven breakpoints, profiles, traces and display animation. Traps on data access or update are provided in SPAM and X-Tree using a descriptor flag bit. This mechanism is also used to implement traps at the beginning and end of procedures for generating call traces and profiles. If the number of memory addresses that must be monitored is small, it may be possible to hold these addresses in special purpose registers (or more expensive associative memory) and compare them with the address used on each memory access [CLW90]. The ability to have an arbitrary number of dynamically controllable
traps is a useful feature provided by SPAM. Interprocess traps for breakpoints and global monitoring are an open research area [BH83, BFM+83, MC86, Coo87].

2.1.3 State Manipulation

Hardware can also provide support for the explicit determination and manipulation of a program's state and for profiling operations. When a program is suspended the user requires a precise reason, which could be provided in a status register as on the Intel iAPX432 architecture. An accurate mapping from the machine language state to the corresponding source language state is also required, for example SPAM and SPLM [Mye73] provide descriptors and symbol tables for this purpose. Special instructions may be required for accessing vital machine state information. More complex operations such as recording successful branches and procedure calls [SS75] can also be provided for tracing and profiling.

2.1.4 Non-processor Hardware

Debugging and profiling information can also be provided by hardware external to the processor. On a small scale, a simple counter can be attached to the system bus [CL87] which is decremented on each instruction and raises an exception on reaching the value zero. Assuming that process execution is reproducible at the instruction level, the counter can be used to provide data breakpoints, instruction counting, profiling and reverse execution. The integration of this type of counter into a processor design, although intended for the support of checkpoints and program restarts, has been performed by Hewlett-Packard [HP87]. Software simulation of this facility has also been shown to be possible using a combination of the program counter and a count of the number of backward branches [MCL89].

Other types of debugging support require so many extra components that processor integration would be too costly at present. Special hardware that interfaces
with the processor memory bus can be used to monitor program execution at
low level, in terms of the addresses of the instructions which are executed and
the data which is accessed [EHCC66, Pla84, BM84, Bem86, TFC90]. Another less
expensive method is to use existing specialised hardware for different debugging-
related purposes. Abramson and Rosenberg [AR83] showed how the virtual ad-
dress translation hardware of the MONADS II computer can be used to detect
when a breakpoint is set within a page and then search a list of breakpoints held
in fast memory. For a single breakpoint per page the overhead is only one memory
cycle per test, however this increases when the number of breakpoints increases.
A different solution proposed by Charlton et al. [CLW90] used a fast, general pur-
pose data cache provided by the user-microprogrammable High-Level Hardware
ORION [Hig84] architecture to hold breakpoint addresses. The usual overhead,
one cache reference and comparison per memory reference to a page containing a
breakpoint, may often be reduced by performing the cache operation in parallel
with the memory reference it corresponds to. When a page does not contain a
breakpoint, the overhead is negligible.

The Test and Measurement Processor (TMP) described in [HW86] is an en-
tirely separate computing system that permanently monitors the special program
events generated by a target computing system. The TMP test and perform-
ance measurement software gathers and analyses these events in parallel with the
execution of the measured system.

The POSIE Project Testbed [HIPW90] is a distributed parallel computing sys-
tem with special monitoring hardware for collecting and processing performance
and debugging data. Events generated by each processor are fed to an event moni-
tor system using a common monitoring bus. The use of a separate monitoring bus
means that the interprocessor message traffic is not disturbed. A further commu-
nications monitor system copies selected message traffic from the interprocessor
communication bus using a hardware filtering mechanism.

Both the TMP and POSIE systems generate an event by writing one or more
words to a specific memory location. Events can therefore be created by short, fast segments of software instrumentation (on the TMP system, the overhead that this imposes has been measured as typically less than 0.1%). This hybrid hardware/software approach is a reasonable compromise between pure hardware monitoring, which tends to be very expensive, and pure software monitoring which, although cheap, requires large amounts of monitoring code to be inserted into a program and is therefore highly intrusive. Rudolf [Rud89] and Reed [Ree89] also present a hybrid approach to monitoring a commercially available system, the Intel iPSC/2 hypercube.

An alternative approach is to arrange for the monitoring system to exhibit the behaviour of a coprocessor [Gor91] and to use existing coprocessor communication protocols for monitoring purposes. Low-overhead coprocessor instructions can be added to existing program code by hand, by compilers or by program analysis tools. The motivation behind this approach is the current trend in microprocessor development towards moving more of the functions of a system onto the processor chip. Advances in wafer-scale integration techniques mean that very soon it will be possible to place several processors and large amounts of memory in a single package, where none of the internal buses can be monitored by external hardware. Also, modern data caching algorithms can arbitrarily delay the appearance of a store operation on an external memory bus. Therefore future microprocessor buses may not be able to properly support bus snooping for any of the previously described monitoring schemes. The advantages of the coprocessor approach are that it is only lightly invasive, it can monitor both low and high level events, it reduces the amount of target state information that must be recorded, requires little special purpose support hardware and exhibits a greater degree of target system independence since, unlike bus protocols, coprocessor protocols are usually constant within the product line of a single manufacturer.

Information gained by program monitoring can also be used to achieve deterministic replay of parallel program execution [LMC87,PL89,For89]. Bacon and
Goldstein [BG91] presented a hybrid hardware / software approach which efficiently monitors the order of memory references on a shared memory multiprocessor architecture, by recording a subset of the cache traffic between the processors and memory. This logging operation is performed in hardware, using extra bandwidth which is available on the system bus to record two byte log entries. The logging hardware has its own dedicated memory and disk, which gives the capacity to monitor up to 24 minutes of fine-grained parallel execution with virtually no degradation of system performance.

2.2 Language Support for Debugging

The design of a programming language is often motivated by the desire to improve program reliability and reduce the costs of maintenance [ACM75, ACM77]. However there is little evidence to suggest exactly which factors in a design actually support these aims [Win84]. The goal of reliable programming is to construct a program that exactly meets its specification, and so facilities which increase the fraction of errors that are detected must be important in this respect. A language design that improves the detection and elimination of compile-time, run-time and logical programming errors, or in other words supports the debugging process, can therefore contribute to this goal.

Compile-time error checking is well understood [Gri71, Bor79, ASU86]. The elimination of logical programming errors is an active research area, extensively surveyed in [Sev87] and [DE88]. However this type of debugging is usually supported by software external to the compiler (in the top layer of the integrated debugging architecture). One language-level feature that can be used to detect logical errors is the assertion [BBG⁺71] mechanism. An assertion is an executable specification of an invariant condition at the point at which it is placed in a program. If at run-time this invariant is found not to be true then an error is
signaled. The object-oriented language Eiffel [Mey88] extends this idea to allow
the programmer to specify pre-, post- and invariant conditions for instances of
a class. The assertion is therefore the language-level equivalent of a conditional
breakpoint.

The work of Chapters 3 to 5 is primarily concerned with language-level mech-
anisms for the prevention and detection of run-time unassigned variable errors, in
order to improve software reliability. Winner [Win84] put forward the following
propositions as a basis for discussing the relationship between language design and
reliable software.

**Proposition 2.1** *A virtual machine, for example a programming language, sup-
ports reliable programming to the extent that it coherently realises the programmer’s
logical view of the goals of his programs.*

An implication of this proposition is that a more reliable program will result if
the system informs the programmer of any deviations from the intended execution
path. Therefore the language design must provide methods for describing the
programmer’s intentions, and if the process of carrying out those intentions causes
an error, this must be accurately reported and appropriate recovery action taken.

**Proposition 2.2** *It is bad for a program to use the value of an object if the
programmer intends the object to be logically unvalued. The word “use” here is
not used in the sense of “examine” ; it is often reasonable to find out whether or
not an object is unvalued.*

It follows that, in the sense of Proposition 2.1, a system is better if it detects
and responds to the use of an object with no value, unless the programmer has
arranged otherwise. In practice, tests conducted by the National Physical Labor-
atory using the Standard Model Pascal Implementation [WH86] found that most
large programs contain errors, the most common of which is the use of an un-
assigned variable. This result has placed more emphasis on the ability to detect
the use of unassigned variables in the assessment of compilers [CW83, amended annually].

Run-time errors such as array bounds exceptions and illegal pointer references can also be viewed as an attempt to use an object with no value. However, as will be demonstrated in Chapter 4, a language may trap these kinds of error using very different mechanisms.

Other types of run-time debugging support which a language may provide include:

- subrange checking, where a value is checked to ensure that it lies within a specified range before assignment to a subrange variable

- tag-field checking, where the tag field of a variant record or union type is checked before a specific component is accessed

- checks for reading past the end of an input file, and

- checks for the deallocation of a referenced variable while it, or any part of it, is in use as a procedure parameter passed by reference or as the record variable of a with statement [WE88].

Winner also stated three requirements which might be used to critically analyse a language design in relation to Propositions 2.1 and 2.2:

1. **Expressive completeness**: it should be possible to conveniently express all the functions that are required.

2. **Consistency in the treatment of objects**: if the internal structure of an object does not affect the intended meaning of some use of that object, then the actual semantics of that use should not depend on the object's structure. For example, the semantics of passing variables as parameters
CHAPTER 2. SUPPORT FOR DEBUGGING

should be the same for array and scalar types. Also, given an operand, the meaning of an operation should not depend on how it is expressed. For example, with suitable variables $X$ and $Y$, the statements $X := Y$ and \texttt{assign}(X,Y) should mean the same.

3. **Implementation efficiency**: run-time speed is an important consideration. However, if a language satisfies the above two criteria it may have difficult or costly implementation problems as a result.

Chapter 5 will illustrate some conflicts which result from the consideration of these factors.

2.3 Software System Support

Early research in software development technologies resulted in the development of *software support environments*, a set of loosely coupled tools which help with the various phases of software development: management, planning, construction and maintenance [KM81, Ost83]. Further research resulted in the development of *programming environments*, the subset of these tools which most directly relate to the programming task, including systems for designing, documenting, editing, coding and debugging programs. A programming environment may be best suited to support and encourage the use of specific languages and programming methodologies. A *language environment* such as Interlisp [TM81] is a tightly integrated programming environment designed around a single language.

Advances in visual display technology led to improvements in language environment user interfaces as exemplified by Smalltalk [Gol84], which introduced a new object-oriented language based on Simula-67 together with the idea of overlapping windows. The environment for the Cedar language [SZBH86] introduced the window tiling paradigm and enhanced the original window concept with icons and
buttons. The Pecan system [Rei85] introduced the idea of presenting the same data simultaneously in different formats or from different perspectives. These “multiple views” share the same internal representation of the data. Whenever this representation changes, the corresponding views are updated simultaneously so that the user can choose to use the most appropriate view at any time. The Garden environment [Rei86] extends the idea of a graphical view to work with languages that are best expressed graphically, rather than textually. Here conceptual models are created by defining objects and operations which can be performed upon them, using a common object-oriented internal representation. Other notable programming environments are the Cornell Program Synthesiser [TR81], the IPE environment for GC [MMF81], COPE [Arc81,CDW84], and Magpie for Pascal [DMS84]. Visual programming environments include Thinglab [Bor86], Rehearsal World [FG84], PT [HA88] and Forms [Amb87]. In [BS86] Bahlke and Snelting describe the PSG system, which uses a formal definition of a language to generate an interactive, language specific programming environment. The generated environment includes a language-based editor, an interpreter and a program fragment library system.

2.3.1 Program Animation

A program animator [Ber91] is a system which evaluates a program step by step, representing program execution as an animated sequence of views of the source program and its associated data structures. This type of tool can help programmers to understand and debug programs [FB63,Bal69,TSP83] as well as teach new programming skills or a new programming language [Pan81,Lea84,BS85]. A debugging system typically provides all of the functionality of a program animator except the display [Ber91], although modern research is starting to recognise the importance of this factor and so provide an execution display of increasing capability [Raj86,DC86,ISO87,Moh88].
2.3.2 Data Display

The display of the structures which a program manipulates is of fundamental importance to a program animation system. Little gain is made from the use of high level languages if a user must descend to a lower level such as assembly language, or unformatted memory dumps, in order to view data structures [Bal69]. Therefore a program animator must, as a minimum, present and control the state of a program in the same high-level symbolic terms as the language it is written in.

In Brooks' model [Bro83] a program is viewed as a set of levels of abstraction, with the lowest level representing the domain of the language and the highest level representing the problem domain. The programmer's task is to construct these levels such that each is defined in terms of the one below. Therefore this theory suggests that a more advanced program animator should be able to present views of the more complex structures present in the higher levels of abstraction.

Myers' Incense system [Mye80] for the Mesa programming language allows the interactive display of program data structures in the form that the user would draw in order to explain them. Standard formats are provided for data display together with an artist facility to allow user-defined presentations. In this way an analogical display can be constructed which uses abstract pictures such as icons, arrows, clocks, graphs etc. to convey the data, and thus present it in a form which is easier to understand by using analogies with the real world.

The BALSA system [BS85] is an integrated environment designed to animate algorithms and programs using a high resolution bitmap display. Complex structures such as stacks, trees and general graphs can be animated to show the operation of the algorithms under study, using custom user-programmed display routines. The algorithm designer identifies key "interesting events" which are used to drive the graphical presentation. Multiple analogical views of the data can be constructed. Large entities can be summarised in one view showing only the high
level structure, while some parts of the structure are shown in great detail in another. BALSA has been used for systems programming, teaching and research.

The idea of a pre-programmed display routine for a specific type of data structure is very powerful and flexible. In PV [BCH85] and Provide [Moh88] a range of pictures created using a bitmap editor can be related to the variables of a program. In Garden [Rei86] a special display can be designed for each data type which is used for program output and animation. In Graphtrace [GKS83], views of Pascal heap structures can be constructed and the user can interactively specify the required layout. The VIPS debugger for Ada [ISO87] integrates graphical data display with facilities to trace program control flow and data dependencies.

Ideally, a program animation system should possess a library of display routines for a variety of common data types. In this case it will often be possible to re-use large parts of existing display routines, on the hopefully rare occasions when a new display is required. One type of data structure that has received considerable attention in this respect is the graph.

2.3.3 Graph Drawing

Graphs [Gib85] are common data structures used to solve a wide range of computational problems. Manual graph layout is a difficult task when there are a large number of vertices and edges, or when the data to be displayed has been generated by some kind of large scale computation e.g. a compiler generated parse tree [MRH91]. The resulting need for automatic graph layout has led to the development of specialised graph drawing algorithms which determine two dimensional coordinates for the vertices and edges. These algorithms are surveyed in [TBB88] and [ET87] and a brief overview will be presented here.
CHAPTER 2. SUPPORT FOR DEBUGGING

Principles

A rendering algorithm is usually specifically designed for a certain type of graph, such as a tree, planar, undirected or directed graph. In some techniques the positioning of the vertices is restricted in some way, for example they may be positioned on concentric circles [Car80], parallel lines [STT81,RDM+87] or on grid points [BNT86,TBB88]. Edges may be drawn as straight lines, segmented straight lines or spline curves.

The readability of a graph, or how well it communicates the information that it contains, is usually heavily influenced by aesthetic principles such as the minimisation of edge crossings [TBB88], the uniform distribution of vertices and edges [War77,Car80], and the minimisation of total area, edge lengths, edge bends and differences in vertex size [Car80]. An algorithm may have to take account of constraints imposed by the user on the positioning of certain vertices and edges. For example there may be a requirement for hierarchy [RT81], symmetry [Ead84, LNS85], node adjacency or the placement of vertices on the graph boundary [Car80]. In many applications a sequence of generated drawings may have only slight differences, and an algorithm may be required to perform only local updates on the regions of the graph that have changed [TBB88].

An interactive application requires fast graph drawing algorithms. However many of the aesthetic principles such as minimisation of edge crossings [Joh82a], area [Sto84] and edge lengths [BC87], together with decision problems such as finding the maximal planar subgraph [GJ79], have been proven intractable. Most algorithms are therefore driven by heuristics.

Trees

Algorithms for the drawing of trees are presented in [WS79,Vau80,RT81]. Each algorithm has time complexity $O(n)$ and produces a result of width $O(n)$. Some associated complexity results are given in [SR81,Ull84].
Planar Graphs

A graph is planar if it can be drawn with no edge crossings. Many algorithms have been developed for this type of graph [Tut63, Woo82, Sto84, CON85, Tam87, TT87]. For example, Woods' algorithm [Woo82] starts by determining a planar embedding (a drawing of the graph on the plane with no edge crossings). The graph theoretical \((s,t)\)-numbering technique is then used to assign vertices to levels within the graph, such that all vertices except those on the top and bottom levels are connected to vertices above and below them. A level by level scan then assigns vertex positions so that there are no edge crossings. Two problems with this algorithm are that the graph is spread over many levels, and strongly connected vertices are not kept close together. Batini's algorithm [BNT86] produces an improved result by minimising the bends in the edges, which tends to keep connected vertices closer together.

Undirected Graphs

Eades [Ead84] developed a method for drawing an arbitrary undirected graph based on the force directed placement [QB79] VLSI technique. In this method the graph is represented as a mechanical system, where the vertices are replaced by steel rings and the edges by springs. The rings are given an initial starting state and then "let go" so that the spring forces move the system to a state of minimal energy. Eades algorithm differs from physical reality in that it does not obey Hooke's law [Tip82] (a physical law that approximates spring behaviour). Another difference is that the repulsive forces are calculated between every pair of vertices, but attractive forces only calculated between neighbours (reflecting his idea that it is only important for a vertex to be near its immediate neighbours).

Kamada and Kawai [KK89] also modelled the graph as a system of springs, however they solved partial differential equations based on Hooke's law to minimise the total spring energy (the sum of the compression and tension forces). This
algorithm also improved on that of Eades by including the idea of an ideal distance between vertices which are not immediate neighbours, proportional to the shortest path between them.

A drawing algorithm for entity-relationship diagrams is presented in [RB85]. Here a divide and conquer technique is used to partition the graph and produce a reasonable layout quickly. Another algorithm for this type of graph is given in [BNT86].

Tamassia [TBB88] presented a graph theoretic approach that considers a wide range of aesthetic principles and constraints, characterising graphs by topology, metrics and shape. The algorithm can be used to lay out general undirected graphs as well as graphs containing hierarchic sub-structures, with vertices placed on a grid.

Davidson and Harrel [DH89] laid out graphs using simulated annealing [KGV83, OvG89], a powerful but computationally expensive optimisation technique commonly used in VLSI design. The problem is recast as one of minimising energy, defined in terms of vertex distribution, proximity of vertices to borders, edge lengths and edge crossings. These factors are weighted to emphasise different aesthetic properties. Although the graphs produced by this technique are of very high quality, simulated annealing is very slow and is therefore impractical for interactive applications.

A much faster technique for more general applications is presented by Fruchterman and Reingold [FR91]. This algorithm considers the problem of drawing the graph as an n-body problem [Ben86, Gre88], where the vertices are represented as atomic particles which attract and repel each other. Following Eades' method, attractive forces are exerted only by neighbouring vertices while all vertices repel each other, so that while connected vertices are drawn close to each other, vertices in general are not positioned too closely. Although this algorithm is designed to be fast, using integer arithmetic and computationally inexpensive calculations,
excellent results are produced at ‘interactive’ speed for moderate sized graphs. Variants of the algorithm are even capable of three dimensional layout projected onto two dimensions, with a choice of view reference point, normal vector and up vector combined with either parallel or perspective projection [FD84].

**Hierarchic Directed Graphs**

Directed graphs are commonly used to visualise ideas and relationships. Examples include Petri nets, syntax trees, call graphs, taxonomies, entity relationship diagrams, automata, transition networks, hardware interconnection networks, queuing networks, data flow diagrams and project scheduling charts. Directed graphs can be used to convey hierarchical relationships by arranging the vertices into *levels*. In a *k-layer* graph the vertex assignment to each level is fixed. A *k*-layer graph is *proper* if the edges only connect vertices in adjacent levels as shown in Figure 2–1. The aesthetic principles which are usually considered when drawing
a hierarchic graph are to minimise the number of edge crossings and total area, together with attempting to draw all the edges in the same direction so that it is easy to identify ancestor and descendant relationships.

Warfield [War77] presented a “crossing theory” for proper $k$-layer graphs which used heuristics to reduce the number of edge crossings. Di Battista and Nardelli [BN87] considered proper $k$-layer graphs with one source vertex, presenting a linear time algorithm that determines if a given graph of this type is planar (and a suitable layout if it is).

Johnson [Joh82a] showed that the problem of minimising the number of edge crossings is NP-complete, while Eades et al. [EMW85] showed that this is true even for bipartite graphs where the vertex positions on one level are fixed. Eades and Kelly [EK86] and Makinen [Mak89a] describe and compare some heuristics for reducing the number of edge crossings in a $k$-layer graph. In [EW86] and [Mak89b] the upper bound on the number of edge crossings for one of these heuristics is shown to be at most three times the minimum possible.

A different heuristic is to minimise the number of exceptions in the order of the vertices on each level. The successors of each vertex are assumed to be completely ordered and an exception occurs if two vertices require different orders for common successors. Unfortunately this approach is equivalent to finding the minimum feedback edge set and is also NP-complete [GJ79].

Warfield’s algorithm [War76] assigns graph vertices hierarchically to levels by placing the subset of vertices with no successors on the bottom level, removing this subset from the graph and repeating the process to find the next highest level until no vertices remain. Maximal cycles are collapsed into proxy vertices which include all of the successors and predecessors of the cycle vertices, producing an acyclic graph as a result. Carpano’s algorithm [Car80] expands these proxies into a three dimensional layout, while Meyer [Mey83] proposed the use of canonical forms to render proxies representing cycles of different sizes. In Davis’ method
CHAPTER 2. SUPPORT FOR DEBUGGING

[Dav85] cycles are temporarily eliminated by reversing the direction of selected edges.

In [Tut63] Tutte presented a sorting measure called a barycentre which measured the average planar position of the neighbours of a vertex. Delarche [Del79] used a one dimensional version of this measure which estimates the best horizontal position for each vertex by averaging that of its immediate ancestors and descendants, to alternately sort the top and bottom levels of a two-level hierarchy. As each level is sorted the opposite level’s barycentres are recalculated, and the process is repeated until a stable set of vertex positions is produced. In this way connected vertices are positioned vertically to produce edges that do not cross.

Carpano [Car80] and Sugiyama, Tagawa and Toda [STT81] generalised this procedure to reduce edge crossings in k-layer hierarchies, by reducing the k-layer graph to a series of two-level hierarchies, and generalising both the barycentre measure and the sorting procedure. The first transformation is achieved by introducing dummy vertices at intermediate levels, to break up long edges which span more than one level into a series of segments. The single barycentre measure is replaced by an up-barycentre (the average position of a vertex’s predecessors) and a down-barycentre (the average position of its successors). In [STT81] the levels are alternately sorted by their up-barycentre in the order 2, 3, ..., k and then by their down-barycentre in the order k – 1, k – 2, ..., 1. A further stage then straightens the edges and reduces their length. The up-down barycentre of Meyer and Davis [Mey83,Dav85] (the average position of the predecessors and successors of a vertex) was used in the GRAB interactive graph browser [RDM+87]. Gansner et al. [GNV88] presented an improved version of the vertex level assignment procedure and an exact solution to the edge length minimisation problem.

Other approaches to hierarchical graph drawing include Trickey’s DRAG graph drawing system [Tri88], Robins’ Lisp-based ISI Grapher [Rob87], May et al.’s circuit layout system [MIM81], and Majewski et al.’s schemata layout system [MKFA86]. Sugiyama and Misue [SM91] have extended the method to draw com-
CHAPTER 2. SUPPORT FOR DEBUGGING

pound directed graphs containing inclusion and adjacency edges. Here vertices are drawn as rectangles and inclusion edges are represented by inclusive relations among them, with adjacency edges drawn as arrows between the corresponding vertex pairs.

2.3.4 Graphical Abstraction

As the size of a graph increases, it becomes harder for a layout algorithm to produce a drawing which conforms to the required aesthetic principles due to the sheer number of vertices and edges that must be placed. Techniques such as zooming and scrolling, although sometimes useful for graph navigation, will not always be sufficient as it simply may not be possible to display all of the parts of the graph that a user may require to view on a finite-sized display. In these cases a mechanism to abstract away unnecessary detail is required.

One type of abstraction method is described in [TN87], where vertices are collected by the user into a nested hierarchy of groups. Here zooming out of a group has the effect of replacing it with an icon, while zooming in expands the icon so that the subgraph appears in its proper place within the part of the graph that is displayed.

Automatic support for graphical abstraction requires an automatic approach to partitioning the graph. Two types of partition have been investigated by Messinger et al. [MRH91]: 1) an application-specific partition based on the semantics of the input graph and 2) a graph-theoretic partition based on a syntactic partitioning algorithm such as that of Kernighan and Lin [KL70].

The primary advantage of a semantic partition is the semantic coherency of the subgraphs which result. Different partitions can be used to provide different views of the same graph, based on specified criteria. However this type of partition can lead to an unbalanced display and poor layout, with some subgraphs having many vertices and some having very few. It may take longer to lay out a set of
unbalanced subgraphs, and a large number of inter-subgraph edges may also be introduced.

A syntactic partition groups together sets of vertices that are more closely connected. Kernighan and Lin's algorithm [KL70] partitions a graph of \( kn \) vertices into \( k \) subsets of \( n \) vertices, with a minimal number of inter-subgraph edges. This method produces good results when a graph has no natural semantic partition. It can be used as part of a divide and conquer scheme which splits up a large graph into smaller subgraphs, lays them out individually and then composes them to create a total graph layout [Mes88, MRH91]. However, a syntactic partition may be misleading if a graph has a natural semantic partition, as it may imply some sort of false meaning to the user.

Another kind of semantic partition is the clustering technique. A clustering algorithm identifies a set of vertices which are more strongly connected to each other than to the rest of the graph. Parker-Rhoads and Needham [PRN60] described a “G-R clump” technique for database information retrieval. Spärk-Jones [SJ68] extended this work using four different types of cluster, namely strings, stars, cliques (complete subgraphs) and clumps. Further analyses of the detection of cliques are presented in [AM70, GM75, McD84, Jer90].

Other types of clustering techniques have been proposed for partitioning large systems into smaller systems [SVCC77, BE79, HB85]. Maarek and Keiser [MK88] and Selby and Basili [SB88] have summarised software engineering cross reference graphs by clustering vertices into strongly connected groups and analysing the edges between them. Hierarchical ascending classification algorithms [Maa89] form small clusters of strongly related vertices, and then combine these to form large clusters which eventually make up a classification tree. Vertex subdivision algorithms divide a set of vertices into several sets containing related vertices, and then recursively divide these sets.

A conceptual clustering method, such as that of the Cobweb system [Fis87], cre-
ates vertex clusters together with *descriptions* which explain the relation between the vertices. This type of clustering is used by the ARCH project [SAP89,SP89] to cluster groups of vertices which are related by *shared neighbours* in order to analyse software system structure.

Chapters 6 to 8 will describe a clustering technique for bipartite graphs which can be used to reduce graph complexity, simplify automatic rendering and highlight the sub-structures that may be present.
Chapter 3

Unassigned Variables

One of the most common programming errors is the use of an unassigned variable [KW90]. Many tools and techniques have been devised that help programmers to eliminate this type of error from their programs. Languages have been designed that eliminate the possibility of unassigned variable errors occurring. Techniques have been devised that help to formally verify programs and hence guarantee the correctness of their operation. Static analysis tools can be used to detect the flows of control that may give rise to unassigned variables. Finally, systems have been devised that can detect the attempted use of unassigned variables at run-time.

This chapter introduces the general problem of unassigned variables and the techniques that language designers have used to deal with it. The use of formal verification and static analysis to eliminate unassigned variable errors is discussed. Chapter 4 considers the problem of detecting the use of dangling references at run-time, presenting some novel methods for adding pointer run-time checks to a language whose definition does not support any checking strategy. Chapter 5 discusses the problems of unassigned composite objects and their impact on language design.
3.1 Introduction

A programming language may impose many different initialisation requirements when a variable is declared. It may define that:

1. no initial value must be provided
2. an optional initial value may be provided by the user
3. an initial value must be provided by the user
4. a default value is provided by the language, and no other value must be provided by the user or
5. a default value will be provided by the language if the user does not provide an initial value

In addition, the particular initialisation requirement may depend upon whether the variable has scalar, pointer or composite type.

A language may or may not admit the possibility of unassigned variables. It may be active or passive (does or does not cause an error to be signaled) when an attempt is made to use the value of an unassigned variable. It may also restrict the times when a variable can possess an unassigned value in three possible ways:

1. **Absent**: there is no notion of an unassigned value and so no object is ever unassigned.
2. **Initial**: an object is unassigned only when created and in the time before its first assignment.
3. **Dynamic**: an object may be unassigned at any time.

Again, these possibilities may depend upon the type of the variable.

The following sections will briefly introduce the strategies of required, default and no initialisation, and show how they are used in several popular programming languages.
3.1.1 Required Initialisation

Functional languages tend to require that an initial value is provided when a variable is defined. In Hope [Bai90] and ML [Wik87] it is not possible to introduce a variable without also providing its value. In VAL [AD79] an object (with the exception of an individual element of a vector) cannot be used before it has a value bound to it. Also, once an object is assigned a value it cannot be changed. The only way to modify an array or record is to construct a new composite object that has the same values in all of the previous positions, except for the particular components that must be changed.

An interesting variation on the theme is found in the Lucid language [WA85]. A Lucid program is usually composed as a set of nested clauses. The scope of a variable in a particular clause is either local (declared or defined in the same clause) or global (its value is expected to be obtained from the first outer clause in which that variable is local). If the variable is global to the outermost clause, its value is obtained from the user environment, and so the machine asks the user for the value as input.

Required initialisation of user-defined objects can be enforced in C++ [Str91], even though the normal language mechanism is to have no default initialisation and allow the user to specify an optional initial value. This is achieved by defining the object construction routines to take the initialisation values as parameters. Object declarations must then be accompanied by suitable values.

3.1.2 Default Initialisation

In most implementations of the Basic [Ten81] language, an object is created by the first reference to it and assigned a default zero value. In Snobol [GPP71] when a new variable is created it is given the null string as a default value. In Ada [Dep83] although scalar and composite types are not initialised by default, variables of
access type are initialised to the null 'pointer'. Simula [Poo86] automatically initialises boolean values to false, arithmetic quantities to zero and pointer types to none. C++ can support default initialisation of user defined types by defining the object constructor routines appropriately.

Default initialisation requires the implementation to carry out special operations when a variable is declared. In a language that supports only global variables, these operations can be carried out when the program is loaded. In a language which supports procedures and functions that may have local variables, some runtime overhead must be incurred to perform the initialisations.

3.1.3 No Initialisation

A large number of imperative programming languages do not provide any variable initialisation whatsoever. For example, Pascal [JW76] and many dialects of Fortran [AEPW81] do not initialise variables of any type. An access to an unassigned variable might consistently give a zero value or a "random" value, depending on the particular implementation. In Lisp [Ste90], every global variable not defined by the implementation is unbound unless it is explicitly assigned a value.

This method does not require the language implementation to carry out any special actions on behalf of the user, and is correspondingly the most economical to implement.

Summary

This section has enumerated the many options that a language definition can adopt when variables are declared. The strategies of required, default and no initialisation have been described together with examples of languages that use them. The next sections will discuss the problems of eliminating unassigned variable errors using formal verification, static analysis, and required or default initialisation.
CHAPTER 3. UNASSIGNED VARIABLES

The final section of the chapter introduces the problem of detecting unassigned variables at run-time.

3.2 Eliminating Unassigned Variable Errors using Formal Verification

In a formally verified program a variable will be *guaranteed* to be valid before its value is used. If formal verification is possible, a language could be designed to take this into account, and so rules and mechanisms for detecting the use of unassigned variables would not be required.

Formal proof techniques normally use mathematical logic to validate the consistency of an *output assertion* (a specification relating to the program’s output) with respect to the program and an *input assertion*. The program is “proved” if it can be shown that the output assertion is true whenever the input assertion is true, for a particular set of program data. It must also be shown that the program terminates to complete the verification.

Two of the approaches that have been taken in designing the necessary mathematical logics to be able to reason about programming languages are:

1. to use logic with a natural, separate notion of computation [Flo67].

2. to tightly integrate the mathematical aspects of programming languages with the computational aspects [CO78].

However the expressive power of most programming languages means that in general the verification logic is not decidable i.e. there is no algorithm to determine the truth or falsehood of every statement in the logic.

As formal mathematical techniques grow very rapidly in complexity, efficient heuristic based approaches become essential. Unfortunately these are not yet
sufficiently developed to be applicable to a large class of programs. In the absence of good heuristics, some verification tools require programmer interaction to help complete the proof [CO78, LGvH79, LP92].

Informal proof techniques follow the logic of formal techniques but without the formal details. This often makes them more amenable to programmers because they are intuitive and not overly mathematical. The types of proof range from simple checks, such as array bounds not being exceeded, to complex logic chains proving that shared data access protocols are correct. However it still requires considerable effort to verify even simple programs, and much work has yet to be done before the techniques will be applicable to very large systems of programs.

Symbolic Execution

Symbolic execution [HK76] is a less powerful technique that attempts to “dry-run” the program using symbolic values rather than specific data as input. It works by taking a program path and substituting all occurrences of computed variables with an expression written in terms of only the input values. Therefore all the conditions that determine the flow of control and the output values can be expressed as equations over the input values [Cow83]. In principle these equations can be solved and used to determine either the feasibility of the particular path or to generate a set of test data that exercises it. However in practice most such paths are infeasible and the equations are very complex to solve [HH85]. The choice of paths is very difficult when dynamically computed loops are involved. Symbolic execution can only generate expressions for the loop indices, which may have a wide range of values and unknown bounds. Other problems arise when operating system calls are used or finite arithmetic is analysed [HH89].
Summary

Proof of correctness techniques have a role to play in the validation and verification process but, with the current state of the art, other verification techniques are still required. Considerable progress has been made with the technique of static analysis, and the next section describes the use of static analysis to detect unassigned variables.

3.3 Eliminating Unassigned Variable Errors using Static Analysis

The term "static analysis" can describe a variety of processes, from manual code inspection to formal proof. The technique involves examining a program in depth to determine its function without executing it. Information relating to the overall structure and quality of the code can be determined, as well as detailed mathematical relationships between the program variables. The following sections discuss the use of flow analysis to determine the behaviour of a program. Subsequent sections describe two commercially available static analysis systems, SPADE and MALPAS, which use flow techniques together with semantic analysis to help verify programs against their specifications.

3.3.1 Flow Analysis

The technique of flow analysis is based on the graphical representation of programs. Firstly a directed graph is constructed that represents the program to be studied. The exact form of the graph depends on the particular technique that is used to analyse it. The three techniques of control-flow, data-flow and information-flow analysis are described below.
CHAPTER 3. UNASSIGNED VARIABLES

Control-Flow Analysis

In control flow analysis, each node of the graph represents a statement or program segment that possibly ends with a branch instruction. Graph edges represent the possible flows of control from one segment to another. The graph is used to analyse the program’s behaviour, locate possible breakpoints, detect unreachable code and identify statements from which no exit can be reached.

Data-Flow Analysis

In data-flow analysis the graph is similar but the nodes usually represent only a single program statement. By analysing the nodes to determine the behaviour of the program variables it is possible to discover anomalies such as unassigned or unreferenced variables. A variable’s behaviour at a node can be classified according to whether it is referenced (r), defined (d), unreferenced (u) or not involved (n). A variable is referenced when its value is used, defined when its value is set, and unreferenced when its value is no longer valid e.g. variables local to a procedure are unreferenced when the procedure exits. A path expression for a variable can be constructed by following a particular path of interest through the graph, yielding, for example, the string ‘dnrnnunnrdnnrnn’. After eliminating occurrences of not involved, a path expression allows the detection of anomalies in the obvious way e.g. the presence of the substring “..ur..” in the expression would indicate that a variable had been referenced when it had no value.

The DAVE system [OF76] is an early example of a static analysis tool that uses data flow analysis to detect the erroneous use of data in Fortran programs. It constructs a flow graph and then performs a depth first search for data flow anomalies in a time proportional to the number of graph edges multiplied by the number of program variables. The system’s primary functions are to detect attempts to reference variables before they have been assigned, and to detect assignment to variables that will subsequently never be used again. It is also
CHAPTER 3. UNASSIGNED VARIABLES

capable of detecting the use of exhausted DO loop indices, mismatched numbers of formal and actual procedure parameters and errors concerning failure to reference COMMON blocks.

Information-Flow Analysis

Bergeretti and Carre [BC85] have described the related technique of Information-Flow Analysis for a simple Pascal-related language. The method involves the successive construction of three binary relations for every program statement $S$:

- $\lambda_S$, from $V$ to $E$
- $\mu_S$, from $E$ to $V$ and
- $\rho_S$, from $V$ to $V$

where $V$ is the set of all program variables and $E$ is the set of all instances of the expression part of every assignment, conditional and repetitive statement in $S$.

A control-flow graph is created where each node represents a single assignment or expression part of a conditional or repetitive statement, for every program statement in $S$. Each edge represents a possible transfer of control. For any variable $v \in V$ and any expression $e \in E$, the condition $v\lambda_S e$ is informally interpreted as “the value of $v$ on entry to $S$ may be used in evaluating the expression $e$ in $S$”. The condition $e\mu_S v$ signifies that “a value of the expression $e$ in $S$ may be used in obtaining the value of the variable $v$ on exit from $S$”. The final condition $v\rho_S v'$ can be interpreted as “the value of $v$ on entry to $S$ may be used in obtaining the value of $v'$ on exit from $S$”. This means that either:

1. the entry value of $v$ may be used in obtaining the value of an expression $e$ in $S$, which in turn may be used in obtaining the exit value of $v'$, or
2. $v = v'$ and $S$ may preserve $v$. 
Relation construction rules can be derived for assignment, compound, conditional and repetitive statements, together with procedure and function calls [BC85]. The flow relations for an entire program can be derived by repeated application of these rules. Tests can then be performed on the relations to detect programming errors such as ineffective statements, unused parameters and unassigned variables. They can also be used to help the programmer understand the program text by automatically extracting “partial programs” consisting of the statements that affect a particular variable at any point of interest.

Expressions whose evaluation may involve the use of an unassigned variable can be detected as follows. Let $S$ be the statement part of a program or subprogram $P$, and let $V^i$ be the set of variables whose initial values are imported by $P$. For any variable $v \notin V^i$, if $v \lambda_se$ for some $e$ in $S$ then the unassigned variable $v$ may be used in the evaluation of $e$.

Alternatively, expressions that refer directly to an unassigned variable can be detected by defining a relation $\theta_S$ from $V$ to $E$, where $v \theta_se$ iff $v$ appears in $e$ and there is a path in the control flow graph from the entry point to the node associated with $e$ that does not traverse any assignments to $v$. Another useful relation is $\hat{\theta}_S$ from $V$ to $E$, where $v \hat{\theta}_se$ iff $v$ appears in $e$ and every path in the control flow graph from the entry point to the node associated with $e$ does not traverse any assignments to $v$. These two relations can be easily calculated at the same time as the $\lambda_S$, $\mu_S$ and $\rho_S$ relations using an appropriate set of rules [BC85].

A problem with the method is that not all of the paths of a program may be executable, and so neither of the conditions $v \lambda_se$ or $v \theta_se$ imply that in any program execution the entry value of $v$ is necessarily used to evaluate $e$. It may only be possible to determine if this is the case by considering the program semantics.

The information given by the $\lambda_S$, $\mu_S$ and $\theta_S$ relations is similar to that given by data flow analysis. However the $\lambda$-relation provides information that is not obtainable via data flow methods. In addition, the $\mu$-relation is a simpler and
more effective method for detecting unreachable code as it directly shows all the statements where execution cannot affect the final values of variables that are ‘live’ on exit from a subprogram.

3.3.2 SPADE

The Southampton Program Analysis and Development Environment [Car87] contains a number of program analysis and verification tools for the development of high-integrity software. SPADE defines a “safe subset” of the target programming language by eliminating features that are not essential but which cause ambiguities and make formal verification very difficult. Annotations or formal comments are added to the language to help resolve ambiguity and allow the specification of a procedure to be contained within its code. Subsets of programming languages such as Ada, Pascal, Cobol 85 and MC68020 assembler are supported.

Firstly a program is translated into SPADE’s Functional Description Language (FDL), a language for describing a program in terms of its states and state transitions. The toolset then operates on this program model. The control-flow analyser identifies unreachable code, multiple-entry loops and statements from which no exit can be reached. The data-flow analyser identifies unused definitions, redundant tests, loop-invariant assignments and uses of unassigned variables. The information-flow analyser detects ineffective statements, unused parameters and global variables, uses of unassigned variables, inconsistencies in import-export relations and loop stability. Further tools such as the verification condition generator, the symbolic interpreter, and the proof checker can be used to perform extensive semantic analysis of the program. Facilities are provided to allow the extraction of a partial program consisting of only the statements that involve specific variables of interest. Also, an algebraic simplifier can be used to simplify the mathematical formulae presented to the user.
Brendish [Bre87] has shown that FDL models can be produced which re-express the program in terms of its high level design. In this way the actions of the original code can be replaced by English-like statements such as AUTHORISED_USER or LOGGED_ON. Formal semantic analysis then produces a “pidgin” English result that is easier to understand, and hence more assurance can be gained that the program meets its control objectives.

**SPARK - the SPADE Ada Kernel**

SPARK is a subset of Ada designed for safety-critical software. It contains the features of the language that are needed for rigorous program development, but has simple semantics so that FDL modelling and analysis with SPADE is practical. For example, SPARK does not allow the use of tasks, exceptions, generic units and access types. Default procedure parameters and initial values for variables are not permitted, and all ranges must be statically determinable.

### 3.3.3 MALPAS

Another tool that is widely used for verifying safety-critical software is the Malvern Program Analysis Suite. In order to use MALPAS, a program must first be translated into an intermediate language called MALPAS IL. The ability to translate a given language into IL depends heavily on whether IL’s model of computation is close enough to that implied by the language. Automatic translators for many common languages including Ada, C, Pascal and Fortran have been produced. In contrast with SPARK, there is an almost complete mapping from Ada to IL. Representation clauses and some tasking constructs are not translated although they are still accepted by the translator, which issues a warning. This allows users to include their own models of the excluded language features if necessary [War91]. Only the sequential parts of each task can be translated to IL – the dynamic and concurrent behaviour must be verified by other tools.
CHAPTER 3. UNASSIGNED VARIABLES

The IL model is used to construct a directed graph with associated algebras for use by the MALPAS analysers. The control-flow analyser attempts to find poorly structured or unreachable code and endless loops. The data-flow analyser classifies the program's variables according to how they are used, revealing errors such as variables written and subsequently never used, writing to procedure parameters specified as input-only, and the use of unassigned variables. The information-flow analyser identifies the input variables and parameters upon which each output parameter of a subprogram depends. This can reveal unexpected dependencies, redundant statements and unused variables. The flow information can also be used to help design dynamic tests. The semantic analyser reveals the algebraic relationships between input and output variables for every possible path through a program section. It therefore reveals what the code actually does under all circumstances. The compliance analyser attempts to verify that a program meets its specification using preconditions, postconditions and assertions that have been inserted into the code as annotations. An algebraic simplifier can be used to help reduce the complexity of the expressions that are produced. In addition the partial programmer can be used to construct partial programs for selected variables. Conditional statements that are not involved in a particular analysis can be excluded, which reduces the number of program paths that must be investigated.

3.3.4 Conclusions and Problems

Static analysis is a very useful method for detecting errors in programs in general, and specifically for detecting simple unassigned variable errors. Good compilers sometimes perform a limited degree of static analysis for these types of errors. However there are many problems and limitations to the technique [HF90], some of which are apparent in SPADE and MALPAS.

1. Syntactic methods of analysis such as the data-flow and information-flow methods tend to be too conservative, as they will usually indicate possible
problems that are precluded by program semantics. The methods assume that all the paths through a program are executable, and as a result large numbers of "false warnings" may be produced.

2. Semantic analysis of large program fragments containing numerous possible paths can produce large amounts of complicated output. Although both systems include tools to help reduce the mathematical complexity of the results, skilled interpretation is often necessary. The structure of the software can influence the ease of this task. Modular software often produces an analysis that is easier to understand [O'N88].

The language used can also influence the analysis as languages often tend to have their own constructs and programming style. The automatic translation of an arbitrary program could produce an intermediate representation and subsequent flow graph that might be difficult to analyse, or give results that might be difficult to understand [O'N88].

Sometimes the result of the analysis can be no more meaningful or intuitive than the original program, or may be substantially longer than the original text [O'N88]. An experienced programmer may find it less time consuming to analyse the original program text, while an experienced analyst might prefer to examine the equations produced by the static analysis.

3. The analysis of composite types tends to be problematical. For example in MALPAS IL they are not directly supported and a functional model of array and record access must be generated [PW90]. Thus a modification of a component of a composite object is treated as a function call that takes the old object and the new component value as parameters and returns a new object.

4. Aliasing, where a single item of data is referred to by two or more names, is often difficult to model statically [War91].
5. In general, array indices are computed and it is therefore difficult to statically analyse array access [HF90]. Static determination of the range of possible values for a computed array index may not yield any useful information, or may produce a complex result that depends too heavily on the analyst's skill to interpret.

6. Dynamically determined pointer variables are extremely difficult to analyse [HF90, War91]. Also, it can be impossible to statically determine the memory requirements of programs that create dynamically allocated data structures. For these reasons pointer types are sometimes held to be inappropriate for safety-critical software [Car87, Wel91]. However in some instances the analysis may be able to reduce the possibilities to a small set e.g. pointers to procedures can only point to the members of the set of procedures.

7. It is very difficult to statically resolve computed loop indices and hence any unfolding of a loop with computed indices may be entirely arbitrary [HH89].

8. The anomalies detected by static analysis require a human *oracle* to determine whether an error exists. Anomalies that give rise to functionally-equivalent programs are very hard to resolve, even with very sophisticated theorem provers [HF90].

9. The technique seems to produce the best results when a program is designed from the outset to be amenable to static analysis [PW90] by, for example, avoiding language features that are difficult to analyse. This strategy should result in an increase in the quality of the software and reduce development costs by detecting errors at the earliest possible stage. However it suggests that the use of static analysis on an “off-the-shelf” program might produce less than optimal results.

Although there are many problems with interpreting the results of static analysis, it is a useful technique for detecting simple unassigned variable errors. In
addition it can suggest the presence of faults of this kind under conditions that might appear only very rarely, and hence would probably escape dynamic testing. For safety-critical software, the use of static analysis is therefore essential. However the limitations of the method suggest that, although it is a useful tool, the best way to eliminate errors from programs is to combine static and dynamic techniques [HF90]. Dynamic analysis is usually driven by static analysis. A good static determination of the possibilities of unassigned variable errors should both allow the immediate elimination of some errors and also lead to a better set of dynamic tests.

3.4 Eliminating Unassigned Variable Errors using Required or Default Initialisation

If a program can be formally verified to be correct, it does not matter (for error checking purposes) if the language definition admits the possibility of unassigned variable errors, as none will occur. Here required or default initialisation will not help to prevent errors of this type. The initialisation simply sets the first value of the program variables, which may or may not be a useful function depending on their subsequent use. For example, it would be inefficient to create an integer variable, initialise it to zero and then immediately read its value from a file. Static analysis techniques might be used to determine which initialisations are actually necessary and hence allow the overhead to be reduced.

If a program cannot be verified, required or default initialisation might appear to be useful techniques for eliminating the possibility of unassigned variable errors. However, there are some problems with these approaches.
CHAPTER 3. UNASSIGNED VARIABLES

Procedure parameters

If a procedure parameter is of \textit{(in out)}-type, it must be created and assigned a value \textit{before} the procedure is called. This may actually reduce the ability of the system to trap errors in the same manner as shown in [KW90]. Consider the example program in Figure 3–1. The value of variable P2 is only set if P1 is greater than or equal to zero. Procedure CALCULATE only increments P2 if P1 is greater than or equal to zero. Now, assume that P1 and P2 have been initialised to zero, and the program is run. If a later part of the program was to erroneously use P2 when P1 was less than zero, the "false" initialisation of P2 would prevent the possibility of an unassigned variable error being signaled. Although passing a parameter which is never used is inefficient, it avoids having to generate extra code to check the value of P1, so that a version of the CALCULATE procedure with the appropriate number of parameters can be called.

If a procedure parameter is \textit{out}-type, it must also be created and assigned a value \textit{before} the procedure is called. This should \textit{never} be a useful function, and can again reduce the ability of the system to detect errors. If there are no semantic checks to catch attempts to use the value of an \textit{out}-type parameter in the procedure, the required or default initialisation may in fact provide a suitable value, and an unassigned variable error cannot be signaled.

Composite types

A similar argument applies when arrays or record types are considered, since the elements of a composite type must be assigned a value when the composite is created. It may not be efficient to repeatedly allocate storage for a new structure containing only those components that are assigned, every time a value for a previously unassigned component becomes available.
Program calc (input, output);

Var P1, P2 : integer;

Procedure CALCULATE (Var P1, P2 : integer);

begin
  if P1 >= 0 then P2 := P2 + 1;
  P1 := P1 + 1
end;

begin
  readln(P1);
  if P1 >=0 then P2 := 3;
  CALCULATE(P1,P2)
end.

Figure 3-1: Example program
User-defined types

Assigning an arbitrary default value to variables which should, strictly speaking, have no value may also be inconvenient. Sometimes it may be difficult to find a logical reason for choosing a suitable default value. For example, it is not obvious what the default value of a variable with type definition:

\[
\text{type } \text{paint\_colour} = \{ \text{green, blue, orange, brown} \}
\]

should be.

Pointer types

Ada initialises access types with the null pointer value by default. This is a good way of catching attempts to use access types if they have not been assigned to reference an object, as the language definition requires that a check for the null value be made anyway, raising a CONSTRAINT\_ERROR exception if necessary. Therefore the detection of an attempt to use an as yet unassigned access type does not impose any extra run-time overhead other than that required to satisfy the language definition.

This default initialisation can reduce the possibility of error, but cannot eliminate it. For example, two access types \( p_1 \) and \( p_2 \) may be created and assigned to reference an object \( \text{obj} \). If \( \text{obj} \) is now deallocated using the construct \( \text{Free}(p_1) \), where procedure \( \text{Free} \) is obtained by suitable instantiation of the generic procedure UNCHECKED\_DEALLOCATION, although \( p_1 \) is now set to null, \( p_2 \) will still reference the storage that used to be occupied by \( \text{obj} \). Logically, \( p_2 \) should also be set to null, but this would require additional, more complex mechanisms. Techniques that can be used to achieve this will be presented in Chapter 4.
Portability

Default initialisation can simplify the programmer's task if the initialisation values are well chosen [Mey90]. However, if a program is translated into another language that does not support default initialisation, extra code must be added to perform the task. The omission of this code is a common source of error. The problem can also arise in porting programs from one machine to another when default initialisation is a language implementation option. Many C programmers have come to depend on default initialisation to zero under UNIX. With heap objects, this only applies to the calloc memory allocation function, not malloc. A classic example [Poo93] of the problems of relying on default initialisation when it is not guaranteed is to be found in the history of development of the Pascal I/O run-time system for the ICL PNX Perq, which was implemented in C. When an attempt was made to open a file, a check was performed to ensure that it was not open already. This involved testing a pointer field in the file descriptor, on the (false) assumption that each time a new descriptor was created using malloc it would be filled with zeros. Spurious “File is already open” errors resulted when sufficient dynamic data operations were executed to require the system to reallocate previously freed memory for use as file descriptors. Although a simple test program had been isolated (which opened and closed files in a loop until the behaviour was shown) the error defeated the programmers who were maintaining the implementation for over three years.

Conclusions

Required and default initialisation can be useful for preventing unassigned variable errors. In languages that support procedures and functions with local variables, the latter strategy requires the implementation to incur initialisation overhead at run-time. Several problems concerning procedure parameters, composite types, user defined types, pointer types and portability have been described. In partic-
ular, it has been shown how the initialisations can sometimes reduce the system's ability to detect errors.

Even with guaranteed initialisation, it would still be possible for unassigned variable errors to occur if the language allowed variables to dynamically change assigned status. If one of the language design goals was to eliminate the possibility of unassigned variable errors it would preclude this language option.

A reasonable case can be made for a language to only allow the declaration of a scalar variable at the point at which a value is ready to be assigned to it. In this way language scope rules can prevent the use of the variable in code that occurs before its declaration. The default initialisation of pointer variables with a null value has also been shown to be useful. Errors concerning pointers are notoriously difficult to trace and so the overhead of performing this initialisation may be very worthwhile.

3.5 Detecting Unassigned Variable Errors in Languages with No Initialisation

A language that does not support required or default initialisation must have a different strategy for dealing with unassigned variables. Two possible approaches are:

1. to simply state in the language definition that the use of an unassigned variable will have an undefined result or

2. to track the assigned state of all the program variables and signal an error if an attempt is made to use the value of an unassigned variable.

The first option requires the least amount of implementation overhead and provides the least amount of protection and help to the programmer. Ada and
some versions of Fortran use this approach for scalar variables. In Ada the detection of attempts to use the value of unassigned scalar variables is optional for the implementation. If the error is detected, the PROGRAM_ERROR exception should be raised. If it is not detected the language standard defines that the program is erroneous and has no meaning from that point onwards. Sometimes program faults can be hidden in such a way that it is not possible to detect whether the program is erroneous. For example, if a program uses an uninitialised counter that coincidentally happens to contain zero during the testing process, a program fault may not be revealed. However if at some later point the program is executed and the counter happens by chance to contain a non-zero value, it may cause an error.

Modula 2 [Wir82] uses the second approach for scalar variables. With composite objects, the assigned status of a component scalar variable is copied when the composite as a whole is copied, but is not checked until the value of the component is accessed.

Standard Pascal [JW76] defines that any access to the value of an unassigned variable is an error, and specifies the situations in which a variable becomes undefined:

1. At the start of the execution of a program, procedure or function, all local variables become undefined.

2. In variant records, all the fields of a variant part become undefined when another variant becomes activated.

3. A file buffer is undefined before the corresponding file is opened, and after each buffer has been written.

4. A loop control variable becomes undefined when the loop is exited.
CHAPTER 3. UNASSIGNED VARIABLES

5. The heap storage referred to by a pointer becomes undefined when first allocated.

6. A pointer becomes undefined when the heap storage to which it refers is deallocated (as do all copies of the pointer).

In addition, a composite object in Pascal is assigned only when all of its components are assigned. Therefore the programmer must define all of the components of a composite before an operation on the whole structure can take place. The problems of composite objects and the difficulties inherent in this particular approach are considered later in Chapter 5.

In Pascal, run-time checks are considered as an issue of quality, and a compiler can claim conformance to the language standard even if it does not detect unassigned variables [KW90]. Although run-time checks are highly desirable, they are usually not implemented due to “prohibitive” run-time overhead [WH83]. However there are usually many occasions during the development of a program when even a very large overhead may be acceptable in order to help detect a particularly elusive problem. Several Pascal environments have implemented effective run-time checking strategies [FL77b,WH86,Kem89]. A common strategy is to provide a compiler directive that disables run-time checks. Checking is typically left active during program development to help detect and locate errors, and then switched off in production code. During development, static analysis can sometimes reduce the overhead by detecting which variables can never be unassigned when referenced and hence do not need run-time checks.

Given that a language design admits the possibility of variables having an unassigned value, the next sections consider how unassigned can be represented by the implementation. Subsequent sections describe the uses of unassigned variables and variables that are never unassigned. The issues of dynamic valuation and propagation of unassigned are discussed. Finally, methods for adding more useful
CHAPTER 3. UNASSIGNED VARIABLES

ways of coping with unassigned variables to languages that ordinarily offer the programmer no support are considered.

3.5.1 Physical representation of Unassigned

A language design that supports the detection of unassigned objects requires the run-time system to actively maintain an explicit, physical representation for the variable status of unassigned. This section will consider the factors and choices involved in this representation. The four possible implementation methods given in [KW90] will be summarised and extended.

Undefined variables have special values

This method attempts to utilise the available memory resources in the most efficient way by encoding the representation of the unassigned value as a special value of the data type e.g. on 16 bit hardware an integer might be defined on the range -32767 to 32767 leaving -32768 as the representation for unassigned. The definition of some types may require a specific representation for unassigned values e.g. the IEEE 754-1985 standard for binary floating point arithmetic specifies reserved values for unassigned real numbers. Alternatively a value that is “unlikely” to occur naturally might be used e.g. the EPC Pascal compiler [EPC91] uses a large negative value, 8181818116, to represent an unassigned integer on 32 bit hardware. A more extreme range restriction may be used in order to make room for the special value e.g. an 8 bit character data type might be restricted to 128 characters (the standard ASCII set) leaving the remaining bit to mark unassigned values. In the approach used by the Model Pascal implementation [WH86] the range of the data type is extended by one value (the unassigned value), and so in the worst case the physical representation must be enlarged by 1 bit. For many types e.g. subranges there may be no effect on the size of the physical representation, whereas for boolean values the representation may double in size.
CHAPTER 3. UNASSIGNED VARIABLES

When the representation does not have to be enlarged the special values method should be the most resource efficient, requiring no extra physical memory and only a single data fetch. However variable initialisation and unassigned status checking become more complex, since either the type of the variable must be known by the run-time system in order to be able to recognise the special unassigned value, or else the unassigned checking and initialisation code must be tailored by the compiler to suit each type. This difficulty sometimes means that an implementation will only try to detect an unassigned value when an attempt is made to use it rather than when it is transferred from one variable to another (in assignment or parameter passing). This is an unfortunate compromise since the use of an unassigned value is often preceded by a transfer, which would be a better and more useful place to trap the error.

If the representation does have to be enlarged across a memory word boundary then fetching the data becomes more complex. Run time efficiency considerations may lead to compromises in the use of storage space, as it will usually be simpler and faster to leave the remaining bits of the extra memory word unused, in order to avoid costly bit manipulation to reconstruct a directly usable data value from a 'packed' representation.

Duplicate Memory Bit-Map

In this method each word of global, stack or heap data has an associated word in a map of unassigned data. Each program operation that checks or changes the unassigned status of a variable refers to or updates this map. Each variable fetch generates an equivalent fetch from the unassigned map. Therefore if relative addressing is always used, the addressing calculation for the second fetch is greatly simplified by merely taking the appropriate base and adding the same offset. Modern hardware architectures often directly support relative addressing and so this calculation is very simple.
The granularity of a subset of the map reflects the corresponding size of the data that is associated with it. For example, a bit-level sub-map would be used for a packed array of bits and a word-level sub-map for a record of word sized data. Therefore half of the data memory must be allocated for the map. However the method is still very attractive due to its overall simplicity combined with the lower cost and larger size of modern real and virtual memory.

Memory represented as a Word-Map

If space is at a premium and there are no individual data items smaller than the minimal addressable memory size (one byte), a map where a single bit represents the undefined status of each byte may be used in the same manner as the previous method. This typically means that the map will only be one eighth of the size of the program data. However the method involves much more complex address calculations with bit masking, and makes it impossible to pack data structures more than the chosen minimum size.

Special Hardware

A hardware implementation of one of the above methods should reduce or eliminate the run-time overhead, by checking the undefined status of variables in parallel with normal program activity. The duplicate memory bit-map method would probably be preferred for its simplicity. However there are considerable difficulties with transparently supporting unassigned checking for data types that occupy less than one word of memory, since both the word containing the data and the corresponding word in the duplicate memory map must be fetched, bit-masked and tested. Ultimately a super-scalar architecture may be required to achieve maximum run-time efficiency.
3.5.2 The Run-time Functions of Unassigned Variables

An unassigned variable can be used as:

1. a placeholder
2. an error detection mechanism or
3. a mechanism for dictating flow of control

Each of these functions will be considered in turn.

Placeholders

An unassigned variable is a placeholder; it reserves space for a value that may be assigned later. If the variable is part of a composite object, placeholders allow the total storage space for the composite to be efficiently allocated in a single operation. They can also give the user flexibility in that they allow a variable to be assigned at a convenient point in the program's execution, rather than at declaration.

Error Detection

An attempt to use the value of a placeholder is clearly erroneous. Therefore if the language requires that there are suitable mechanisms to detect the attempted use of the value of a placeholder at run-time, they can reveal logical programming errors. This is only a useful function for programs that have not been formally verified.

Flow of Control

Winner [Win84] argued that it is often reasonable to ask whether an object has no value, and proposed language constructs such as an unassigned choice in case
statements, and status tests such as assigned() and unassigned() boolean functions. These constructs require the run-time system to track the assigned status of every program variable. There must be an internal representation of the unassigned value for every data type, so that there is a way to distinguish unassigned objects from assigned objects.

If a program makes use of the assigned state of its data in order to control its execution, it will not be possible to disable the state tracking mechanisms in production versions of the code. This ability may be very important [KW90] as run-time checking in production code may impose an unacceptable overhead.

In general, languages have not tended to support the capability of using the assigned state of the program’s data to control its execution, which may be due to the difficulties and performance costs associated with doing so.

3.5.3 Variables that are Never Unassigned

Some data types, for example memory mapped I/O devices, may never be unassigned i.e. they always have a value, or they may be ‘write only’. Occasionally a user may require a type that is never unassigned [Win84]. The Pascal type definition could be extended as shown in the following examples to allow this:

```pascal
type bit = never unassigned 0..1;
type io_read = never unassigned 0..255;
```

An interesting problem concerning never unassigned types occurs when the particular hardware architecture detects an unassigned variable using the special values method. The hardware trap must be disabled for these variables, or in cases where this is impossible the trap code must be able to determine that the error can be ignored in this case. Winner [Win84] states that placing a value different from the special unassigned value in uninitialised never unassigned variables would be
an appropriate alternative action. However this is clearly not the case since the variable might receive the special unassigned value by accident or by malicious intent during any stage of its existence. Another approach would be to define that it is an error to use an uninitialised never unassigned type, and that any such use will have an unpredictable result.

3.5.4 The Dynamic Valuation and Propagation of Unassigned Variables

If a language allows a variable's assigned status to change dynamically there must be a mechanism to give the variable the special value unassigned. Winner [Win84] defined a destructive assignment procedure ERASE(in out v), that changes the status of a variable v to unassigned or raises an exception if v is already unassigned. Using this procedure to represent the operation of removing a variable's value, he presented four conditions under which the unassigned value might be propagated:

1. Strong antipropagation: unassigned may never be passed from one variable to another. No function may return unassigned, no output parameter may return unassigned and no input parameter may import unassigned.

2. Escapable antipropagation: Like (1), keeping the assignment rule but relaxing parameter rules to allow escapes.

3. Weak antipropagation: Like (1), but allowing parameters and functions to pass unassigned.

4. Propagation permissive: unassigned may be copied.

The definition of the ERASE procedure is inconsistent with strong antipropagation, since the parameter always returns the value unassigned. Weak antipropagation may introduce problems when the result of a function is used in an arithmetic operation, for example:
Here if \( \text{func} \) returns an \textit{unassigned} result then the value of the expression is undefined. The propagation permissive regime gives no protection from errors concerning unassigned variables and so is inconsistent with Proposition 2.2.

As shown in Section 3.1 the ability to pass an unassigned variable as a function parameter may be useful. In addition escapable antipropagation is consistent with both Propositions 2.1 and 2.2, giving flexible protection for scalar objects, and is therefore Winner's recommended regime for a dynamic-active language.

3.5.5 Adding Language Support for the Detection of Unassigned Variables

Language designs where the notion of unassigned variables is absent, or the problem is effectively ignored, are acknowledged to be highly problematical [KW90, Mey90]. Several attempts have been made to add more useful semantics and facilities to languages that adopt this approach, with varying degrees of success. This section will describe some of the tools and techniques that have been implemented for the C [KR88] and Ada languages.

Improving Unassigned Variable Detection in C

The simple semantics of the C language provide little direct help to the programmer in detecting unassigned variables. Sometimes extremely simple errors can be detected using the standard C static analyser \textit{lint} [Joh78]. However this tool is unable to properly analyse composite objects and pointer types, and can generate very large numbers of false warnings (a result of one warning per line of code is common). This may be a reason why few experienced C programmers make use of it [Ste92].
The Robust C [FYP93] system extends the C language using a replacement for the standard preprocessor that translates the extensions to standard C. Facilities are provided to detect array bounds violations, ignored library function error conditions, incorrect looping constructs, range violations and attempts to use unassigned array elements. Two methods are provided to solve the latter problem. In the general case, one extra bit is allocated for each array element which indicates whether or not it has been defined. The extra bits are cleared when the array is allocated, set when an element is defined and checked when an element is referenced. The second method, a variant of the 'special values' technique, can be used if a value that is out of range for the data type can be found. The programmer can specify the out of range value when the array is declared, and the preprocessor generates code to initialise the array with this value. When an element is referenced it is compared with the special value, and an error signaled if appropriate. The programmer is also allowed to directly assign an element with the value. This can be used to give array elements a dynamic assigned status.

Improving Unassigned Variable Detection in Ada

Borie et. al. [BPM93] have shown how default initialisation and unassigned variable detection can be added to Ada programs using abstract data types. The same basic algorithm is used for both these facilities: a reference to an unassigned variable either returns the default value for variables of that type or signals an error.

The algorithm is based upon Fischer and LeBlanc's method for detecting dangling references [FL77a], which will be described in Chapter 4. A lock and a key are associated with every variable that could be unassigned when referenced (i.e. every variable not eliminated using static analysis). When a variable is defined the lock is set to match the key. If the lock and key do not match, the variable is unassigned and either a default value can be returned or an error can be generated.
A variable's key is its address in memory. The lock is an extra memory cell associated with the variable. It holds a pointer to an element of an array which holds the actual lock values. The elements in this LockValues array are allocated dynamically and sequentially. A variable fence partitions the array into its allocated and unallocated portions. If a variable \( m \) has been defined, \( m.\text{Lock} \) must point to an array element in the allocated part of LockValues, and \( \text{LockValues}[m.\text{Lock}] \) must match the key for \( m \), its address \( m.\text{Key} \). Figure 3–2 shows an example of an unassigned variable \( m \) and Figure 3–3 shows the data structures after \( m \) has been assigned. Figure 3–4 presents the algorithms for the major system operations.

The algorithms ensure that there is a constant start-up time. There is no \textit{a priori} initialisation of the LockValues array, nor of the user's variables and their associated locks. The rather complex arrangement of pointers to elements of the LockValues array ensures that a lock and key cannot initially match coincidentally.

Figure 3–5 compares the execution times of the standard methods for providing default initialisation and detection of unassigned variables with that of the lock and key approach. \( V \) is the size of the variable space for a given program. \( D \) and \( R \) are the total number of variable assignments and references during a particular execution respectively. The \( O(D+R) \) terms hide a larger constant \( k \) in proportion
Initialise:

\[ \text{fence} = 0. \]

Define \( m \):

\[
\text{if (m.Lock} \leq 0) \text{ or (m.Lock} > \text{fence}) \text{ or (LockValues}[m. \text{Lock}] \neq m. \text{Key}) \text{ then}
\]
\[
\{ \text{fence} = \text{fence} + 1
\]
\[
\text{m.Lock} = \text{fence}
\]
\[
\text{LockValues}[m. \text{Lock}] = m. \text{Key}
\]
\[
\} \]
\[
\text{Assign the intended value to } m.
\]

Reference \( m \):

\[
\text{if (m.Lock} \leq 0) \text{ or (m.Lock} > \text{fence}) \text{ or (LockValues}[m. \text{Lock}] \neq m. \text{Key}) \text{ then}
\]
\[
\text{return the default value or signal an error as appropriate.}
\]
\[
\text{else}
\]
\[
\text{return } m.
\]

**Figure 3-3: After the assignment \( m = 22 \)**

**Figure 3-4: Lock and Key algorithms**
to the constant c associated with the $O(V)$ term in the standard methods. The lock and key approach will perform better whenever $k(D + R) < cV$. A typical example would be where a static array is used to hold input data of variable size, and the worst case number of elements is much less than the average case.

The approach has been used to add default initialisation and unassigned variable checking to array types in Ada, using the package mechanism. A package exports suitable Define and Reference operations e.g. Define(n,4,5) is called to assign the value 5 to the fourth element of array n. Therefore the programmer need only make use of the technique in cases where it is useful.

Further experimental validation is required to accurately determine the types of programs for which the technique is suitable [BPM93]. The syntax of the programs that use the package is clearly very different from normal, which may make it difficult to add and remove the facility as dictated by run-time performance requirements.

**Summary**

The active support of unassigned scalar objects leads to some language design choices. Several methods have been described for representing the unassigned value and the problems of dynamic valuation and propagation of this value have
been discussed. Methods for adding more useful run-time checking strategies to the C and Ada languages have been considered.

The problems of unassigned pointer variables are more complex, since the status of a pointer is not simply dependent upon its own value but also depends upon the status of the storage area that it references. These issues are described in the next chapter.
Chapter 4

Tracking Pointer Status at Run-Time

Introduction

The classic dangling reference problem, where a pointer references storage space that has been deallocated, is usually very difficult to detect and trace. Some languages such as Pascal try to reduce the potential for pointer errors by only allowing pointers to heap objects to be created. A heap object $\textbf{obj}$ is allocated using the construct $\texttt{new(p)}$, where $p$ is of type pointer to $\textbf{obj}$, and $p$ is assigned to reference it. If $\textbf{obj}$ is a composite object, it is not possible to create a pointer to one of its components. Other languages such as C allow the creation of pointers to heap \textit{and} stack allocated objects. There are no restrictions on the creation of pointers to components of composite objects, and arithmetic can even be directly performed on the addresses of variables.

Language run-time systems commonly do not provide any protection against dangling reference errors due to the overhead that may be incurred by doing so. It is also possible to argue [WSH81] that the pointer is a low level feature designed
for use in situations where high level data constructs are not sufficiently expressive, and so it is not reasonable to expect security from such a low level facility. While this argument has some validity, it is not always true. In particular, a “safe” pointer system is often desirable during program development, at times when the associated run-time overhead is not a significant factor. In some instances it may be worthwhile to pay even a high performance penalty in order to ensure reliability or to trace a particularly elusive error.

When suitable tools for formal verification of pointer operations are not available, or when they are inadequate for the required task, run-time checking may be the only available option. The following section will outline some of the technical problems of detecting dangling references at run-time. Three run-time system designs that perform the task are presented. Subsequent sections will describe novel implementations of two of these methods for a language whose run-time system does not support any checking strategy.

4.1 Dangling Reference Detection

A language definition that allows the explicit destruction of objects under direct programmer control can potentially lead to the creation of dangling references. This issue is particularly important for language design and implementation as the use of a dangling pointer (or indeed an uninitialised or null pointer) can easily lead to catastrophic program failure [FL80]. Wirth [Wir76] noted that nothing less than an automatic garbage collector would be required in order to implement a secure version of the PASCAL object deallocation procedure dispose, and therefore correctly deal with any dangling references that might be created. A garbage collector can identify a dangling reference $dp$ if it points to an object on the free list. However if the particular free list object space is then reallocated to a new object of the same type as the original, although $dp$ is not, technically, a
dangling reference it would be an error to use it if, as would usually be the case, the programmer intended it to still reference the original object.

A pointer validity check performed in software will typically incur an overhead greater than the time taken to execute the actual pointer operation itself. For this reason many language implementations do not provide any form of run-time pointer checking, or if they do, they provide an option to eliminate the checks in production code. However, even while developing code it may not be practical to check every pointer operation if strict real-time constraints must be met or if the number of pointer operations is very large.

An alternative scheme is to simply hand-code some form of pointer checking at the appropriate places in the program. This approach can lead to increased program size and complexity and is likely to increase the potential for programming errors.

Run-time systems that detect the attempted use of a dangling reference do so by performing checks on the validity of a pointer before attempting to dereference it. To do this reliably, the system must be able to track the status of pointers throughout their existence, and to be able to detect when the objects that they reference have been deallocated. Three methods for performing this task will be presented.

### 4.1.1 Never Deallocate

One way of detecting an attempt to use a dangling reference is by never actually deallocating the referent object; instead the object is marked as deleted using a flag [Mae92]. The flag is checked on every pointer dereference and errors reported as appropriate. However, the storage space overhead that is incurred using this technique is potentially high. If virtual memory is used, a high level of page faulting may result from having a large heap that is only sparsely populated by currently valid objects.
4.1.2 Double Indirection

An improvement on the previous method is to use an indirection table as shown in Figure 4–1. When an object \texttt{obj} is created, a pointer that references it \PTobj is inserted into the pointer table \PT. When the object is deallocated, \PTobj is set to \texttt{null}. All user pointers that are supposed to reference \texttt{obj} are set up by the run-time system to actually reference \PTobj.

Two implementation options are available when a pointer \p is dereferenced. The first option does not allow \texttt{null} pointers to be distinguished from dangling pointers. The system always arranges that \p references an entry in \PT and is never actually \texttt{null}. A \texttt{null} pointer is represented by setting its corresponding entry in \PT to \texttt{null}. When \p is dereferenced, yielding \PTobj, this pointer is compared with \texttt{null}. If \PTobj is \texttt{null} a Null Pointer error is signaled, otherwise it is dereferenced giving \texttt{obj}. When the user assigns a \texttt{null} pointer to reference an object \texttt{obj}, the system assigns it with \PTobj and the \texttt{null} \PT entry can be garbage collected. When an assigned pointer is set to \texttt{null} the system must create a new \texttt{null} \PT entry.

The second option distinguishes \texttt{null} pointers from dangling pointers by allowing \p to be \texttt{null}. When \p is dereferenced it is firstly compared with \texttt{null}, and a Null Pointer error is signaled if appropriate. Otherwise \p is dereferenced, obtaining \PTobj, and this pointer is compared with \texttt{null}. If \PTobj is \texttt{null} a Dangling Pointer error is signaled, otherwise \PTobj is dereferenced yielding \texttt{obj}.

Overheads

The overhead of dereferencing a pointer in this method is the single extra memory reference that is required to fetch the \PT entry, and the comparisons with \texttt{null}. The ability to distinguish \texttt{null} pointers and dangling pointers is useful, as it provides more diagnostic information to the programmer, and can be obtained for the very low overhead of one extra \texttt{null} pointer comparison.
The major factor to be considered is the complexity of managing the pointer table itself. Each referent object created over the lifetime of the program will have an entry in \( \text{PT} \). An entry can never be reallocated unless it is certain that there are no user pointers that reference it. This could only be determined by a conservative garbage collector [BW88], which scans every program data structure in search of pointers to the heap in order to mark objects that are "in use". This operation may be computationally prohibitive.

There are many possible strategies for implementing \( \text{PT} \). If the maximum number of objects that will be created over the lifetime of the program can be calculated, the pointer table can simply consist of a fixed size array. In the general case this will not be possible and so some form of dynamic structure will be required. \( \text{PT} \) pointers could be created and allocated individually on the heap when needed, although this may not lead to efficient space utilisation if the run-time system has to store housekeeping information with each heap object. A more efficient strategy would be to allocate a large block of space on the heap, fill consecutive locations of this space with pointers when required, and allocate a new block when the old block is full. In this way the number of heap allocations will be very small in comparison to the number of insertions of new pointers into \( \text{PT} \).

In the majority of cases an insertion operation will simply consist of performing a range check on the pointer to the next free entry in the current block \( P_{\text{next}} \).
copying the new pointer to this position and then incrementing \( P_{\text{next}} \) to reference the new next free entry.

In conclusion, the efficiency of the double indirection method as a whole will be influenced by the particular implementation strategy that is chosen for the pointer table.

### 4.1.3 The Key Method

Fischer and LeBlanc have described how a PASCAL run-time system can detect dangling references using a technique that associates a key value with each pointer and referent object [FL80]. The high level structure of their method is shown in Figure 4–2.

Each referent object is associated with a unique non-zero \( n \)-bit integer key that is allocated at run-time when the object is created. Each pointer that references the object is represented as a pair consisting of the object’s address together with a copy of this key. When a pointer is dereferenced, a check is made to ensure that the two keys match. When the referent object is deallocated, the key is set to zero to facilitate the detection of dangling pointers. This of course does not guarantee complete safety as it is possible that a storage reallocation might accidentally recreate a previous key value at an unfortunate memory location, however with
careful definition of $n$ this possibility can be made arbitrarily unlikely. The limit on the number of unique keys that can be generated is $2^n - 1$.

**Overheads**

The allocation of a referent object incurs the small cost of producing a new key. The key must be set to zero on deallocation. The pointer and the referent object require additional storage space for their keys, which will impose some extra overhead on copying operations. Dereferencing a pointer requires, in principle, two additional memory references to fetch the keys for comparison. This overhead may be reduced if the particular hardware uses some form of prefetching, so that the pointer and its key can be fetched from memory in effectively one operation.

### 4.1.4 Summary

Several methods for the detection of dangling references at run-time have been described. All these methods rely upon the run-time system to carry out specialised actions on the programmer's behalf when creating and destroying heap objects and when performing pointer operations. They impose checking overheads on every pointer operation, which can lead to potential performance problems.

The following sections will describe a novel method for adding pointer checking to a relatively new language, C++, in a way that does not rely upon any specialised run-time system. It will also be shown how the method can help to alleviate the performance problems with run-time checking.
4.2 Adding Run-time Checking to C++

Many language designs and implementations do not provide the programmer with any help to detect dangling references due to the complexities of the task and the overheads that can be imposed. For example, the C programming language [KR88] provides very little in the way of run-time checking at all. Several attempts have been made to add run-time checking to the language by designing tools which insert calls to a run-time package [Ken83,Ste92] or by extending the language using a replacement preprocessor [Ros92,FYP93]. Of these systems only Kendall's Bcc [Ken83] is able to detect dangling pointers to the heap (by never reallocating freed memory), but it is not able to detect dangling pointers to the stack, an error which has been found to occur more frequently [Ste92]. In addition Bcc code has been estimated to be around thirty times slower than normal code, and is hence impractical for normal use.

The more recent language C++ [Str91], which is, except for a few minor details, a superset of the C language, shares its predecessor's lack of pointer run-time checking. However, C++ allows the construction of pointer substitutes that add functionality to the basic pointer. Smart pointers [Str87] and variations thereof have been used to implement persistent objects [SGS+89,Str91], reference counting [Ken91,Cop92], distributed objects [SDP92] and garbage collection [Ede92a]. The following sections will show how two of the previously described dangling reference detection strategies can be extended to C++ using smart pointers.

4.2.1 C++ and Smart Pointers

C++, while based on C, supports object-oriented programming through its powerful facilities for defining new data types and the operations that may be performed upon them. A C++ class, or user-defined type, provides data abstraction, initial-
CHAPTER 4. TRACKING POINTER STATUS AT RUN-TIME

<table>
<thead>
<tr>
<th>Operation</th>
<th>Using raw pointer</th>
<th>Using smart pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declare pointer :</td>
<td>int *p ;</td>
<td>smartpointer p ;</td>
</tr>
<tr>
<td>Declare integer :</td>
<td>int number ;</td>
<td>int number ;</td>
</tr>
<tr>
<td>Set p to point to number :</td>
<td>p = &amp;number ;</td>
<td>p = &amp;number ;</td>
</tr>
<tr>
<td>Set number to 1 :</td>
<td>*p = 1 ;</td>
<td>*p = 1 ;</td>
</tr>
</tbody>
</table>

Figure 4-3: Simple example

isation, destruction, dynamic typing, implicit type conversion and a mechanism for overloading standard operators. C++ makes use of C's ability to manipulate data efficiently, so that the use of a class does not impose any significant run-time overhead compared with the use of the equivalent C code.

One of the unique language facilities is the ability to substitute user-defined class objects (smart pointers) for basic, language defined pointers (raw pointers). Smart pointers typically encapsulate a raw pointer to a referent object together with additional operations that provide extra functionality. Operator overloading allows smart pointers to be used in normal pointer expression syntax. For example, the pointer dereference operator '*' is overloaded so that when it is applied to a smart pointer it returns the referent object. The assignment operator '=' is also overloaded so that raw pointers can be assigned to smart pointers. Figure 4-3 shows a simple example where the pointer substitute transparently replaces the raw pointer in all ways except declaration syntax.

4.2.2 Extending the Key Method to C++

Method Overview

The high level structure of the method is shown in Figure 4-4. A pointee-object, or referent object, consists of the object methods and data together with a unique
non-zero \(n\)-bit integer \(key\), that is allocated at run-time when the object is created. Each \(pointer-object\) that refers to this pointee-object consists of a basic, language defined pointer together with a copy of this key. When a pointer-object is “dereferenced”, a check is made to ensure that the two keys match. When the pointee-object is deallocated, the key is set to zero.

When a pointee-object \(obj1\) is copied to another pointee-object \(obj2\), care must be taken to ensure that the \(key\) data member is \textit{not} copied as otherwise the dereferencing of a pointer that referenced the original \(obj2\) will raise a key mismatch exception. When a pointer-object is copied, on the other hand, the \(key\) data member \textit{must} be copied.

\textbf{C++ Implementation}

Figure 4-5 shows a simple C++ implementation of a template pointer-object class \(ptr\) together with a base pointee-object class \(pte\) from which user classes would be derived. In a multiple inheritance hierarchy, where classes may be derived from more than one base class, the safe convention is that \(pte\) should be declared to be a \textit{virtual} base class. A derived class object may only contain a single virtual base class sub-object and hence an object of a class derived from \(pte\) may only contain a single key.
Assuming that a \textit{ptr} object \texttt{p} is assigned to reference a valid \textit{pte} object \texttt{obj}, if \texttt{obj} is deallocated then any attempt to "dereference" \texttt{p} will be detected. This assumption is always true when \texttt{p} is immediately assigned with a pointer returned by the \texttt{new} function, using the construct "\texttt{ptr p = new obj}".

Problems arise when the assumption does not hold. The smart pointer must be assigned using a raw pointer to \texttt{obj}. If \texttt{obj} has been deallocated (and hence the raw pointer is dangling) the storage the object used to occupy may or may not have been reused. Therefore the value in the memory location that used to be "\texttt{obj.key}" may or may not be zero. If it is zero, the mechanism will detect an attempt to assign \texttt{p} to reference the deallocated object. However if it is not zero, the value in "\texttt{obj.key}" will be interpreted by the smart pointer as the new key of the object it is being assigned to reference. It follows that the mechanism is unable to reliably detect the assignment of the smart pointer using a dangling raw pointer, and so the user must exercise care that this situation does not arise.

A related error can also occur if \texttt{p} is assigned an erroneous address produced by a \textit{cast} expression, for example:

\[
\texttt{ptr<obj> p = (obj*) \& int_array[0]}
\]

This kind of type error only tends to arise as a result of poor programming style.

These possibilities of defeating the mechanism could be prevented by say declaring \texttt{ptr.make()} and \texttt{ptr.destroy()} member functions to respectively create and destroy a \textit{pte} object, declaring the \textit{pte} constructors \texttt{private} and \texttt{friends} of the \texttt{ptr} class so that a \textit{pte} object can only be created via a \texttt{ptr} object, and removing the facility of assigning a \texttt{ptr} object using a raw pointer to a \textit{pte} object. However this would create a pointer system similar to PASCAL, and hence would require an example program to be extensively modified in order to take advantage of the new facility. We have found that the implementation given here usually requires
a minimum amount of user-program alteration, and is very useful for detecting pointers that were once valid but have subsequently become invalid.

Figure 4–6 shows a simple example of the types of error that can be detected when objects are allocated on the heap, while Figure 4–7 shows that the facility can also be used to detect dangling pointers to objects allocated on the stack.

Implementation Overheads

The given implementation imposes a reasonable additional overhead. The construction of a referent object incurs the cost of producing a new key, that must be set to zero on destruction. Construction of a smart pointer requires the initialisation of the raw pointer and the key. Both the smart pointer and the referent object require additional storage for the key data members. “Dereferencing” a smart pointer requires two additional memory references to fetch the keys. If pte is a virtual base class, the compiler imposes the additional storage overhead of a pointer to the base class sub-object in each pte-derived class object, and requires a corresponding extra memory reference using this pointer to fetch the key.

If a particular machine architecture provides hardware detection of attempts to dereference a null pointer, or produces a segmentation violation when this occurs, the overhead can be reduced by eliminating the software checks.

Extensibility

The functionality of the pointer substitute can easily be extended if required. For example, if the copying of a dangling pointer p2, in the statement ‘p1 = p2’, was to be viewed as an error then it would suffice to overload the ptr ‘=’ operator to check that ‘p2.key == p2->key’, raising an exception if this was found not to be the case.
class pte {

    public: int getkey() { return key; }
    pte() { key = getnextkey(); }
    pte(const pte &n) { key = getnextkey(); }
    pte & operator = ( const pte & n ) { return *this; }
    virtual ~pte() { key = 0; }

    private: int key;
}

} class ptr {

    public: R* operator -> () { if ( p==NULL ) throw NullPointer();
        if ( p->getkey() ! = key ) throw BadKey();
        return p;
    }

    R & operator * () { if ( p==NULL ) throw NullPointer();
        if ( p->getkey() != key ) throw BadKey();
        return *p;
    }

    void operator = ( R* q ) { p = q;
        if ( q != NULL ) { key = q->getkey();
            if ( key == 0 ) throw BadKey(); }
    }

    ptr ( R* q ) { p = q;
        if ( q != NULL ) { key = q->getkey();
            if ( key == 0 ) throw BadKey(); }
    }

    ptr () { p = NULL; key = 0; }
    ~ptr () {};

    private: R* p;
    int key;
}

Figure 4-5: C++ implementation
class store : public pte {
    public : int data1 ;
} ;

main() {
    store* obj1 = new store ; /* Initialise raw pointer */
    ptr<store> p1 = obj1 ;    /* Initialise smart pointers */
    ptr<store> p2 = new store ;
    p2->data1 = 10 ;            /* Store initial data */

    *p1 = *p2 ; /* Copy object */
    delete &*p2 ; /* Delete object */
    int contents = p2->data1 ; /* Exception BadKey raised here */
    /* ( Should have read p1->data1 ) */
}

Figure 4-6: Heap example

ptr<store> p3 ; /* Global smart pointer */

void f () { /* Function definition */
    store obj1 ; /* Declare object on the stack */
    p3 = &obj1 ; /* Assign smart pointer */
}

main() {
    f();
    p3->data1 = 10 ; /* Exception BadKey raised here */
}

Figure 4-7: Stack example
The use of the smart pointer has only a small impact on the syntax of the given example programs. One problem that is encountered is that the delete function requires a pointer operand, not an object of type ptr<store>, and hence each invocation must take the form delete &*smartpointer. This construction simply dereferences the smart pointer, obtaining the referent object, and then takes the address of this object, yielding a raw pointer to it. The delete &*operand construction is used in preference to a member function smartpointer.delete() as it will have the same net result if operand is a raw or a smart pointer (i.e. it will yield a raw pointer to the referent object). Therefore the extra checking mechanism can be turned on or off as required simply by modifying the pointer declarations.

Edelson [Ede92b] has shown that in general C++ does not support the idea of a pointer substitute well enough to allow the seamless replacement of basic pointers by smart pointers, due to problems concerning implicit pointer type conversions, and also that supporting pointers to const objects actually requires two smart pointer classes per object class, with the class for the pointer to the non-const object being derived from the class for the pointer to the const object.

4.2.3 Extending the Double Indirection Method to C++

Another implementation of the smart pointer will now be presented. The new implementation uses an extension of the double indirection method for detecting dangling references from Figure 4-1. In C++, raw pointers to an object obj may coexist with smart pointers that reference it (indeed smart pointers are initialised using raw pointers). Consider the example shown in Figure 4-8. If obj₁ is deallocated using the construct delete p, the indirect pointer table entry PToObject₁ will not be updated to null and hence sp will become a dangling smart pointer. In order to avoid this possibility, the obj₁ destructor must be able to set PToObject₁ to null and so each referent object must be augmented with another pointer prev
that points back at the corresponding PT entry. Figure 4–9 illustrates the overall structure of this augmented method.

C++ Implementation

The new template pointer-object class `ptr` and base pointee-object class `pte` are shown in Figure 4–10. The `pte` constructors call a globally accessible pointer table manager `PTman` to make an entry for the object in the table.

The user-level behaviour of the smart pointer is the same as that of the key based mechanism. The new implementation has a similar limited ability to detect
attempts to assign the smart pointer using a dangling raw pointer. Consider the assignment \( p = q \), where \( p \) is a \( ptr \) object and \( q \) is a raw pointer to \( obj \), an instance of a subclass of \( pte \). If \( obj \) has been deallocated, \( \text{"obj.revptr"} \) may or may not be \( \text{null} \), and if it is not \( \text{null} \) it will be interpreted by the mechanism as the pointer to the corresponding \( PT \) entry.

A more secure version of the implementation might attempt to verify that \( \text{"obj.revptr"} \) references a pointer in \( PT \). This would require the system to keep careful track of all the \( PT \) entries so that they can be searched. If consecutive memory locations are used to store the entries, bounds checking can be used to perform an efficient search. The extra security that this technique gives will impose an extra overhead upon every smart pointer assignment. Complete safety is not guaranteed as a storage reallocation might accidentally create a bit pattern at an unfortunate memory location which the system can interpret as a pointer to a \( PT \) entry. However with a large address space and a relatively low number of referent objects this possibility is fairly unlikely.

**Implementation Overheads**

Construction of a referent object requires the \( PT \) manager to create an entry in the pointer table for the object and return a pointer to this entry. The speed of this operation will be governed by the strategy used to implement \( PT \). Destruction of a referent object requires that the pointer to the \( PT \) entry, and the entry itself, be set to \( \text{null} \). The smart pointer and the referent object both require storage for their \( \text{void**} \) data members. "Dereferencing" the smart pointer requires one additional memory reference to fetch the \( PT \) entry, and two comparisons with \( \text{null} \) to perform the checking functions. In common with the key-based method implementation, hardware checks for attempts to dereference \( \text{null} \) pointers can reduce the software overhead. The overhead that the compiler imposes if \( pte \) is declared to be a virtual base class is also the same.
class pte {

public: void** getrevptr() { return revptr; }
    pte() { revptr = PTman->MakeEntry(this); }
    pte(const pte& n) { revptr = PTman->MakeEntry(this); }
    pte& operator = (const pte& n) { return *this; }
    virtual ~pte() { *revptr = NULL; revptr = NULL; }

private: void** revptr;
}

template <class R> class ptr {

public: R* operator -> () { if ( p == NULL ) throw NullPointer();
    R* pointer = *(R**) p;
    if ( pointer == NULL ) throw DanglingPointer();
    return pointer; }

    R& operator * () { if ( p == NULL ) throw NullPointerQ;
    R* pointer = *(R**) p;
    if ( pointer == NULL ) throw DanglingPointer();
    return *pointer; }

    void operator = ( R* q ) { if ( q != NULL ) {
    p = q->getrevptr();
    if ( p == NULL ) throw DanglingPointer(); }
else p = NULL; }

    ptr ( R* q ) { if ( q != NULL ) {
    p = q->getrevptr();
    if ( p == NULL ) throw DanglingPointer(); }
else p = NULL; }

    ptr () { p = NULL; }
~ptr () {};

private: void** p;
}

Figure 4-10: C++ implementation
4.2.4 Comparison of the Two Implementation Methods

The following analysis assumes that \textit{pte} is \textit{not} declared to be a virtual base class. The analysis for the virtual case is similar to that presented.

(a) Storage Overhead

The storage requirements for each method are as shown in Figure 4–11. Assuming that the \textit{n}-bit integer key occupies the same amount of storage as one raw pointer, the key technique uses less storage space than the double indirection method when the number of \textit{pte}-type objects exceeds the number of \textit{ptr}-type objects. Similarly, the double indirection method uses less storage when the number of \textit{ptr}-type objects exceeds the number of \textit{pte}-type objects.

(b) Time Overhead

The time overhead for both methods is determined by:

1. the time taken to construct the referent objects
2. the number of memory references required to assign the smart pointer \textit{and}
3. the number of memory references required to “dereference” the smart pointer

In the key method, the construction of a \textit{pte}-derived object requires that a new key is generated. A simple way to do this is to start with a random \textit{n}-bit global
master key and produce a new key when required by incrementing the master. When the master key overflows it can be reset to 1. Given sufficiently large $n$, this should be more than adequate for most applications and can be implemented very efficiently.

In the double indirection method, the construction of a pte-derived object requires that the PT manager inserts a pointer to the object into its table. As previously described, the speed of construction of a pte-derived object will therefore be determined by the strategy that the pointer table manager uses to store its data.

The key method requires three memory references to assign a non-null smart pointer and two memory references to dereference it. The double indirection method requires two memory references for assignment and one memory reference for dereferencing. Therefore the double indirection method requires one less memory reference than the key method to both assign and dereference a smart pointer. This apparent speed advantage may be offset by the cost of creating the pte-derived objects.

**Overall Conclusions**

While the double indirection method has a distinct speed advantage when assigning and dereferencing the smart pointer, the overall "best method" to use may be influenced by the respective storage overheads of the two methods, the object creation and destruction patterns of the particular application program and the time taken to create the pte-derived objects. Other factors that might affect the total speed of operation are the availability of hardware based null pointer checking, the overheads of pointer casting and the use of registers for temporary values.
4.2.5 Summary

The smart pointer systems that have been presented are simple tools to help the programmer to detect the use of dangling references. They enable the program itself to detect an error of this kind rather than having to rely upon the language run-time system to do so. As a result, dangling pointer checks can be easily added to any given C++ language implementation. Users can precisely target the code they wish to analyse, and switch the extra checks on and off as required by selectively modifying the variable declarations. It may therefore not be necessary to incur extra checking overhead for every pointer operation, and hence the total amount of run-time checking overhead can be correspondingly reduced. The method can be extended to support pointers to constant objects and cope with more complex class hierarchies using the methods presented by Edelson [Ede92b].

In other situations pointers are used to increase the speed of access to data using pointer arithmetic, where calculations are directly performed upon the memory addresses of variables. However code that contains pointer arithmetic is notoriously difficult to understand and debug. The following section will describe a different smart pointer system that can be used to add run-time pointer arithmetic checking to a C++ implementation.

4.3 Objects and Pointer Arithmetic

C++ pointer arithmetic can be used to provide more efficient access to certain types of variable, especially array variables. For example, Figure 4-12 shows a very efficient, compact section of code that copies a string 't' to a string 's' (assuming that 't' is assigned before the copy).
This example is typical of the use of pointer arithmetic to access an array variable: a pointer to an element ‘e’ is held in a variable ‘p’. A pointer to element ‘e + 1’ is obtained by adding the size of the element to ‘p’. Therefore no array bounds checking takes place and so run-time security is sacrificed in favour of speed and syntactical simplicity.

The efficiency of pointer arithmetic cannot be duplicated via array access by index, as shown in Figure 4–13, because although the array access construct ‘s[i]’ generates a pointer to element ‘s[i]’, after the pointer has been used to access the element it is simply discarded.

Clearly there will be occasions when some measure of run-time protection is required and so a bounds checking technique is needed, preferably one that does not discard any potentially useful data. Also, when debugging a program there are times when an access to an array element using a pointer has to be strictly monitored and controlled. Ideally it should be possible for a system to provide these facilities efficiently, with a localised impact on the syntax of any program that uses them.

A pointer class\(^2\) that implements array bounds checking will be presented. This

---

\(^2\) This work was originally performed before version two of the C++ language and its associated reference manual [Str91] became available. The new language revision corrects two of the implementation problems that had previously been found with the method, by distinguishing the pre- and post-fix application of the ‘++’ and ‘-’ operators and
CHAPTER 4. TRACKING POINTER STATUS AT RUN-TIME

char s[], t[] ; /* Definitions */
int i = 0 ;
while ( s[i] = t[i] ) i++ ; /* Copy operation */

Figure 4-13: Copy string t to s using standard array access

technique can be extended to provide monitoring and diagnostic information in the same fashion. The localised nature of the additional code means that it can be very easily added and removed as required, with no major effect on the code of the main body of the program.

Implementation

Figures 4-14 and 4-15 show a simple implementation of an array bounds checking template class Abe. The constructor takes a pointer to the first element of the array and the number of elements as arguments. The post-fix implementation of the ‘++’ operator is used in Figure 4-16 to perform the same string copy operation as Figure 4-12. Another example of the use of the class is given in Figure 4-17, the output from which is shown in Figure 4-18.

The system-defined Bounds() exception handler would typically access the runtime system information in order to be able to provide the user with the program line number where the error occurred, together with other diagnostic information such as current variable values. The program would then be terminated.

Other pointer arithmetic operations can also be provided. C++ defines that the subtraction of two pointers yields the number of elements between them, as

providing the template facility to allow class generalisation. However further problems remain as will be shown.
template<class R> class Abc {

public: Abc ( R* s, int n ) { val = s ; max = s + n - 1;
    maxpl = max + 1 ; min = s ; minml = s - 1 ; }
~Abc () {};

    Abc& operator ++ () { ++val;
    if ( val > maxpl ) throw Bounds("++") ; return *this ; }
    Abc operator ++ ( int ) {
    if ( ! temp ) temp = new Abc ( min, max - min + 1 ) ;
    temp->val = val ; if ( ++val > maxpl ) throw Bounds("++") ;
    return *temp ; }

    Abc& operator -- () { --val;
    if ( val < minml ) throw Bounds("--") ; return *this ; }
    Abc operator -- ( int ) {
    if ( ! temp ) temp = new Abc ( min, max - min + 1 ) ;
    temp->val = val ; if ( --val < minml ) throw Bounds("--") ;
    return *temp ; }

    friend Abc& operator + ( int n, Abc oc ) ;
    Abc& operator + ( int n ) { val += n ;
    if ( val > maxpl ) throw Bounds("+") ; return *this ; }
    Abc& operator += ( int n ) { val += n ;
    if ( val > maxpl ) throw Bounds("+=") ; return *this ; }

    int operator - ( Abc oc1 ) { return val - oc1.val ; }
    Abc& operator - ( int n ) { val -= n ;
    if ( val < minml ) throw Bounds("-") ; return *this ; }
    Abc& operator -= ( int n ) { val -= n ;
    if ( val < minml ) throw Bounds("-=") ; return *this ; }

    friend int operator == ( Abc oc1, Abc oc2 ) ;
    R& operator * () { if ( val<min || val>max ) throw Bounds("*") ;
    return *val ; }

    Abc* temp = NULL ;
} ;

Figure 4-14: Abc pointer class definition
template<class R> int operator == ( Abc<R> oc1, Abc<R> oc2 ) {
    return oc1.val == oc2.val ;
}

template<class R> Abc<R>& operator + (int n, Abc<R> oc ) {
    oc.val += n ;
    if ( oc.val > oc.maxpl ) throw Bounds("+");
    return oc ;
}

**Figure 4–15:** Abc pointer class definition

```cpp
char s[10], t[10] ; /* Definitions */
Abc<char> os(s,10), ot(t,10) ;
while ((*os++ = *ot++)) ; /* Copy operation */
```

**Figure 4–16:** Copy string t to s using the Abc pointer class
illustrated by the definition of the ‘-' operator. The comparison operators ‘! =’, ‘<’, ‘>’, ‘<=' and ‘>=' can also be defined in a similar manner to the ‘==’ operator implementation.

This approach differs in two important ways from that suggested by Stroustrup [Str91, p241] as will now be shown.

Special Cases

In considering array bounds checking for C++ it might be argued that two special cases exist, namely when a pointer addresses an array element one greater than the defined maximum or one less than the defined minimum (for example in Figure 4-12 if the maximum element of the array addressed by ‘s’ contains the ‘\0’ character then ‘s’ will address the element one greater than this at the end of the loop). If we wish to allow the use of this type of loop then we must consider the dereferencing of such a pointer to be erroneous but permit its construction.

Post-increment and Post-decrement operations

The recent introduction into C++ of the ability to distinguish overloaded pre- and post-fix increment and decrement operations has created a major implementation problem for post-fix expressions. If ‘p’ and ‘t’ are simple pointers to a type X, then the statement “p = t ++” assigns the value of ‘t’ to ‘p’ and then increments ‘t’ by

\[ 3 \]

Note that this type of code can be avoided entirely in many cases by the use of an optimising compiler – the most basic task of this tool is the identification of loops such as that shown in Figure 4–12 and the corresponding production of much more efficient object code that uses machine address arithmetic. Therefore the need to use explicit pointer arithmetic is elegantly eliminated; however it will assumed in this discussion that an optimising compiler is not being used.
CHAPTER 4. TRACKING POINTER STATUS AT RUN-TIME

```c
int main() {
    char s[10];
    Abc<char> os(s,6);

    sprintf(s,"abcdefghi");
    for (int i=1; i<10; i++) {
        printf("Letter %i is %c \n",i,*os);
        os += 2;
    }
}
```

Figure 4-17: Example program using the Abc pointer class

sizeof(X). Ideally, when ‘p’ and ‘t’ are objects, C++ would translate “p = t ++” into the equivalent instruction sequence “p = t ; t ++”. Instead however the post-fix operator must be defined as a function that returns an object of the same type as ‘p’. It is simply not possible to have the post-fix operator return the basic pointer contained within the smart pointer as suggested by Stroustrup, since for example in the expression “*t ++” the basic unary operator ‘*’ would then be applied to this basic pointer, instead of the overloaded unary operator ‘*’ being applied to the object ‘t’ as required.

The postfix operator must both update the object and return its previous value, so it must construct a new copy of object ‘t’, perform its function upon the original object, and return the copy. However this means that reference constructions of the form “Abc<char>& = t ++” will not work properly, as the reference will be constructed to a copy of the previous value of ‘t’, not ‘t’ itself. Therefore this type of expression must be avoided if the smart pointer is to function correctly. Hopefully future revisions of C++ will rectify this shortcoming in the language, perhaps by performing the equivalent expression translation that was suggested above.
Letter 1 is a
Letter 2 is c
Letter 3 is e

** Bounds exception raised at line 8 of procedure 'main()'.
** Pointer 'os' is out of range.
** Program terminated.

Figure 4-18: Example program output

Summary

This section has shown that it is possible to perform pointer arithmetic using smart pointers for single classes, resulting in a very straightforward and usually syntactically transparent way of attaching extra run-time monitoring to pointer arithmetic. The technique achieves the goal of code localisation (within a class description) and therefore a simple redefinition of the pointer class suffices to remove the extra checks and regain the efficiency of true pointer arithmetic when required. As previously described, Edelson [Ede92b] has shown that C++ is unable to transparently support the generalisation to class hierarchies, constant objects and multiple inheritance; however the methods shown are sufficiently useful to be worth serious consideration as monitoring and debugging tools for many otherwise difficult situations.
Chapter 5

Composite Objects

This chapter will consider what it means for a composite object to be unassigned, together with the problems of tracking a composite object's state. Language constructs that allow the control of run-time unassigned checking for both scalar and composite objects will be presented.

5.1 Unassigned status

The issue of unassigned composite objects has been considered by Winner [Win84]. The many problems involved in defining what it means for a composite object to be unassigned stem from the so-called “dual view”, where arrays and records can be viewed as both composites of other objects and as objects in their own right. Dijkstra [Dij76] attempts to resolve many of these problems by eliminating the dual view, so as to subset the composite to manipulate its parts when required. He introduces the array operation \( ax:alt(i,x) \), which is semantically equivalent to the Algol 60 assignment operation \( ax[i]:=x \), in an attempt “to stress that such an operation affects the array \( ax \) as a whole”. This is subsequently abbreviated to \( ax:(i)=x \), “a notation that is somewhat shorter, reminiscent of the so much more...
familiar assignment statement, and still reflects by its opening ‘ax:’ that we must view it as affecting the array variable ax’.

It seems clear that the syntactical addition of an extra colon character to the assignment statement is not really the issue, it is the fact that the operation is being viewed differently. There is simply no way of avoiding the fundamental truth (notationally or otherwise) that the state of a composite object is defined by the state of its components. Therefore altering a component will always affect the total state of the composite. The unassigned status of the composite can simply be viewed as a part of the total state. The unassigned status of the components may contribute to the unassigned status of the composite and thus its total state.

Assuming one of the antipropagation regimes from Chapter 3, Winner [Win84] identified two approaches that could be taken when defining a composite object to be unassigned:

1. A composite object is unassigned iff ∀ components P, P is unassigned.

2. A composite object is unassigned iff ∃ a component P, where P is unassigned.

Under Definition 1, the “unassigned = all” approach, with Proposition 2.2 the use of an unassigned composite object should only be viewed as an error if it involves an incorrect use of an unassigned component. This implies that objects should be treated consistently, regardless of whether they are independent or components of another object.

If a composite object has at least one assigned component then the composite is assigned and may be copied. For example, if X and Y are arrays with X[1] = 1 and X[2] unassigned then the statement Y := X would be legal, propagating the unassigned value in X[2] to Y[2]; however the statement Y[2] := X[2] would be erroneous. Therefore element by element array operations may give different
results to object level operations, which is an inconsistency resulting from the dual view.

Under Definition 2, the “unassigned = some” approach, with Proposition 2.2 the object-level operation given above would be erroneous. The approach prohibits the use of partially assigned composite objects, which is unfortunate as the use of objects of this type is very common. For example a string variable is usually represented as an array of characters, and so a string containing less than its maximum number of characters will only be partially assigned. Therefore the use of escapes or specious initialisations to circumvent the antipropagation mechanisms would be encouraged. As will be shown in later sections, this can have a serious impact upon the system’s capability to detect errors.

In summary, the “unassigned = all” approach treats the unassigned elements of composite objects inconsistently in assignments, while the “unassigned = some” approach avoids this inconsistency by discouraging composite object level assignments.

5.2 State Tracking

Given either the “unassigned = some” or the “unassigned = all” approach, the set of propagation rules that is adopted may present situations in which the assigned state of a composite object must be determined. Two methods of state evaluation are apparent:

- **Lazy evaluation**: the state is evaluated when it is actually required or

- **Continuous evaluation**: the state is evaluated when the assigned status of one of the components changes.
state = undefined
for each element in the composite do
    if the element is defined then
        begin
            state = defined
            exit
        end
od

Figure 5-1: Composite state under “unassigned = all” definition

In the lazy evaluation approach the composite state does not require evaluation on every component assignment, whereas with continuous evaluation the composite state is always immediately available, but each component assignment carries an extra time penalty.

Therefore, defining $TM$ as the total time spent maintaining the composite state and $TS$ as the time taken to calculate the composite state at a given instant we see that:

$$TM_{continuous} \geq TM_{lazy}$$

but $TS_{lazy} \gg TS_{continuous}$

With lazy evaluation the assigned status of the composite under the “unassigned = all” definition may be determined using the algorithm in Figure 5-1. Figure 5-2 shows the corresponding algorithm for the “unassigned = some” definition. Both algorithms have run-time $O$(number of composite object components).

Altering the state of a component $P$ from unassigned to assigned or vice versa may alter the assigned status of the composite (and will always alter its total
state = defined
for each element in the composite do
  if the element is undefined then
    begin
      state = undefined
      exit
    end
  end

Figure 5-2: Composite state under "unassigned = some" definition

<table>
<thead>
<tr>
<th>Defn</th>
<th>P Before</th>
<th>P After</th>
<th>Composite Before</th>
<th>Composite After</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>unassigned</td>
<td>assigned</td>
<td>unassigned</td>
<td>assigned</td>
</tr>
<tr>
<td></td>
<td>unassigned</td>
<td>assigned</td>
<td>assigned</td>
<td>assigned</td>
</tr>
<tr>
<td></td>
<td>assigned</td>
<td>unassigned</td>
<td>assigned</td>
<td>may be unassigned</td>
</tr>
<tr>
<td>2</td>
<td>unassigned</td>
<td>assigned</td>
<td>unassigned</td>
<td>may be assigned</td>
</tr>
<tr>
<td></td>
<td>assigned</td>
<td>unassigned</td>
<td>assigned</td>
<td>unassigned</td>
</tr>
<tr>
<td></td>
<td>assigned</td>
<td>unassigned</td>
<td>unassigned</td>
<td>unassigned</td>
</tr>
</tbody>
</table>

Figure 5-3: Component 'P' and composite state changes
state) as shown in Figure 5-3. Therefore with continuous evaluation under either
definition, each component assignment and unassignment will require extra state
checking operations in order to maintain the assigned state of the composite.

It is important to note that if the programming language allows pointers to
components of composite objects, it may be impossible to tell whether or not a
component has changed status on assignment, or even to determine which com-
posite has been affected. Therefore continuous state evaluation may not be viable
in this case. Also, as a component may be passed as a parameter to a procedure
or a function, if the language permits aliasing via parameter passing then all as-
signments to parameters would require extra checks. When aliasing is forbidden
it would be sufficient to check the component’s assigned state upon return to the
point of call.

5.3 Examples

Having examined the theoretical issues of what it means for a composite object to
be unassigned, and the problems of state tracking, this section will describe the
methods that several popular programming languages have adopted for dealing
with composite objects. The advantages and disadvantages of each scheme will be
discussed.

“Unassigned = some”

The definition of Standard Pascal states that any access to the value of an un-
assigned variable is an error. A Pascal composite object is assigned only when
all of its components are assigned. As shown in [KW90], this can actually reduce
the system’s ability to detect errors. It is not possible to pass a partially assigned
array as a parameter under this definition, and so programmers will be encour-
aged to “falsely” initialise array elements so that “spurious” error traps can be avoided. However if an access is made to an array element that should have been unassigned it will be successful, returning the “false” initialisation value instead of raising an error.

In practice, after encountering several spurious errors on structure transfer operations, a programmer would be likely to either initialise the elements of all structures or switch off the unassigned variable checks [KW90]. The protection given by the language definition would therefore be substantially reduced.

Unassigned for Scalar Types Only

In Ada the detection of unassigned scalar variables is optional and there is no notion of unassigned for composite objects. This approach for composites is the least demanding on an implementation and does not impose any run-time overhead above that required to detect unassigned scalar variables. However it provides little assistance during the development and testing of programs.

In Modula 2, unassigned variable errors can only arise on access to scalar variables, not composite objects. When a composite object is copied the assigned status of each component is copied but not checked until the value of a component is accessed. Therefore partially assigned structures can be passed as parameters. This approach should trap most errors as most Modula 2 structures will, in the end, be accessed as simple types [KW90].

“Unassigned = all”

If every component of a composite object is unassigned, an operation that copies the structure is, in all probability, an error. Therefore errors could be detected earlier than in the Modula 2 definition if composite-level operations on totally unassigned structures are also trapped. By considering unassigned scalar variables
to be totally unassigned, the rule becomes: operations accessing totally unassigned objects are errors [KW90]. This rule has been successfully implemented in a Pascal programming environment [Kem89] using a word-map representation for unassigned.

When writing files of structures, care must be taken to write out the assigned states of the components so that they are available when the file is read back in. At the implementation level, this is done automatically in the special values method for unassigned, and can be achieved in the bit- and word-map methods by writing the unassigned map to a parallel file that is later read synchronously with the data file.

This rule treats composite objects inconsistently when assigned as a whole and when assigned on a component by component basis, as previously described. Also, some transfers of totally unassigned structures may be valid. For example, a function to sum the components of an array should be able to return a sum of zero when passed a totally unassigned array and a size variable indicating that it has no elements [KW90].

5.4 Simplifications

The dual view of composite objects creates special problems and inconsistencies that should be resolved. For any particular example it is the combination of the viewpoint of an operation, the definition of what it means for a composite object to be unassigned, and the assigned states of the components which dictates whether an exception should or should not be raised. Winner suggests that the problem might be simplified by defining that:

"... composites are never unassigned and that only incorrect uses of elementary unassigned components will be considered incorrect."
As the use of an unassigned component may occur some considerable time after it was propagated in a composite object-level assignment, it may be much more difficult to trace the error back to its original source. In the ideal case the system would provide the maximum help to the programmer by trapping all unintentional propagation of unassigned variables. This simplification also suffers from the inconsistency problem. The copying of an object containing unassigned components will not generate an error if it is carried out at object level, for example by using a memory region block copy, but it will raise an error if an iterative component-level copy procedure is used. As described in previous sections several languages use this simplification as it requires no additional run-time overhead above that used already to check for unassigned scalars. However these arguments tend to suggest that a more helpful system should not adopt this approach.

Another Solution

[Win84] also suggests that if a noncontiguous subdomain of the components can be specified, then some of the problems raised in the previous discussion could be solved.

“For example, suppose one could name, for array c, c(assigned) meaning an object consisting of all the assigned components of c. In this case the “unassigned = some” approach could be used because the usable portion of the partially filled composite is nameable.”

Figures 5–4 through 5–7 consider the use of this notation in each of the possible propagation contexts. By specifying the assigned part of a composite using the notation s(assigned), an effective escape from raising an error is obtained when required. Consider two structures s and t that have the same type. The operation of assigning s(assigned) to t under the “unassigned = some” definition should signal an error if the equivalent assignment t := s, under the “unassigned = all”
### Figure 5-4: Copy operation

<table>
<thead>
<tr>
<th>Location</th>
<th>Statement</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Within function</td>
<td><code>result ::= s(assigned)</code></td>
<td>Generates an error if <code>s(assigned) = { }</code>.</td>
</tr>
<tr>
<td></td>
<td><code>result ::= s</code></td>
<td>Generates an error unless <code>s(assigned) = s</code>.</td>
</tr>
<tr>
<td>When calling</td>
<td><code>struc ::= fn_name</code></td>
<td>Generates an error unless <code>fn_result(assigned) = fn_result</code>.</td>
</tr>
<tr>
<td></td>
<td><code>struc ::= fn_name(assigned)</code></td>
<td>Generates an error if <code>fn_result(assigned) = { }</code>.</td>
</tr>
</tbody>
</table>

### Figure 5-5: Function result

<table>
<thead>
<tr>
<th>Location</th>
<th>Statement</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Within procedure</td>
<td><code>s ::= result</code></td>
<td>Generates an error unless <code>result(assigned) = result</code>.</td>
</tr>
<tr>
<td></td>
<td><code>s ::= result(assigned)</code></td>
<td>Generates an error if <code>result(assigned) = { }</code>.</td>
</tr>
<tr>
<td>When Calling</td>
<td><code>proc_name(out s)</code></td>
<td>Generates an error unless <code>s(assigned) = s</code>.</td>
</tr>
<tr>
<td></td>
<td><code>proc_name(out s(assigned))</code></td>
<td>Generates an error if <code>s(assigned) = { }</code>.</td>
</tr>
</tbody>
</table>

### Figure 5-6: Out mode parameter 's'
CHAPTER 5. COMPOSITE OBJECTS

<table>
<thead>
<tr>
<th>Location</th>
<th>Statement</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Within procedure</td>
<td>$p := s.data1 + 2$</td>
<td>Generates an error if $s.data1$ is unassigned.</td>
</tr>
<tr>
<td></td>
<td>$b := s$</td>
<td>Generates an error unless $s(assigned) = s$.</td>
</tr>
<tr>
<td></td>
<td>$b := s(assigned)$</td>
<td>Generates an error if $s(assigned) = { }$.</td>
</tr>
<tr>
<td>When calling</td>
<td>$\text{proc.name}(\text{in } s)$</td>
<td>Generates an error unless $s(assigned) = s$.</td>
</tr>
<tr>
<td></td>
<td>$\text{proc.name}(\text{in } s(assigned))$</td>
<td>Generates an error if $s(assigned) = { }$.</td>
</tr>
</tbody>
</table>

Figure 5-7: In mode parameter ‘s’

definition, would signal an error. Function results and in/out mode parameters using the (assigned) operator should also operate in this way.

However, once used with a particular structure, the (assigned) operator would have to be used in all subsequent references to that structure in order to avoid signaling an unintentional error, until it was certain that each component of the structure was assigned. This problem might over-encourage the use of escapes and hence defeat the purpose of having an antipropagation mechanism.

A Better Solution

If a composite structure is unassigned under the “unassigned = all” definition then it is also unassigned under the “unassigned = some” definition, while the converse is not true. The following section presents a notation which allows a temporary escape from the “unassigned = all” definition to the “unassigned = some” definition.
Figure 5-8: Example propagations

**Definition**

Under the “unassigned = all” approach and given a suitable composite object \( s \) the statement:

\[
\text{fully assigned } s
\]

is defined to raise an exception if there are any components of \( s \) which are unassigned.

Figure 5-8 illustrates the use of the new construct in each of the possible propagation contexts, under the “unassigned = all” definition. In each case an exception is raised if the argument is unassigned by the “unassigned = some” definition. If all instances of the statement \( fn\_result := \ldots \) in a particular function use the fully assigned construct then it is not required at the point of function call, as the status of the composite will have been checked just prior to this point. Similar observations also apply to both in and out mode parameters.

In conclusion, this new notation eliminates the state tracking problem of the (assigned) construct, and the programmer has complete control over the strictness of the “unassigned” definition being applied at any point. Therefore the
inconsistency in the treatment of unassigned elements in assignments of the "unassigned = all" approach may be explicitly circumvented by the programmer when required, and the flexibility of the dual view of composite objects is available under programmer control.

5.5 Composite State Tracking revisited

Under strong antipropagation, state checking must take place on variable assignment, on returning a function result and on using an in / out parameter. In the "unassigned = all" approach a check must be made for the case where all the components are unassigned. When the fully assigned construct is used, this simply changes to a check for the case where all the components are assigned. Therefore on average the new construct should require no additional run-time overhead to implement in the cases where state checking is taking place already according to the antipropagation rule in use.

It is important to note that on its own the new construct does not allow a selective increase in the level of antipropagation. For example, under weak antipropagation and the "unassigned = all" rule with composite object s, the statement:

\[
\text{fn\_result} := \text{fully assigned}\ s
\]

is not equivalent to escapable antipropagation, since if s contained only a few elements that were assigned, by the "unassigned = all" rule s is assigned and yet the example would raise an error since s is not assigned by the "unassigned = some" rule.

If the ability to selectively increase the level of antipropagation for both the "unassigned = all" and the "unassigned = some" definitions is required, another construct must be introduced:
assigned $s$

which raises an exception if the composite object $s$ is unassigned under whichever definition is being used. For example, under the “unassigned = all” rule and weak antipropagation the statement:

$$fn\_result := \text{assigned} \ struct$$

raises an exception if $struct$ is unassigned, which is equivalent to the result under the “unassigned = all” rule with escapable antipropagation.

This construct can also be applied to scalar objects – for example under escapable antipropagation the statement:

$$\text{proc\_name(out assigned b)}$$

will raise an exception if $b$ is unassigned, which is the same result as that generated under strong antipropagation.

A corresponding construct:

maybe unassigned $s$

can be defined to decrease the level of antipropagation when required. For example, under strong antipropagation the statement:

$$\text{proc\_name(in maybe unassigned s)}$$

will not raise an exception when $s$ is unassigned.
Summary

In summary, the **fully assigned** construct provides a temporary escape from the "unassigned = all" definition for composite objects, to the stricter "unassigned = some" definition. The **assigned** construct provides a temporary increase in the level of antipropagation for both scalars and composite objects. The **maybe unassigned** construct provides the corresponding temporary decrease. The use of the new constructs gives much more control to the programmer over the run-time system's actions and allows the adoption of the most appropriate strategy at each point.

As previously described in Chapter 3, run-time checks are used during program testing to help with the detection and location of errors, but are often removed from production code for efficiency reasons. Winner's proposed language elements such as an **unassigned** choice in 'case' statements allow variable status to conditionally control normal program execution. These facilities make it difficult to quickly disable run-time checking for production software [KW90]. In contrast, the new language elements presented here control the circumstances under which an error condition should be signaled to interrupt normal program execution. This type of check can be easily removed from production software by simply making the new constructs take no action.
Chapter 6

Graph Display

6.1 Introduction

As noted in Chapter 2, many attempts have been made to devise systems for the display of graphs or graphical data structures, and many algorithms have been created to lay out specific types of graph. No single system or algorithm claims to be able to produce perfect results for all types of graph. Indeed sometimes the sheer complexity of the graph itself leads to very difficult display-related problems.

The fundamental goals of these systems are to lay out a graph in such a way that it is easy to understand and, when appropriate, to make sure that the layout conveys any inherent hierarchical or sub-structural orderings or clusterings. The following three chapters assume the previously described techniques to ensure data validity, and describe a new graph clustering technique, the use of Edge-Reduction Nodes, to simplify complex directed graphs. This work is related to that of Newberry [New89], who independently developed the notion of the functionally equivalent “edge concentration”. A more theoretical approach to the development and use of this technique is adopted here, and in doing so the true nature of the problem is revealed.
This chapter presents the clustering technique together with a basic theory of the type of graph for which it is applicable. It is shown that many factors and goals combine together to affect how the technique should be applied to a graph. Chapter 7 demonstrates that for one particular goal, the identification of the largest complete bipartite subgraph contained within a bipartite graph, the obvious purely graph theoretical approach cannot be used. Chapter 8 presents several heuristic-based algorithms to perform the clustering operation, developing in the process a novel self-timing algorithm (and indeed algorithm class). The effects of these algorithms are demonstrated on several examples drawn from large-scale software engineering.

6.2 Graph Theory

The following sections present a simple theory of two-level, or bipartite graphs.

6.2.1 Definitions and Notation

A graph where every pair of distinct vertices defines an edge is called a complete graph. The complete graph with $n$ vertices is denoted $K_n$. If the vertices of a graph $G$ can be partitioned into two subsets, $V_1$ and $V_2$ such that every edge of $G$ connects a vertex in $V_1$ to a vertex in $V_2$ then $G$ is said to be bipartite. A complete bipartite graph is a bipartite graph in which every vertex in $V_1$ is connected to every vertex in $V_2$. This graph is denoted $K_{x,y}$ where $|V_1| = x$ and $|V_2| = y$. Except where stated otherwise, it is assumed that all graphs referred to in this work are directed.

A path is a sequence $v_i, e_i, v_{i+1}, e_{i+1}, \ldots, e_{j-1}, v_j$ of alternating vertices $v_a$ and edges $e_a$ such that for $i \leq a < j$, $e_a$ is incident with $v_a$ and $v_{a+1}$.

Two vertices $V_i$ and $V_j$ are connected if there exists a path from $V_i$ to $V_j$. 
A graph $G$ is connected if every vertex $V_i \in G$ is connected to every vertex $V_j \in G$, $v_i \neq v_j$, otherwise it is disconnected.

Two graphs $G_1$ and $G_2$ are isomorphic if there is a one-to-one correspondence between their vertices, such that the number of edges joining any two vertices in $G_1$ is equal to the number of edges joining the corresponding two vertices in $G_2$.

An edge may be divided into two edges in series by the insertion of a vertex of degree two. Two graphs are homeomorphic if one can be made isomorphic to the other by inserting or deleting vertices of degree two in this way.

A minimally vertex connected (mvc) bipartite graph is defined as a bipartite graph where there is at least one edge incident at each vertex. This definition improves the efficiency of the algorithms presented later. Each vertex $V_i$ of an mvc-bipartite graph is connected to some other vertex $V_j$, even if the graph as a whole is disconnected. Figure 6-1(a) shows an example of an mvc-bipartite graph while Figure 6-1(b) shows a graph that does not satisfy the definition.

$N_{x,y}$ represents the bipartite graph where $|V_1| = x$, $|V_2| = y$ and $|E| = 0$. Figure 6-2 shows $N_{2,3}$.

$B^z_{x,y}$ represents the set of mvc-bipartite graphs that are possible using the 'vertex skeleton' $N_{x,y}$ with $z$ edges.

$T^z_{x,y}$ is the cardinality of the set $B^z_{x,y}$.

$G_{x,y}$ represents the set of bipartite graphs that are possible using the 'vertex skeleton' $N_{x,y}$.

6.2.2 Theory

Lemma 6.1 An mvc-bipartite graph on $N_{x,y}$ must have at least $\max(x, y)$ edges.

Proof: There must be at least one edge incident at each vertex and hence $|E| \geq \max(x, y) \times 1$. 


Figure 6-1: Mvec-bipartite graph definition

Figure 6-2: $N_{2,3}$
Lemma 6.2 When an mvc-bipartite graph on $N_{x,y}$ has $\max(x,y)$ edges, the degree of each vertex lies between 1 and $\text{abs}(x - y) + 1$.

Proof:

- When $x > y$, $|E| = x$. Each top vertex has 1 edge incident. Each bottom vertex has $1 \leq \text{number of edges} \leq x - y + 1$ incident.

- When $x = y$, $|E| = x$. Each top vertex has 1 edge incident. Each bottom vertex has 1 edge incident.

- When $x < y$, $|E| = y$. Each top vertex has $1 \leq \text{number of edges} \leq y - x + 1$ incident. Each bottom vertex has 1 edge incident.

The result follows. \[ \blacksquare \]

When $|E| = \max(x,y)$ and $x > y$ (with a symmetric argument for $x < y$), the number of combinations of vertex degree for the bottom vertices is equal to the number of ways that $y$ natural numbers less than $x$ can add up to $x$ i.e.

$$n_1 + n_2 + n_3 + \ldots + n_y = x \quad n_p \in \mathbb{N}, 1 \leq p \leq y$$

e.g. the (bottom) vertex degree possibilities for an mvc-bipartite graph on $N_{4,3}$ are 2-1-1, 1-2-1 and 1-1-2. If $\forall i,j > 0$, $i,j \leq y$, $i \neq j$, $n_i \neq n_j$, then there are $y!$ permutations of these degrees. If however there are $m_1$ cases where $n_p$ appears in the series, $m_2$ cases where $n_q$ appears in the series ... and $m_t$ cases where $n_r$ appears in the series, the number of unique permutations is given by:

$$P = \frac{y!}{\prod_{j=1}^{t} m_j !}$$

Therefore if there are $t$ distinct sets of natural numbers less than $x$ that add up to $x$, and hence $q_1 \ldots q_t$ different permutations for the vertex degrees, and we denote by $r_k$ the number of different graphs that can be drawn for vertex permutation
CHAPTER 6. GRAPH DISPLAY

$q_k$, then the total number of graphs possible on $N_{x,y}$ with $\max(x,y)$ edges is given by:

$$ \text{total} = \sum_{s=1}^{t} q_s \cdot r_s $$

Lemma 6.3 An mvc-bipartite graph on $N_{x,y}$ can have at most $xy$ edges.

Proof: The maximum number of edges is reached when every vertex $x$ is connected to every vertex $y$ i.e. the graph is $K_{x,y}$. Therefore there are a total of $x \times y$ edges.

Lemma 6.4 The graph $K_n$ contains $\frac{1}{2}n(n-1)$ edges.

Proof: Each vertex in $K_n$ has $n-1$ edges incident. There are therefore a total of $n(n-1)$ edge incidences and hence $\frac{1}{2}n(n-1)$ edges.

Lemma 6.5 In $K_{x,y}$ there are at least $\min(x,y)$ edges incident at each vertex.

Proof: From the definition, there are $y$ edges incident at each top row vertex and $x$ edges incident at each bottom row vertex. Therefore each vertex has at least $\min(x,y)$ edges incident.

Lemma 6.6 In $K_{x,y}$ there are at most $\max(x,y)$ edges incident at each vertex.

Proof: Similar to Lemma 6.5.

Theorem 6.1 The number of edges in an mvc-bipartite graph lies in the range $\max(x,y) \leq |E| \leq xy$.

Proof: From the definition of mvc-bipartite graphs and Lemmas 6.1 and 6.3.

Theorem 6.2 A bipartite graph on $N_{x,y}$ cannot contain any vertices of degree 0 if $|E| > xy - \min(x,y)$.

Proof: Consider a graph $K_{x,y}$. The disconnection of a single vertex requires that all the incident edges must be removed i.e. the minimum number of edges which must be removed is $\min(x,y)$, from Lemma 6.5. From Lemma 6.3, a graph
CHAPTER 6. GRAPH DISPLAY

$G_{x,y}$ on $N_{x,y}$ has at most $xy$ edges and so if $G_{x,y}$ is to contain an isolated vertex, $|E|$ must be $\leq xy - \min(x, y)$. The result follows.

Theorem 6.3 $J_n$ is the set of graphs in $G_{x,y}$ with $n$ edges. The cardinality of $J_n$ is $^xyC_n$.

Proof: From Lemma 6.3 the maximum number of edges a graph in $J_n$ may have is $xy$. If a graph on $N_{x,y}$ with $n$ edges is constructed, then $n$ edges out of a possible $xy$ edges must be selected, and the result follows.

Corollary 6.1 In $J_n$, if $n < \max(x, y)$ then every graph $H \in J_n$ has at least one isolated vertex and hence there are $^xyC_n$ graphs of this form in the set.

Proof: Immediate from Lemma 6.1 and Theorem 6.3.

The graph $K_{x,y}$ will contain instances of $K_{a,b}$, $1 \leq a \leq x$, $1 \leq b \leq y$ as subgraphs. This leads to the following observation:

Lemma 6.7 The number of instances of the specific graph $K_{a,b}$ contained within $K_{x,y}$ where $1 \leq a \leq x$, $1 \leq b \leq y$ is given by:

$$\text{number of instances} = ^xC_a \times ^yC_b$$

Proof: By simple combinatorics.

Lemma 6.8 The total number of instances of all graphs of the form $K_{a,b}$ contained within $K_{x,y}$ where $1 \leq a \leq x$, $1 \leq b \leq y$ is given by the equation:

$$\text{total number of instances} = \sum_{a=1}^{x} \sum_{b=1}^{y} \text{number of } K_{a,b} \text{ in } K_{x,y}$$

$$= \sum_{a=1}^{x} \sum_{b=1}^{y} (^xC_a \times ^yC_b)$$

Proof: $K_{x,y}$ must by definition contain instances of $K_{x,y}$, $K_{x-1,y}$, ..., $K_{1,y}$, $K_{x,y-1}$, $K_{x-1,y-1}$, ..., $K_{1,y-1}$, ..., $K_{x,1}$, $K_{x-1,1}$, ..., and $K_{1,1}$. Using Lemma 6.7 the result follows.
6.2.3 Calculating $T^z_{x,y}$

In this section a calculation is presented for the enumeration of mvc-bipartite graphs. Defining $I^z_{x,y}$ as the number of graphs on $N_{x,y}$ with $z$ edges, where there exists at least one vertex with degree 0, then:

$$T^z_{x,y} = |G_{x,y} \text{ with } z \text{ edges} | - I^z_{x,y}$$

$$= x^y C_z - I^z_{x,y} \quad (6.1)$$

Informally, $I^z_{x,y}$ is the sum of the numbers of mvc-bipartite graphs with $z$ edges which can be constructed on all possible vertex skeletons contained as subsets of, but not including, $N_{x,y}$. Each of these graphs is unique and possesses at least one vertex of degree 0. Therefore:

$$I^z_{x,y} = \sum_{a=1}^{x} \sum_{b=1}^{y} \left( T^z_{a,b} \times \text{number of } K_{a,b} \text{ in } K_{x,y} \right)$$

$$- \left( T^z_{x,y} \times \text{number of } K_{x,y} \text{ in } K_{x,y} \right)$$

$$= \sum_{a=1}^{x} \sum_{b=1}^{y-1} \left( T^z_{a,b} \times \text{number of } K_{a,b} \text{ in } K_{x,y} \right)$$

$$+ \sum_{a=1}^{x-1} \left( T^z_{a,y} \times \text{number of } K_{a,y} \text{ in } K_{x,y} \right) \quad (6.2)$$

Substituting Equation 6.2 into Equation 6.1 gives:

$$T^z_{x,y} = x^y C_z - \sum_{a=1}^{x} \sum_{b=1}^{y-1} \left( T^z_{a,b} \times \text{number of } K_{a,b} \text{ in } K_{x,y} \right)$$

$$- \sum_{a=1}^{x-1} \left( T^z_{a,y} \times \text{number of } K_{a,y} \text{ in } K_{x,y} \right) \quad (6.3)$$

Mvc-bipartite graphs on an arbitrary vertex skeleton may therefore be enumerated using this recursive technique. Figure 6–3 shows some values of $T^z_{x,y}$ for low values of $x$ and $y$. 
6.2.4 Graph Planarity and Edge Crossings

A graph $G$ is planar if it is isomorphic to a graph $G'$, where all the vertices and edges of $G'$ are contained in the same plane and at most one vertex occupies or at most one edge passes through any point on that plane. $G'$ is therefore embedded in the plane and is a planar representation of $G$. The notation $\tilde{G}$ is used to represent an embedding of $G$.

$\tilde{G}$ divides the plane into connected regions called faces, each bounded by edges of the graph. Euler's formula, given in Theorem 6.4 below, relates the number of vertices, edges and faces in a connected planar graph. In the following, for a graph $\tilde{G}$, $n(G)$ represents the number of vertices, $e(G)$ the number of edges and $f(G)$ the number of faces.

**Theorem 6.4** If $G$ is a connected planar graph then for any $\tilde{G}$:

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

**Proof:** By induction on $f(G)$ – see for example [Gib85, p69].
Lemma 6.9 If \( d(f) \) is the number of edges bounding the face \( f \) and \( n(i) \) is the number of vertices of degree \( i \), then for for any \( G \):

\[
2e(G) = \sum_i d(f_i) = \sum_j jn(j)
\]

**Proof:** Each edge contributes to the degree of the vertices forming its endpoints, and the result follows.

Corollary 6.2 For any simple connected planar graph \( G \) with \( e(G) > 2 \), \( e(G) \leq 3n(G) - 6 \).

**Proof:** Each face of \( G \) is bounded by at least three edges and so \( \sum_i d(f_i) \geq 3f(G) \). The result follows by substituting into Theorem 6.4 and using Lemma 6.9.

Corollary 6.3 For any simple connected planar bipartite graph \( G \) with \( e(G) > 2 \), \( e(G) \leq 2n(G) - 4 \).

**Proof:** Each face of \( G \) is bounded by at least four edges and so \( \sum_i d(f_i) \geq 4f(G) \). The result follows by substituting into Theorem 6.4 and using Lemma 6.9.

Some standard results from graph theory can now be shown. The graph \( K_5 \) has five vertices and ten edges and so by Corollary 6.2 it cannot be planar. In a similar fashion, the graph \( K_{3,3} \) has six vertices and nine edges and so by Corollary 6.3 it cannot be planar.

These results are used in the derivation of Kuratowski's theorem, which is stated below in Theorem 6.5.

**Theorem 6.5** A graph is planar if and only if it has no subgraph homeomorphic to \( K_5 \) or \( K_{3,3} \).

**Proof:** See for example [Gib85, p77-80].
Figure 6-4: Planar constructions

Bipartite graphs

Theorem 6.6 The graphs $K_{1,n}, K_{2,n}, K_{n,1}$ and $K_{n,2}$, $\forall n \in \mathcal{N} - \{0\}$ are planar.

Proof: Figures 6-4(a) and 6-4(b) show constructions which can be used to draw planar representations of $K_{1,n}$ and $K_{2,n}$, $\forall n \in \mathcal{N} - \{0\}$ respectively. In a similar manner, these constructions can be used to draw $K_{n,1}$ and $K_{n,2}$.

Although a planar bipartite graph can be drawn with no edge crossings, the bipartite nature of the graph may be somewhat less apparent and so less information may be conveyed. The next sections consider whether a planar bipartite graph may be drawn with no edge crossings when the vertices are arranged on a two dimensional coordinate plane, with the top vertices on the line $Y = 1$ and the bottom vertices on the line $Y = 0$, henceforth referred to as the $Y$ drawing constraint.
Definitions

A bipartite graph $G$ is drawn under the $Y$ drawing constraint when:

- $G = (V, E)$, $V = L \cup R$, $L \cap R = 0$.

- $\forall v \in L$, $v$ is positioned on the horizontal line $Y = 1$ and has $X$ coordinate in the range $0..|L|$.

- $\forall v \in R$, $v$ is positioned on the horizontal line $Y = 0$ and has $X$ coordinate in the range $0..|R|$.

A directed bipartite graph $G$ contains an undirected cycle if the graph $G'$ formed from $G$ by making each edge undirected contains a cycle. The minimum number of vertices in an undirected cycle is defined as four.

**Lemma 6.10** An undirected cycle contains an even number of vertices.

**Proof**: If a cycle contained an odd number of vertices then the start vertex would lie in $L$ and the finish vertex would lie in $R$, or vice versa, contradicting the definition of a cycle.

**Theorem 6.7** A bipartite graph containing an undirected cycle, drawn under the $Y$ drawing constraint must contain at least one edge crossing.

**Proof**: Consider the following coordinate framework.
We start by drawing an edge from a point $l_1$ on $Y = 1$ to a point $r_1$ on $Y = 0$.

Next we add another edge from $r_1$ to another point on $Y = 1$. There are two choices: $l_2$ to the left of $l_1$ or $l_3$ to the right.

Next we draw another edge from $l_2$ or $l_3$ to another point on $Y = 0$.

At some time in this process an edge must be drawn from some $r_n$ on $Y = 0$ back to $l_1$ to complete the cycle. This edge must cross either the edge $l_2 - r_1$ or $l_3 - r_1$. Therefore the graph must contain at least one edge crossing.

It is important to note that there are other mvc-bipartite graphs that cannot be drawn without edge crossings under the Y drawing constraint, even if they do not contain a cycle. One such example is shown in Figure 6-5.

If an edge must cross another edge then it should cross only one other edge, in order to minimise the number of edge crossings. Considering general graphs
Figure 6–5: Graph that must contain an edge crossing

Drawn on the 2D coordinate plane, let $J$ denote the set of edges that are involved in edge crossings. Then:

1. if each edge crosses only one other edge then the number of edge crossings
   \[ \frac{|J|}{2} \]

2. if each edge crosses at least one other edge then the number of edge crossings
   \[ \geq \frac{|J|}{2} \]

3. if each edge crosses every other edge then the number of edge crossings
   \[ \frac{|J| \times (|J| - 1)}{2} \]

Therefore:
\[ \frac{|J|}{2} \leq \text{number of crossings} \leq \frac{|J| \times (|J| - 1)}{2} \]

A special construction can be used to render a cycle
\[ C_n = l_1 - r_1 - l_2 - r_2 - l_3 - r_3 \ldots l_n - r_n - l_1 \]
under the Y drawing constraint so that there are a maximum of $n-1$ edge crossings. The top vertices are arranged from left to right as $l_1l_2l_3\ldots$ and the bottom vertices as $r_n r_{n-1} r_2 \ldots$ as shown:
This vertex interleaving ensures that each edge crosses a maximum of one other edge. However neither the leftmost edge (the edge between the leftmost top vertex and the leftmost bottom vertex) nor the rightmost edge can be involved in an edge crossing, and so the total number of crossings is \( \frac{2n-2}{2} = n - 1 \).

Considering an arbitrary mvc-bipartite graph \( W_{x,y} \) rendered under the Y drawing constraint, at most \( xy - 2 \) edges can be involved in edge crossings by a similar argument. However the following results reveal a much tighter bound on the number of edge crossings in this type of graph.

**Lemma 6.11** The number of edge crossings in the graph \( K_{m,n} \) drawn under the Y drawing constraint is equal to the number of distinct instances of the subgraph \( K_{2,2} \) it contains.

**Proof:** The end-points of each pair of edges which cross in the graph \( K_{m,n} \) are by definition the vertices of a \( K_{2,2} \) subgraph. Therefore the total number of edge crossings must equal the number of distinct instances of \( K_{2,2} \) contained within the graph, i.e.

\[
\text{number of crossings} = \binom{n}{2} \times \binom{m}{2} = \frac{n!m!}{(n-2)!2!(m-2)!2!} = \frac{(n-1)n(m-1)m}{4}
\]

\[\blacksquare\]
Theorem 6.8 The number of possible edge crossings in an mvc-bipartite graph $W_{x,y}$ rendered under the Y drawing constraint lies in the range:

$$0 \leq \text{number of crossings} \leq \frac{(x-1)x(y-1)y}{4}$$

Proof: The least possible number of edge crossings is zero. The largest possible number of crossings will be generated when there are the maximum possible number of edges – this is by definition the case when $W_{x,y}$ is in fact $K_{x,y}$. By Lemma 6.11 the result follows.

Summary

The Y drawing constraint preserves hierarchical information in the bipartite graph at the expense of potential edge crossings. It is not possible to draw an arbitrary planar mvc-bipartite graph under this constraint without edge crossings, even with vertex rearrangement. Indeed the rearrangements that are possible may destroy some of the original information which was implied by vertex locality.

Non-planar Bipartite Graphs

Non-planar mvc-bipartite graphs must now be considered. The inherent complexity of this type of graph usually makes it exceptionally difficult to render so that the maximum possible amount of connectivity information is revealed. The number of edge crossings that may be present in such graphs (e.g. Figure 6-6) may lead to the finished result having very little discernible structure.

The clustering technique depicted in Figure 6-7 may be used to great effect in this case. Here a non-planar complete bipartite graph is transformed into an equivalent planar graph representation using an “edge multiplexer” or Type I Edge
CHAPTER 6. GRAPH DISPLAY

The intended meaning of this node is that there is a path from each 'upper' vertex to each 'lower' vertex. The planar graph therefore contains the same connectivity information as the non-planar graph, but in a form that is more readily understandable and which clearly reveals the inherent structure.

Figure 6–8 depicts the corresponding transformation of the undirected non-planar graph $K_6$ to a planar representation using a Type II Edge Reduction Node. Here the meaning of the node is that there is a path from each vertex to every other vertex.

Figure 6–9 shows the graph of Figure 6–6 after an edge reduction using two Type I nodes.

The following section will describe several strategies which may be adopted when attempting to apply this clustering technique to arbitrary bipartite graphs.

---

Reduction Node. The intended meaning of this node is that there is a path from each 'upper' vertex to each 'lower' vertex. The planar graph therefore contains the same connectivity information as the non-planar graph, but in a form that is more readily understandable and which clearly reveals the inherent structure.

Figure 6–8 depicts the corresponding transformation of the undirected non-planar graph $K_6$ to a planar representation using a Type II Edge Reduction Node. Here the meaning of the node is that there is a path from each vertex to every other vertex.

Figure 6–9 shows the graph of Figure 6–6 after an edge reduction using two Type I nodes.

The following section will describe several strategies which may be adopted when attempting to apply this clustering technique to arbitrary bipartite graphs.

---

1The Type I edge reduction node, although rendered differently, has the same functional specification as Newberry's edge concentration node.
CHAPTER 6. GRAPH DISPLAY

Figure 6-7: The Type I Edge Reduction Node

Figure 6-8: The Type II Edge Reduction Node

Figure 6-9: Reduced graph
6.3 Bipartite Graph Information Content

For general graphs, the use of the Type I clustering technique requires that the vertices are already allocated to levels within the graph, using an algorithm such as that presented by Warfield [War76]. The resulting hierarchy gives some semantic context to the bipartite graphs formed between the levels, and the edge reduction node clusters attempt to refine this to emphasise the stronger relationships between the vertices. This clustering may imply some sort of non-existent semantic relationship to the user, similar to the effect of using a syntactic partition in a graph with a natural semantic partition as previously discussed in Chapter 2. However it may also imply a real and important relationship. In both cases, the use of the technique may be justified as it usually makes the graph as a whole easier to understand by reducing the number of edge crossings, while the meaning of the resulting clusters can determined by the user of the application.

6.3.1 Motivation for edge reduction algorithms

The goal of an edge reduction algorithm for a bipartite graph is to detect the presence of complete bipartite subgraphs and replace them with the equivalent edge reduction node subgraphs. There are many strategies that might be adopted depending on the underlying motivation, as described in the following sections.

Reduction of the number of edge crossings to zero

If a bipartite graph does not contain any subgraphs homeomorphic to $K_{3,3}$ then it is planar and can be drawn with no edge crossings [Gib85]. Therefore, given a graph where the only subgraphs homeomorphic to $K_{3,3}$ are actual instances of $K_{3,3}$, these subgraphs could be edge reduced and Woods’ algorithm [Woo82] used to draw the graph with no edge crossings. An improvement to this approach would
be to attempt to edge reduce all instances of $K_{3,3}$ or larger, as this may reduce
the required number of edge reduction nodes and hence the graph complexity.

This approach relies heavily on the layout algorithm to produce an aesthetically
pleasing result. However the goal of planarity may inevitably conflict with
a desire to keep certain nodes close together. If node adjacency actually con-
veys some information about the graph then although graph legibility may have
been improved, structural information may have been sacrificed in doing so. The
advantages and disadvantages of this approach would have to be considered care-
fully for each graph that was processed. Also, the class of graphs for which the
technique is suitable is very limited.

Minimum edges in the final graph

Newberry defined the 'optimal' set of edge reductions to be the set that would give
the smallest number of edges in the final graph. The fundamental idea behind this
approach is that by reducing the number of edges, the number of edge crossings
is often reduced as a side effect. Clearly another major factor is that with fewer
edges, the graph is less complicated and hence conveys the underlying information
better.

Production of the best structural summary

Another possible approach is to try to produce the best structural summary of
the graph in question. A careful definition of the meaning of best in this context
is required. It might mean that all the complete subgraphs should be represented
in the final graph, or the subset of complete subgraphs which produces the least
number of edge crossings, or the subset of complete subgraphs which can be laid
out to convey the maximum amount of structural information etc.
Detection of the largest substructures

This goal might be desired in certain circumstances if it conveys the most appropriate information for a certain problem, for example when the largest common subset of `#include` files must be found in a C program composed of many small files. Issues such as producing the least number of edge crossings and conveying the desired structural information would also have to be considered in this context.

Best summary in given time

The determination of the “best” possible set of edge reductions might take a long time. In some cases the best possible is not actually required and a less optimal solution would give acceptable results e.g. data structure layout in program debugging. Here heuristics might be used to attempt to produce a reasonable structural summary of the graph.

6.3.2 Detecting complete bipartite subgraphs

There are three basic approaches to detecting complete bipartite subgraphs in bipartite graphs:

1. exhaustive searching

2. detection using graph theoretical methods

3. cluster detection and pattern matching

Exhaustive searching will be considered in the next section. Chapter 7 will describe a graph theoretical method for complete subgraph detection, while Chapter 8 will describe the cluster detection and pattern matching approach.
Exhaustive searching

It is possible to detect every complete bipartite subgraph if a test is made for the existence of the appropriate edges between every set of nodes in the graph that may form a complete subgraph. However this approach will not work in polynomial time as will be demonstrated below.

**Definition 6.1**

\[
\binom{x}{p} = \frac{x!}{p!(x-p)!}
\]

**Definition 6.2**

\[
E_x = \binom{x}{2} + \binom{x}{3} + \ldots + \binom{x}{x-1} + \binom{x}{x}
\]

The maximum number of complete bipartite subgraphs with at least two \( x \) and \( y \) vertices which a graph on \( N_{x,y} \) may contain is given by:

\[
C_{max} = \sum_{a=2}^{x} \sum_{b=2}^{y} \binom{x}{a} \times \binom{y}{b}
\]

\[
= \sum_{a=2}^{x} \sum_{b=2}^{y} \binom{x}{a} \binom{y}{b}
\]

Consider the following set of terms from this expression:

\[
\sum_{a=2}^{x} \binom{x}{a} = \binom{x}{2} + \binom{x}{3} + \ldots + \binom{x}{x-1} + \binom{x}{x} = E_x
\]

Now \( \sum_{a=0}^{x} \binom{x}{a} = \binom{x}{0} + \binom{x}{1} + E_x = 2^x \)

\[\Rightarrow E_x = 2^x - x - 1\]
This means that the number of possible complete bipartite subgraphs increases exponentially with $x$, so an exhaustive search and test strategy is not viable for large values of $x$. Other methods, such as those described in the next two chapters, are therefore required.
Chapter 7

Graph Theoretical Methods

If the graph $K_{2,2}$ is detected when it is in fact part of a larger $K_{3,2}$, at least one edge crossing which could have been eliminated will always remain, as illustrated by Figure 7-1 where all possible positions of vertex 'c' are enumerated. Extending this observation to larger subgraphs, it should always be worthwhile to search for the largest complete bipartite subgraph in order to eliminate the greatest possible number of edge crossings at once. This will also have the effect of revealing the largest clusters, which may have some significance for the user.

One possible approach to applying the edge reduction technique to a graph might be to find a complete bipartite subgraph, edge reduce it and repeat the

Figure 7-1: $K_{2,2}$ reduced instead of $K_{3,2}$
process until no further reducible subgraphs remain. By the above argument, it would seem useful to search for the largest remaining complete bipartite subgraph each time, in order to eliminate the greatest possible number of edge crossings in each pass.

7.1 Determining the presence of a $K_{x,y}$ subgraph

**INSTANCE** Bipartite graph $G = (V, E)$, $V = L \cup R$, $L \cap R = 0$, positive integer $S \leq |E|$.  

**QUESTION**: Are there two disjoint subsets $V_1, V_2 \subseteq V$ such that $|V_1| + |V_2| = S$ and such that $u \in V_1, v \in V_2$ implies that $\{u, v\} \in E$?

The related problem where the condition $|V_1| + |V_2| = S$ is replaced by $|V_1| = |V_2| = S$ is NP complete [GJ79]. [GJ79] and [Joh76] suggest a polynomial time algorithm for the former problem which will be presented shortly. However it will be shown that this technique does not work in the sense required for all possible bipartite graphs, contrary to what otherwise might be expected.

**Definitions**

A subset of a graph's vertices is called an **independent set** if no two vertices in the subset are adjacent.

A **vertex cover** is a subset of a graph's vertices such that for every edge in the graph, at least one of its end-points is contained within the subset.

A **matching** of a graph is a subset of its edges where no two members of the subset are adjacent. A **maximum cardinality matching** $M$ is a matching containing a maximum number of edges, i.e. for any matching $M'$, $|M| \geq |M'|$. A **perfect matching** occurs when every vertex of the graph is an end-point of an
exists a complete bipartite subgraph of size $S$ in $G$

$\iff$

exists an independent set of size $S$ in $\bar{G}$

$\iff$

exists a vertex cover of size $n - S$ in $\bar{G}$

$\iff$

exists a maximum matching of size $n - S$ in $\bar{G}$

Figure 7-2: Problem transformation sequence

edge in the matching. A matched edge is an edge in a matching. A matched vertex is an end-point of a matched edge.

The problem of determining the existence of a complete bipartite subgraph of size $S$ in a bipartite graph $G$ may be recast as the problem of finding a maximum cardinality matching $M$ as shown in Figure 7-2. Maximum cardinality matchings may be found in polynomial time using network flow techniques [Gib85,CLR90]. The following sections formally present network flow theory and the maximum flow problem, together with Ford and Fulkerson’s classical method for finding a maximum matching in a bipartite graph.

7.1.1 Network flow Theory

A flow network $G = (V, E)$ is a directed graph in which each edge $(u, v) \in E$ has an associated capacity $c(u, v) \geq 0$. If $(u, v) \notin E$ then $c(u, v) = 0$. There is a source vertex $s$ and a sink vertex $t$, and for every vertex $v \in V$ there is a path $s \leadsto v \leadsto t$. 
A flow in $G$ is a function $f : V \times V \to \mathbb{R}$ that satisfies the following three properties:

**Capacity constraint:** \( \forall u,v \in V, f(u,v) \leq c(u,v) \)

**Skew symmetry:** \( \forall u,v \in V, f(u,v) = -f(v,u) \)

**Flow conservation:** \( \forall u \in V - \{s,t\}, \sum_{v \in V} f(u,v) = 0 \)

Therefore the net flow from vertex $u$ to vertex $v$ is given by $f(u,v)$. The value of a flow $f$ is given as $|f| = \sum_{v \in V} f(s,v) - \sum_{v \in V} f(v,s)$, the total net flow out of the source.

Given a flow network $G = (V,E)$ and a flow $f$, consider a pair of vertices $u,v \in V$. The amount of additional net flow that can be pushed from $u$ to $v$ without exceeding the capacity $c(u,v)$ is called the residual capacity of $(u,v)$, given by $c_f(u,v) = c(u,v) - f(u,v)$. For example, if $c(u,v) = 10$ and $f(u,v) = 4$ then $c_f(u,v) = 6$, so six more units of flow may be pushed along edge $\{u,v\}$. Also, if $c(u,v) = 10$ and $f(u,v) = -4$ then $c_f(u,v) = 14$.

The residual network of $G$ induced by $f$ is:

$$G_f = (V,E_f), \text{ where } E_f = \{(u,v) \in V \times V : c_f(u,v) > 0\}$$

Therefore each residual network edge $(u,v)$ can take on an additional positive net flow, given by $c_f(u,v) = c(u,v) - f(u,v)$.

**Definition 7.1** Given a flow network $G$ and functions $f_1, f_2 : V \times V \to \mathbb{R}$, the flow sum $f_1 + f_2$ is the function from $V \times V$ to $\mathbb{R}$ defined by:

$$(f_1 + f_2)(u,v) = f_1(u,v) + f_2(u,v)$$
Lemma 7.1 If \( f' \) is a flow in \( G_f \) then the flow sum \( f + f' \) is a flow in \( G \) with value \( |f + f'| = |f| + |f'| \).

Proof:

Skew symmetry: \( \forall u, v \in V \),

\[
(f + f')(u, v) = f(u, v) + f'(u, v) = -f(v, u) - f'(v, u) = -(f(v, u) + f'(v, u)) = -(f + f')(v, u)
\]

Capacity constraint: \( \forall u, v \in V \), \( f'(u, v) \leq c_f(u, v) \), so

\[
(f + f')(u, v) = f(u, v) + f'(u, v) \leq f(u, v) + (c(u, v) - f(u, v)) = c(u, v)
\]

Flow conservation: \( \forall u \in V - \{s, t\} \),

\[
\sum_{u \in V} (f + f')(u, v) = \sum_{u \in V} (f(u, v) + f'(u, v)) = \sum_{u \in V} f(u, v) + \sum_{u \in V} f'(u, v) = 0
\]

Therefore: \( |f + f'| = \sum_{u \in V} (f + f')(s, v) \)

\[
= \sum_{u \in V} (f(s, v) + f'(s, v)) = \sum_{u \in V} f(s, v) + \sum_{u \in V} f'(s, v) = |f| + |f'|
\]

An augmenting path \( p \) is a simple path from \( s \) to \( t \) in the residual network \( G_f \). The residual capacity of \( p \), \( c_f(p) = \min \{c_f(u, v) : (u, v) \text{ is on } p\} \), is the maximum amount of net flow that can be pushed along the edges of \( p \).
Lemma 7.2 Given a flow network $G = (V, E)$ and a flow $f$ in $G$, let $p$ be an augmenting path in $G_f$. Define a function $f_p : V \times V \to \mathbb{R}$ as follows:

$$f_p(u,v) = \begin{cases} 
c_f(p) & \text{if } (u,v) \text{ is on } p \\
-c_f(p) & \text{if } (v, u) \text{ is on } p \\
0 & \text{otherwise}
\end{cases}$$

Then $f_p$ is a flow in $G_f$ with $|f_p| = c_f(p) > 0$.

Corollary 7.1 Given a flow network $G = (V, E)$, a flow $f$ in $G$, an augmenting path $p$ in $G_f$, and function $f_p$ as defined in Lemma 7.2, define a function $f' : V \times V \to \mathbb{R}$ by $f' = f + f_p$. Then $f'$ is a flow in $G$ with value $|f'| = |f| + |f_p| > |f|$.

Proof: Immediate from Lemmas 7.1 and 7.2.

A cut $(S, T)$ of a flow network $G = (V, E)$ is a partition of $V$ into $S$ and $T = V - S$ such that $s \in S$ and $t \in T$. If $f$ is a flow, the net flow across the cut is given by $f(S, T)$ and the capacity by $c(S, T)$.

Theorem 7.1 For any cut $(S, T)$ of the network $G$, the value of the flow is given by:

$$|f| = \sum_{u \in S} \sum_{v \in T} f(u, v) - \sum_{u \in T} \sum_{v \in S} f(u, v)$$

$$= (\text{flow from } S \text{ to } T) - (\text{flow from } T \text{ to } S)$$

Proof:

By definition: $|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$

For $u \in S - \{s\}$, $0 = \sum_{v \in V} f(u, v) - \sum_{v \in V} f(v, u)$

Summing these two equations over all $u \in S$:

$$|f| = \sum_{u \in S} \left(\sum_{v \in V} f(u, v) - \sum_{v \in V} f(v, u)\right)$$
CHAPTER 7. GRAPH THEORETICAL METHODS

Now \[ \sum_{u \in S} \sum_{v \in V} f(u, v) = \sum_{u \in S} \sum_{v \in S} f(u, v) + \sum_{u \in S} \sum_{v \in T} f(u, v) \]
and \[ \sum_{u \in S} \sum_{v \in V} f(v, u) = \sum_{u \in S} \sum_{v \in S} f(v, u) + \sum_{u \in S} \sum_{v \in T} f(v, u) \]
Also \[ \sum_{u \in S} \sum_{v \in S} f(u, v) = \sum_{u \in S} \sum_{v \in S} f(u, v) \]
and so the theorem follows.

Corollary 7.2 The value of \(|f|\) for any network is bounded by the capacity of any cut \((S, T)\).

Proof:

For any cut \((S, T)\), \[ |f| = \sum_{u \in S} \sum_{v \in T} f(u, v) - \sum_{u \in T} \sum_{v \in S} f(u, v) \]
\[ \leq \sum_{u \in S} \sum_{v \in T} c(u, v) - \sum_{u \in T} \sum_{v \in S} f(u, v) \]
\[ = c(S, T) - \sum_{u \in T} \sum_{v \in S} f(u, v) \]
\[ \leq c(S, T) \]

The maximum flow problem is that of finding a flow of maximum value from \(s\) to \(t\) in a flow network \(G\).

Theorem 7.2 (Max-flow min-cut) Given a flow network \(G = (V, E)\) with source \(s\) and sink \(t\) together with a flow \(f\), the following conditions are equivalent:

1. \(f\) is a maximum flow in \(G\).

2. The residual network \(G_f\) does not contain any augmenting paths.

3. \(|f| = c(S, T)\) for some cut \((S, T)\) of \(G\).
Proof:

(1) $\Rightarrow$ (2): Hypothesis: $f$ is a maximum flow in $G$. Assume that $G_f$ has an augmenting path $p$. By Corollary 7.1, the flow sum $f + f_p$ is a flow in $G$ with value greater than $|f|$, which contradicts the hypothesis that $f$ is a maximum flow.

(2) $\Rightarrow$ (3): Assume that $G_f$ has no augmenting path. Next define the set $S = \{v \in V :$ there exists a path from $s$ to $v$ in $G_f\}$, and $T = V - S$. This partition $(S, T)$ is a cut with $s \in S$ and $t \notin S$ (since there is no path from $s$ to $T$ in $G_f$).

Now, for each pair of vertices $u$ and $v$ where $u \in S$ and $v \in T$, $f(u, v) = c(u, v)$, since otherwise $(u, v) \in E_f$ and $v \in S$. By Theorem 7.1, $|f| = f(S, T) = c(S, T)$.

(3) $\Rightarrow$ (1): By Corollary 7.2, $|f| \leq c(S, T)$ for all cuts $(S, T)$. Therefore if $|f| = c(S, T)$ then $f$ is a maximum flow.

7.1.2 The Ford-Fulkerson method

The classical iterative Ford-Fulkerson method depends on the basic ideas of augmenting paths, residual networks and cuts, and may be implemented in different ways with different run-time complexities. The general strategy is shown in Figure 7-3. Starting with $\forall u, v \in V, f(u, v) = 0$, giving an initial zero flow, at each iteration the flow value is increased by finding an augmenting path, then augmenting the flow along this path. The process is repeated until no further augmenting path can be found. The max-flow min-cut theorem shows that when the process terminates, a maximum flow has been found.

Figure 7-4 shows a basic implementation of the method. At each iteration, any augmenting path $p$ is found and the flow is augmented by the residual capacity $c_f(p)$. The net flow $f[u, v]$ between each pair of vertices $u, v \in V$ connected by an edge is updated.
initialise flow $f$ to 0
while there exists an augmenting path $p$ do
    augment flow $f$ along $p$
return $f$

Figure 7–3: Ford-Fulkerson Method

for each edge $(u, v) \in E[G]$ do
begin
    $f[u, v] := 0$
    $f[v, u] := 0$
end
while there exists a path $p$ from $s$ to $t$ in the residual network $G_f$ do
begin
    $c_f(p) := \min \{ c_f(u, v) : (u, v) \text{ is in } p \}$
    for each edge $(u, v)$ in $p$ do
        begin
            $f[u, v] := f[u, v] + c_f(p)$
            $f[v, u] := -f[u, v]$
        end
end

Figure 7–4: Ford-Fulkerson Algorithm
Maximum bipartite matchings

The Ford-Fulkerson method can be used to find a maximum matching in a bipartite graph in polynomial time. A bipartite flow network $G' = (V', E')$ is constructed for the bipartite graph $G = (V, E)$ by creating source and sink vertices such that $V' = V \cup \{s, t\}$. If $V = L \cup R$, then

$$E' = \{(s, u) : u \in L\} \cup \{(u, v) : u \in L, v \in R \text{ and } (u, v) \in E\} \cup \{(v, t) : v \in R\}$$

Unit capacity is assigned to each edge in $E'$.

**Definition 7.2** A flow $f$ on $G = (V, E)$ is **integer-valued** if:

$$\forall (u, v) \in V \times V, f(u, v) \in \mathbb{Z}$$

**Lemma 7.3** Let $G = (V, E)$ be a bipartite graph with $V = L \cup R$ and let $G' = (V', E')$ be the corresponding flow network. Then:

$M$ is a matching in $G \Leftrightarrow$ there is an integer-valued flow $f$ in $G'$ with $|f| = |M|$.

**Proof**: Define $f$ as follows: if $(u, v) \in M$ then $f(s, u) = f(u, v) = f(v, t) = 1$ and $f(u, s) = f(v, u) = f(t, v) = -1$. For all other edges $(u, v) \in E'$, $f(u, v) = 0$.

Note that $f$ can be obtained by flow augmentation along each of the paths induced by edges in $M$ and hence satisfies skew symmetry, the capacity constraints and flow conservation. The net flow across the cut $(L \cup \{s\}, R \cup \{t\}) = |M|$ and so by Theorem 7.1 $|f| = |M|$.

Now let $f$ be an integer-valued flow in $G'$ and let

$$M = \{(u, v) : u \in L, v \in R \text{ and } f(u, v) > 0\}$$
Each vertex \( u \in L \) has only one edge that enters it, \((s, u)\) with capacity 1. For each \( u \in L \), one unit of positive net flow enters \( u \) iff there is exactly one vertex \( v \in R \) such that \( f(u, v) = 1 \). Therefore at most one edge leaving each \( u \in L \) has positive net flow. A similar statement can be made for each \( v \in R \) and hence the set \( M \) is a matching.

For every matched vertex \( u \in L \), \( f(s, u) = 1 \), and for every edge \((u, v) \in E - M\), \( f(u, v) = 0 \). Therefore:

\[
|M| = \sum_{l \in L} \sum_{r \in R} f(l, r)
\]

\[
= \sum_{l \in L} \sum_{v \in V'} f(l, v) - \sum_{l \in L} \sum_{l' \in L} f(l, l') - \sum_{l \in L} f(l, s) - \sum_{l \in L} f(l, t)
\]

\[
= 0 - 0 + \sum_{l \in L} f(s, l) - 0
\]

\[
= \sum_{v \in V'} f(s, v)
\]

\[
= |f|
\]

\[\blacksquare\]

**Theorem 7.3** In the Ford-Fulkerson method, if the capacity function \( c \) returns only integer values then the value of the maximum flow \( |f_{\text{max}}| \) is an integer, and for all vertices \( u \) and \( v \), \( f(u, v) \) is an integer.

**Proof:** By induction on the number of iterations \((i)\). Firstly, define \( c \) as the function \( c(u, v) : V \times V \rightarrow \mathcal{I} \).

Let \( i = 0 \) (i.e. there is no path \( p \) from \( s \) to \( t \) in \( G_f \)). Clearly \(|f| = 0 \) and \( \forall u, v \in V, f(u, v) = 0 \). \( 0 \in \mathcal{I} \Rightarrow \) the theorem is true for \( i = 0 \).

Assume that the theorem is true for \( i = n \) and consider \( i = n + 1 \). If there is a path \( p \), then \( c_f = \min \{ c_f(u, v) : (u, v) \text{ is in } p \} \). Now, we know that \( c_f(u, v) \in \mathcal{I} \) and so as \( \min \{ c_f(u, v) \in \mathcal{I} \} \), \( c_f(p) \in \mathcal{I} \). This means that the new values for \( f[u, v] \in \mathcal{I} \) and hence that \(|f| \in \mathcal{I} \).

Therefore the theorem is true \( \forall i \in \mathcal{N} \). \[\blacksquare\]
This theorem is used to prove the following vital corollary to Lemma 7.3.

**Corollary 7.3** The cardinality of a maximum matching in a bipartite graph $G$ is the value of a maximum flow in its corresponding flow network $G'$.

**Proof:** Let $M$ be a maximum matching in $G$ and suppose that the corresponding flow $f$ is not maximum. Therefore there is a maximum flow $f'$ in $G'$ with integer capacity such that $\|f'\| > \|f\|$ and so $f'$ corresponds to a matching $M'$ in $G$ with cardinality $|M'| = \|f'\| > |M|$, and so $M$ cannot be a maximum matching.

Similarly it can be shown that if $f$ is a maximum flow in $G'$ the corresponding matching is a maximum matching on $G$. 

---

**Run-time complexity**

With integral capacities, the Ford-Fulkerson implementation of Figure 7–4 has run-time complexity $O(E \cdot f_{\text{max}})$, where $f_{\text{max}}$ is the maximum flow (the edge initialisation code runs in $O(E)$ and the while loop executes at most $\lfloor f_{\text{max}} \rfloor$ times, as on each iteration the flow increases by at least one unit). Now the cardinality of a bipartite graph matching is at most $\min(|L|, |R|) = O(V)$ and so $\lfloor f_{\text{max}} \rfloor$ in $G'$ is $O(V)$. Therefore a maximum matching in a bipartite graph can be found in time $O(V E)$.

Using the transformations of Figure 7–2, it is possible to detect the presence of a complete bipartite subgraph of size $S$ in $G$ in time $O(V E)$, by finding the maximum matching in $\overline{G}$. Unfortunately the problem of detecting the subgraph is much easier than the problem of actually determining the vertices which compose it, as will be demonstrated in the following sections.
7.2 Finding a $K_{x,y}$ subgraph

INSTANCE: Bipartite graph $G = (V, E)$, $V = L \cup R$, $L \cap R = 0$, positive integer $S \leq |E|$.

QUESTION: Can two disjoint subsets $V_1, V_2 \subseteq V$ be found such that $|V_1| + |V_2| = S$ and $u \in V_1$, $v \in V_2$ implies that $\{u, v\} \in E$?

The basic detection algorithm given in Figure 7-2 will now be extended in an attempt to actually find the largest possible $K_{x,y}$ subgraph. Indeed it is only when an attempt is made to find this subgraph that the flaw in the detection algorithm becomes apparent.

7.2.1 Extended algorithm and observations

Figure 7-5 presents an overview of the extended algorithm. Firstly, the maximum matching in the bipartite graph is determined using the previously described methods. König's theorem [Kon31] states that the number of edges in a maximum matching on a bipartite graph is equal to the minimum number of vertices in a vertex cover, and so each edge in the matching must have one of its associated vertices in the vertex cover. For example Figure 7-6 shows a matching of size three, giving a vertex cover of size three, an independent set of size five and thus a complete bipartite subgraph of size five.

Considering each vertex $v \in L$, if there is no matching edge incident, then $v$ cannot be in the vertex cover. If, by following an edge not in the matching from this vertex, a vertex $w \in R$ can be reached then $w$ must be in the vertex cover. If a matching edge can be followed in the reverse direction, back to a vertex $u \in L$ then $u$ cannot be in the vertex cover (since only one vertex belonging to an edge in the matching may be in the vertex cover).
CHAPTER 7. GRAPH THEORETICAL METHODS

begin

Determine the maximum matching in the flow network as before
Search(s,L,R)
Search(t,R,L)

end

Search (start-vertex, S1, S2):

For each edge of the form (start-vertex, \( v \in S1 \)) do begin
If there is no matching edge incident at \( v \) then begin
Label \( v \) as 'not in the vertex cover'
Searchfrom(\( v, S1, S2 \))
end
end

Searchfrom(\( v,S1, S2 \)):

If there exists an edge not in the matching connecting \( v \in S1 \) to an unlabelled vertex \( w \in S2 \) then begin
Label \( w \) as 'in the vertex cover'
Traverse the edge in the matching incident at \( w \) to vertex \( x \in S1 \)
If \( x \) is unlabelled then begin
Label \( x \) as 'not in the vertex cover'
Searchfrom(\( x, S1, S2 \))
end
end

Figure 7–5: Vertex Labelling Algorithm Overview
Unfortunately, this process may not visit and label all of the vertices in the graph, as will now be shown. The flow paths associated with the matching are obviously vertex disjoint and hence the situation shown in Figure 7–7(a) can never arise. If there is an edge, not in the matching, from $s$ to a vertex $e \in L$ or from $t$ to a vertex $e \in R$, then the vertices will be labelled according to the algorithm, as shown in Figure 7–7(b). Figures 7–7(c) and 7–7(d) also show cases where the vertices will be labelled according to the algorithm. Figure 7–7(e) shows the remaining problem case. Vertices $X$ and $Y$ may remain unlabelled iff the search from $s$ does not reach $Y$ and the search from $t$ does not reach $X$ (backtracking along the matching edge will label the other vertex in both cases).

So, defining $V_L \subseteq L$, $V_R \subseteq R$, $v_l$ as any vertex $e \in V_L$, and $v_r$ as any vertex $e \in V_R$, $X$ and $Y$ will remain unlabelled iff there is no path:

$$s \xrightarrow{\text{edge}} v_l \left( u \xrightarrow{\text{edge}} v_r \xrightarrow{\text{edge}} v_l \right)^* \text{ or}$$
A depth or breadth first search strategy will examine every possible path of this form. The case analysis shows that unlabelled vertices may only occur when \( X \) and \( Y \) are the end vertices of an edge in the matching, and so once the search is complete it suffices to check only the vertices associated with each edge in the matching for an unlabelled state. If the pair of vertices are indeed unlabelled, one vertex of the two should be selected and placed in the vertex cover.

When all \( m \) edges in the matching have unlabelled vertices, the upper bound on the number of possible vertex covers, for the subgraph consisting of the unlabelled
vertices and the edges which connect them, is $2^m$. Figure 7–8 illustrates a case where this upper bound is not obtained and there are only six instead of eight vertex covers. This means that a random selection of one vertex from a pair, to be placed in the total vertex cover, is not guaranteed to be correct.

**Lemma 7.4** The graph $\overline{G_{x,y}}$ formed by inverting an mvc-bipartite graph $G_{x,y}$ may have a maximum matching of size $\min(x, y)$.

**Proof:** Each edge in a maximum matching is vertex disjoint.

**Lemma 7.5** The set of vertices $\in L$ or the set of vertices $\in R$ form a vertex cover for a bipartite graph.

**Proof:** Every edge in the graph has a vertex $\in L$ and a vertex $\in R$.

**Theorem 7.4** The vertex cover for the subgraph consisting of the unlabelled vertices and the edges that connect them, together with the vertex cover found by the search algorithm for the labelled vertices, will form a vertex cover for the entire graph.

**Proof:** Any edge from a labelled to an unlabelled vertex cannot be in the matching (as a matching edge will be incident at the unlabelled vertex already). The depth or breadth first search was clearly unable to traverse this edge, which means that the labelled vertex must have been on the opposite side of the graph from the start point (if it had been on the same side then the edge would have been traversed and there would have been no unlabelled vertices). Therefore the labelled vertex is in the vertex cover. This means that all edges from labelled vertices to labelled vertices, and all edges from labelled vertices to unlabelled vertices, will be covered by the vertex cover found by the search algorithm. The only edges that remain uncovered are those between unlabelled vertices. Once the vertex cover for the subgraph consisting of the unlabelled vertices and the edges from unlabelled vertices to unlabelled vertices has been determined, the union of the two vertex covers forms a vertex cover for the whole graph.
Using Lemma 7.5, a vertex cover for the unlabelled subgraph could be obtained by selecting all the \( L \) or \( R \) vertices. Figure 7–8 shows that simply selecting one vertex from each pair of the edges in the matching is not guaranteed to form a vertex cover. The wide-ranging effects of these observations on the design of the new algorithm will now be considered.

7.2.2 The Unlabelled Vertex Subgraph Problem

The search algorithm must clearly be augmented to be able to determine a vertex cover for the unlabelled vertex subgraph. There are two possible cases:

(a) the graph consists entirely of unlabelled vertices.

(b) the unlabelled vertices graph is a subgraph of the main graph.

The following sections will describe case (a), deriving several results concerning the minimum problem size and the numbers of possible vertex covers for graphs of this type. A later section will briefly consider case (b). Conclusions on the effectiveness of the graph theoretical method as a whole will then be drawn.
(a) Graph consists entirely of unlabelled vertices

A random choice of graph vertices may not form a vertex cover when there exists at least one edge, not in the matching, between unlabelled vertices. In this case at least one of the vertices of each of these edges is constrained to lie in the vertex cover. If it is the case that the degree of the vertices of an edge in the matching is one, then either vertex may be selected for the vertex cover.

Consider Figure 7-9. Example (a) illustrates an application of the edge reduction nodes to a graph $U$. Examples (b) and (c) show how the number of edge crossings changes when one edge is removed from $U$ (and hence one non-matching edge is added to $\overline{U}$). This type of edge may therefore lead to either:

- **more** crossings in the final graph if the use of an edge reduction node is prevented or
- **less** crossings in the final graph if an edge that would have caused extra crossings is removed.

It is important to note that by selecting all $L$ or all $R$ vertices as the vertex cover, all the possible complete bipartite subgraphs will not be found. In this case the algorithm will not be able to reduce the number of edge crossings by the amount that would otherwise be possible.

**Minimum problem size**

The minimum size of unlabelled graph $U_{m,m}$ which can contain a complete bipartite subgraph will now be determined. Firstly, note that in $\overline{U_{m,m}}$, no vertex may have degree $m$ as this would mean that in $U_{m,m}$ the vertex had degree zero and hence the graph would not satisfy the mvc-bipartite definition. Also, no vertex in $\overline{U_{m,m}}$ may have degree zero since it would therefore already be labelled by the search algorithm. Therefore $\min(d(v \in U_{m,m})) = 1$ and $\max(d(v \in U_{m,m})) = m - 1$. If the
Figure 7-9: Example applications
smallest usefully detected complete bipartite subgraph is $K_{2,1}$ then the minimum problem size is $K_{2,1}$ contained within the graph $U_{3,3}$, as shown in Figure 7-10.

In general $U_{m,m}$ may contain instances of $K_{1,m-1}$, $K_{2,m-2}$, ..., $K_{m-2,2}$ or $K_{m-1,1}$. The heuristic-based selection of an appropriate vertex cover is clearly difficult, even though the numbers of such potential vertex covers can be determined as will be shown in the next section.

**Maximum and minimum numbers of vertex covers**

$H_{m,m}^k$ is defined as the set of unlabelled vertex graphs $\overline{U_{m,m}}$ with $k$ edges. Each graph $\in H_{m,m}^k$ contains a perfect matching, and $m \leq k \leq m(m-1)$. Figure 7-11 shows the set $H_{3,3}^4$.

For graph set $H_{m,m}^k$ there exist upper ($V_{max}$) and lower ($V_{min}$) bounds on the number of possible vertex covers for each member graph. These values may be determined by graphical enumeration – some examples are shown in Figure 7-12. The maximum possible upper bound on the number of vertex covers for each set of the form $H_{m,m}^j$ occurs when $j = m$, this bound being $2^m$.

Graphical enumeration can be used to investigate various other graph prop-
### Figure 7-11: $H_{3,3}^4$


![Graphs](image)

**Table 7-13: Upper and lower bounds by graph enumeration**

<table>
<thead>
<tr>
<th>Graph</th>
<th>Total graphs</th>
<th>$V_{min}$</th>
<th>$V_{max}$</th>
<th>Graph</th>
<th>Total graphs</th>
<th>$V_{min}$</th>
<th>$V_{max}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$H_{3,3}^3$</td>
<td>1</td>
<td>8</td>
<td>8</td>
<td>$H_{5,5}^5$</td>
<td>1</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>$H_{3,3}^3$</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>$H_{5,5}^5$</td>
<td>20</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>$H_{3,3}^3$</td>
<td>9</td>
<td>4</td>
<td>4</td>
<td>$H_{5,5}^7$</td>
<td>190</td>
<td>16</td>
<td>20</td>
</tr>
<tr>
<td>$H_{3,3}^2$</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>$H_{5,5}^8$</td>
<td>1140</td>
<td>8</td>
<td>18</td>
</tr>
<tr>
<td>$H_{4,4}^4$</td>
<td>1</td>
<td>16</td>
<td>16</td>
<td>$H_{5,5}^9$</td>
<td>4835</td>
<td>4</td>
<td>14</td>
</tr>
<tr>
<td>$H_{4,4}^5$</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>$H_{5,5}^{10}$</td>
<td>15344</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>$H_{4,4}^6$</td>
<td>66</td>
<td>8</td>
<td>10</td>
<td>$H_{5,5}^{11}$</td>
<td>37560</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>$H_{4,4}^7$</td>
<td>212</td>
<td>4</td>
<td>8</td>
<td>$H_{5,5}^{12}$</td>
<td>71940</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>$H_{4,4}^8$</td>
<td>423</td>
<td>2</td>
<td>7</td>
<td>$H_{5,5}^{13}$</td>
<td>108055</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>$H_{4,4}^9$</td>
<td>516</td>
<td>2</td>
<td>5</td>
<td>$H_{5,5}^{14}$</td>
<td>126140</td>
<td>2</td>
<td>6</td>
</tr>
</tbody>
</table>

**Figure 7-12: Upper and lower bounds by graph enumeration**

properties and test hypotheses. Most of the results given in this section were either suggested or proven using this technique.

Figure 7–13 shows that the set of vertex degrees does not uniquely determine the number of vertex covers that a graph possesses: here both graphs have four vertices of degree one and four vertices of degree two. Figure 7–14 shows that the number of vertex covers that a graph possesses does not uniquely determine the set of vertex degrees either, with the first graph having three vertices of degree one, four of degree two and one of degree three, and the second graph having two vertices of degree one and six of degree two.
Lemma 7.6 If the number of vertex covers is two for a graph \( P \in H^k_{m,m} \), then \( \forall \) vertices \( v \in P \), \( d(v) > 1 \), where \( d(v) \) is the degree of vertex \( v \).

Proof: Choosing all the vertices \( \in L \) or all the vertices \( \in R \) will produce two vertex covers for every graph in \( H^k_{m,m} \). If there are no more than two vertex covers then there can be no way of covering all the edges of a graph using a mixture of \( L \) and \( R \) vertices.

Suppose that there was a vertex \( X \in L \) of degree one in the graph, as shown in Figure 7-15. The vertex cover possibilities are:

1. all the vertices \( \in L \).
2. all the vertices \( \in R \).
3. all the vertices \( \in L \) except for \( X \) together with the vertex \( X' \in R \).

Therefore the number of vertex covers is three. The argument for a vertex \( Y \in R \) of degree one is similar. But we know that there is no way of covering all the edges of the graph using a mixture of \( L \) and \( R \) vertices and so there cannot be any vertices in the graph with degree one. As there are no vertices of degree zero, \( \forall \) vertices \( v \), \( d(v) > 1 \).
Lemma 7.7 For all vertices $v$, $d(v) > 1 \not\Rightarrow$ the number of vertex covers is two.

Proof: See the counterexample in Figure 7–16, where $d(v) = 2$. 

If the number of edges in $U_{m,m}$ is greater than $m$, the resulting vertices with degree greater than one tend to reduce the number of possible vertex covers. However, this section concludes with two Lemmas which show that the relation between the maximum possible number of vertex covers $V_{\text{max}}$, and the maximum possible number of vertices of degree one $D_{\text{max}}$, for a graph $f \in H_{m,m}^k$, is not straightforward.

Lemma 7.8 Given a graph $f \in H_{m,m}^k$, $f$ has $V_{\text{max}}$ vertex covers $\not\Rightarrow f$ has $D_{\text{max}}$ vertices of degree one.

Proof: See the example graphs from $H_{5,5}^9$ in Figure 7–17, where $V_{\text{max}} = 14$ and $D_{\text{max}} = 6$.

Lemma 7.9 Given a graph $f \in H_{m,m}^k$, $f$ has $D_{\text{max}}$ vertices of degree one $\not\Rightarrow f$ has $V_{\text{max}}$ vertex covers.

Proof: See the example graphs from $H_{4,4}^9$ in Figure 7–18, where $V_{\text{max}} = 4$ and $D_{\text{max}} = 2$. 

Figure 7–16: Counterexample
CHAPTER 7. GRAPH THEORETICAL METHODS

Figure 7–17: Example graphs from $H_{5,5}^9$

Figure 7–18: Example graphs from $H_{4,4}^9$
(b) Subgraph of Unlabelled Vertices

When $U_{m,m}$ is a part of a larger graph, a slightly different approach must be used. In this instance it may be perfectly acceptable to use all the unlabelled $L$ or $R$ vertices as part of the total vertex cover. Figure 7–19 shows an example where it is in fact essential to do this. Here the unlabelled vertex subgraph has two possible vertex covers and so all the vertices $\in R$ are selected. Note that if the vertices $\in L$ had been selected then the $K_{2,3}$ subgraph would not have been found.

This example demonstrates that the vertex cover found by the search algorithm must be considered when determining the vertex cover for the unlabelled vertex subgraph. It is simply not possible to consider both vertex covers independently.
Conclusions

Considerable difficulties are encountered when attempting to find a vertex cover for the unlabelled graph. An inappropriate choice of vertices may lead to no edge reducible bipartite subgraph being found. An exhaustive search through the set of potential vertex covers is NP-complete in general (but might have acceptable run-time for small graphs, say $m \leq 5$).

The problem of inappropriate choice of vertices for the vertex cover is not confined to the unlabelled vertex subgraph problem, but is sometimes an unfortunate result of the search algorithm itself, as will now be demonstrated. Consider Figure 7-20. By Figure 7-2 a matching of size three yields a vertex cover of size three, which gives an independent set of size nine and a complete bipartite subgraph of size nine, so the $K_{4,2}$ subgraph has not been detected. Indeed a literal interpretation of the definition of a complete bipartite graph is required, in order to view the detection procedure as correct. Recalling the definition of a bipartite graph $G = (V, E)$, where $V = L \cup R$, $L \cap R = 0$, we must allow $L$ or $R$ to be the empty set and must view the graph on $N_{x,0}$ as a complete bipartite subgraph. Therefore in Figure 7-20 the algorithm has detected $K_{9,0}$. In the unlabelled vertex subgraph problem on $U_{m,m}$, when all the $L$ or $R$ vertices are selected for the vertex cover, the graphs $K_{0,m}$ or $K_{m,0}$ are detected respectively. This observation is formalised in Lemma 7.10.

**Lemma 7.10** The smallest complete bipartite subgraph that can be detected by the algorithm of Figure 7-2 is of size $\max(x, y)$.

**Proof:** By Lemma 7.4 the size of the largest possible matching in a graph $G_{x,y}$ is $\min(x, y)$. Therefore the smallest complete bipartite subgraph that may be found in the graph $G_{x,y}$ by Figure 7-2 is of size $x + y - \min(x, y) = \max(x, y)$.

This definition is unproductive for the purposes of finding edge reducible complete bipartite subgraphs, and leads to some unexpected results, such as for
example that every graph contains subgraphs which are bipartite (indeed p-partite) and complete. An edge reduction algorithm actually requires to detect complete bipartite subgraphs of the form $K_{p,q}$, $0 < p, q < \max(x, y)$, $p, q \in \mathcal{N}$, and using this definition the search procedure will fail for certain types of graph. This form of complete graph is detected when $G_{x,y}$ contains a matching of size $\min(x, y) - r$, $r < \min(x, y)$, $r \in \mathcal{N} - \{0\}$, giving a complete bipartite subgraph of size $x + y - \min(x, y) + r = \max(x, y) + r$. However as shown by Figure 7–21, the search algorithm can still lead to unlabelled vertices in this case. Therefore we actually need to find the graph that has been detected before we can tell if a suitable graph has been detected.

These results show that for edge reduction purposes this purely graph theoretical approach to the detection of complete bipartite subgraphs is unreliable, and could only ever be of limited applicability. The requirement to detect complete bipartite subgraphs of size less than $\max(x, y)$ means that other approaches, such as those described in the next chapter, must be used.
Figure 7-20: Counterexample

Figure 7-21: Unlabelled vertices example
Chapter 8

Cluster Detection and Pattern Matching

This chapter describes several algorithms which detect complete bipartite subgraphs in bipartite graphs, in order to perform the edge reduction node clustering technique. Each of the example graphs was drawn using variations of the directed graph layout algorithm developed by Sugiyama et al. [STT81]. As described in Chapter 2, the algorithm uses the barycentre heuristic to reduce the number of edge crossings between adjacent levels of the graph. This heuristic does not guarantee to produce the minimal number of crossings, and so it may be possible to draw the graphs shown here with fewer crossings. Also, different layouts can be created by applying weights to the vertices and permuting their initial configurations. Other variations can be produced by changing the sizes of the vertices, the minimum vertex separation on particular levels, and the spacing between each level. However, this chapter is more concerned with the problems of avoiding the selection of sets of complete bipartite subgraphs which result in a very poor final presentation. Later sections investigate the effects of using some more complex connective properties of the edge reduction node. Therefore minor variations in the positions of the vertices (and hence the number of edge crossings) produced by the rendering algorithm tend to be relatively unimportant.
The following section will describe two approximate solution algorithms for edge reduction. In later sections a structure-oriented algorithm will be presented.

8.1 Approximate solutions

An approximation algorithm attempts to find a “reasonable” set of complete bipartite subgraphs which can be edge reduced, producing a final result that approximately satisfies the user’s goal. No deliberate attempt is made to present any structural information – the task is just to simplify the graph using any selection of easily determined edge-reducible subgraphs. In many cases this will produce an acceptable final product.

8.1.1 Newberry’s algorithm

This section will present an algorithm developed by Newberry [New89], for a bipartite graph $G = (V, E), V = L \cup R, L \cap R = 0$. This algorithm attempts to approximately satisfy the goal of finding the set of complete bipartite subgraphs which will yield the fewest number of edges in the reduced graph, by finding a set of complete bipartite subgraphs such that the vertices $E R$ of any subgraph are not permitted to be a subset of the vertices $E R$ of any other subgraph.

The algorithm is shown in Figure 8-1. In the following explanation an intersection is defined to be an instance of $K_{2,m}$ and a concentration is defined to be an instance of $K_{n,p}$. $L_x$ or $R_x$ denote the vertices $E L$ or the vertices $E R$ of a concentration $x$ respectively. Given the graph $G$, the algorithm firstly creates a list of all possible intersections, sorted in increasing order of number of vertices $E R$. Each intersection is added in turn to an initially empty list of concentrations. If the vertices $E R$ of one subgraph are a subset of the vertices $E R$ of another, then the larger subgraph is split into two, with each piece being added to the list of
Comment: Create a list of independent concentrations that are at least of size \( M \).

1. Set the initial list of concentrations to the empty list.

2. Sort the successors of each vertex to make the calculation of the intersection easier in the next step.

3. For each pair of vertices \( \in L \), calculate their intersection and maintain it on a list of intersections that is sorted in increasing order based on the number of vertices \( \in R \) (a sorted list leads to fewer splits).

4. LOOP: for each intersection \( I \) in the intersection list:
   - If the size of \( I < M \) then discard \( I \) and continue LOOP.
   - Compare the intersection \( I \) with each concentration \( C \) in the concentration list. If \( R_I = R_C \) then add \( L_I \) to the vertices in \( L_C \) and continue LOOP.
   - Compare the intersection \( I \) with each concentration \( C \) in the concentration list.
     - if \( R_I \subset R_C \) then split \( C \) into \( I \) and \( C' \), add these to the concentration list and continue LOOP.
     - otherwise if \( R_I \supset R_C \) then split \( I \) into \( I' \) and \( C \), add these to the concentration list and continue LOOP.

5. Merge edge concentrations that have the same set of vertices \( \in R \) and discard those where the size is \( < M \).

**Figure 8-1:** Newberry's approximate solution
The algorithm requires a minimum size $M$ for concentrations $C$ to be specified by the user, where $M = (|R_C| - 1)(|L_C|)$. This means that subgraphs of the form $K_{1,n}$ or $K_{n,1}$ will not be detected. A final pass through the list merges concentrations with the same set of vertices $\in R$ and discards concentrations that are too small. The run time for graph $G$ is $O(|L|^4)$.

Figure 8–2 shows an application of this algorithm to an example bipartite graph. The vertices $\in R$ of each intersection are

\{ CDEFG, ABCDEFG, CDEFGHI \}

and so the cluster construction proceeds as shown in Figure 8–3.

Figure 8–4 shows a counterexample which Newberry uses to illustrate that the heuristic of splitting the concentrations "may not always be optimal". Here many
1. Concentration list $CL = \{ \}$.  

2. Add intersection $\{ \text{CDEFG} \}$. $CL$ is now $\{ \{ \text{CDEFG} \} \}$.  

3. Add the intersection $\{ \text{ABCDEFG} \}$. Find that the existing concentration $\{ \text{CDEFG} \}$ is a subset so split $\{ \text{ABCDEFG} \}$ into $\{ \text{AB} \}$ and $\{ \text{CDEFG} \}$. $CL$ is now $\{ \{ \text{CDEFG} \} \{ \text{AB} \} \{ \text{CDEFG} \} \}$.  

4. Add the intersection $\{ \text{CDEFGHI} \}$. Find that the existing concentration $\{ \text{CDEFG} \}$ is a subset so split $\{ \text{CDEFGHI} \}$ into $\{ \text{CDEFG} \}$ and $\{ \text{HI} \}$. $CL$ is now $\{ \{ \text{CDEFG} \} \{ \text{AB} \} \{ \text{CDEFG} \} \{ \text{CDEFG} \} \{ \text{HI} \} \}$.  

5. After merging $CL = \{ \{ \text{AB} \} \{ \text{CDEFG} \} \{ \text{HI} \} \}$.  

**Figure 8-3:** Cluster construction for Figure 8-2  

Top level vertices are connected to the set $\{ \text{CDEGH} \}$, however this concentration is split as vertices 5 and 6 are connected to the subset $\{ \text{GH} \}$.  

The clustering algorithm can be used with a non-bipartite directed graph by applying it repeatedly to different 'levels' of the graph and specifying different minimum concentration sizes as required.  

**Failure Modes**  

For certain types of bipartite graph the clustering technique may not produce an acceptable result, and may even make the graph harder to understand. For example consider the graph in Figure 8-5. Here there are a large number of complete bipartite subgraphs where the sets of vertices $\in R$ have very small intersections, and this results in a great deal of edge interference in the reduced graph. Therefore Newberry's algorithm may not produce ideal results for every graph. When there are a large number of complete subgraphs of this form within the graph
CHAPTER 8. CLUSTER DETECTION AND PATTERN MATCHING

Figure 8–4: Counterexample

Newberry's solution (13 crossings)

before (87 crossings)

optimal solution (3 crossings)
it may sometimes produce a better result if only a few are displayed. This is the underlying principle of the new algorithm that will be described in the next section.

8.1.2 Another approximate algorithm

Figure 8–6 presents a new approximate edge reduction algorithm for a bipartite graph $G = (V, E)$, $V = L \cup R$, $L \cap R = 0$. Firstly a list of all subgraphs of the form $K_{2,n}$, $n > 0$, is created. Clusters with the same vertices $\in R$ are then amalgamated, or merged together. Next, pairs of clusters are selected which have intersecting vertices and which do not cover more than $T$ vertices that are covered already, where $T$ is an integer value defined by the user, typically in the range three to five. This will mean that these pairs will be relatively independent of previously detected clusters and hence there should be little edge-interference between them. Finally, any remaining single clusters which do not cover more than $T$ previously covered vertices are selected.

The effect of this strategy is to introduce the edge reduction nodes into the graph such that complete bipartite subgraphs may be formed between the vertices $\in L$ and the reduction nodes, and also between the reduction nodes and the
1. Form all instances of $K_{2,n}$, $n > 1$, and place in a set $S$.

2. Amalgamate($S$).

3. Let $M = \emptyset$.

4. Determine the pair of clusters, with a non-null intersection of vertices, which cover the largest number of vertices, at most $T$ of which may be covered already by clusters in set $M$. Move this pair from set $S$ to set $M$.

5. Repeat stage 4 until there are no pairs left.

6. Add to $M$ any single clusters from $S$ if they do not cover more than $T$ already covered vertices.

$M$ is now the set of candidate complete bipartite subgraphs.

**Figure 8–6:** *New approximate algorithm*
vertices $\in R$. These subgraphs may then be reduced in further applications of the algorithm to the appropriate new levels in the graph.

Figure 8–7 illustrates the result of applying this algorithm to the example bipartite graph of Figure 8–2. Cluster construction proceeds as shown in Figure 8–8 with $T = 4$. The new algorithm produces a 'similar' result to Newberry’s algorithm for this example, however there are no edge crossings after the second pass. A second pass of Newberry’s algorithm also reduces the number of edge crossings to zero at the expense of introducing another two edge reduction nodes.

Figure 8–9 illustrates the result of the algorithm on the counterexample graph of Figure 8–4. Here the number of edge crossings is three after two passes which compares with the three crossings of the 'optimal' solution of Figure 8–4 (the introduction of a third edge reduction node in these graphs will reduce the number of crossings to two).

Figure 8–10 shows the results of applying the algorithm to the difficult case of Figure 8–5. As only two edge reduction nodes are used, the final result is acceptable. Graph readability is improved and the number of edge crossings is reduced.
On the first pass:

1. \( S = \{ \{ \text{CDEFG} \}, \{ \text{ABCDEFG} \}, \{ \text{CDEFGHI} \} \} \).

2. Amalgamating set \( S \) has no effect.

3. \( M = \{ \} \).

4. Add \( \{ \text{ABCDEFG} \} \) and \( \{ \text{CDEFGHI} \} \) to \( M \) and remove from \( S \).

5. \( M = \{ \{ \text{ABCDEFG} \}, \{ \text{CDEFGHI} \} \}, S = \{ \text{CDEFG} \} \).

On the second pass:

1. \( S = \{ \{ \text{CDEFG} \} \} \).

2. Amalgamating set \( S \) has no effect.

3. \( M = \{ \} \).

4. Add \( \{ \text{CDEFG} \} \) to \( M \) and remove from \( S \).

5. \( M = \{ \{ \text{CDEFG} \} \}, S = \{ \} \).

**Figure 8–8:** Cluster selection for Figure 8–7
The run time of the algorithm in the worst case is $O(|L|^4)$ (stage 4 calculates the intersection of each of the pair of the $O(|L|^2)$ clusters formed in stage 1). However on average the algorithm will be much faster than this as it will not usually be the case that every pair of vertices $\in L$ will be part of a $K_{2,p}$ subgraph, $1 \leq p \leq |R|$. Also, the amalgamation operation will usually reduce the number of clusters being considered.

Figures 8-11 through 8-14 illustrate further examples of the results of Newberry's algorithm compared with the new algorithm. Figure 8-12 shows that occasionally the use of fewer edge reduction nodes will allow better vertex placement and hence, as a side effect, reduce the number of edge crossings. Figure 8-13 shows an example where the heuristic of splitting the large subgraphs produces a better result in that there are fewer edge crossings. However a new level and a further two edge reduction nodes are introduced as a side effect. Figure 8-14 shows an example of the result of modifying the algorithm to allow the construction of clusters of the form $K_{n,1}$ – a new level and an essentially redundant edge reduction node have been created.

The two approximation algorithms have the same goal – to identify a 'reasonable' set of complete bipartite subgraphs which can be edge reduced and so not surprisingly in certain cases they tend to produce 'similar' results. The new algorithm attempts to refine the set of clusters which are selected, to avoid generating a poor solution in the case where the clustering technique is not suitable for the particular graph. However further improvements in the final product are possible, using a more structured approach which will be described in the next section.
Figure 8–9: New algorithm applied to Figure 8–4

Figure 8–10: New algorithm applied to Figure 8–5
Figure 8.11: Fig program [New89] (simplified) and Newberry's result

before (896 crossings)

after Newberry's algorithm (18 crossings, 2 passes, M=12,6)
New solution (9 crossings, 2 passes, T=3)
CHAPTER 8. CLUSTER DETECTION AND PATTERN MATCHING 172

before (33 crossings)

Newberry's solution (3 crossings, 2 passes, M=4,2)

New solution (8 crossings, 1 pass, T=4)

Figure 8–13: Texchk program from [New89]
CHAPTER 8. CLUSTER DETECTION AND PATTERN MATCHING

Figure 8-14: Excerpt from Composed_view.o function calls relation
8.2 Structure-oriented Algorithm

A structure-oriented algorithm attempts to take maximum advantage of the connective properties of the reduction node in order to display the internal detail of the graph more clearly. It is impossible in general to display every complete bipartite subgraph contained within a bipartite graph using edge reduction node clusters, due to interference between subgraphs that are not independent. The examples have also shown that many different sets of reduction node clusters can be used to display the same graph. This notion of equivalence under the Edge Reduction Node (ERN) graph transformation, or ERN-equivalence is formalised below.

**Definition 8.1** Consider a labelled graph $G^e$ which contains edge reduction nodes. The reverse edge reduction graph transformation is defined to replace each edge reduction node cluster with the equivalent complete bipartite subgraph to form the graph $G^b$. The graph $G^u$ is formed from $G^b$ by removing the labels from the vertices. Using this transformation, two graphs $G_1^e$ and $G_2^e$ which contain edge reduction nodes are defined to be ERN-equivalent iff $G_1^u = G_2^u$.

Two graphs can be tested for ERN-equivalence in polynomial time using depth first search.

Figure 8-15 shows an example of two ERN-equivalent graphs. Here $G_2$ is the more concise graph as it has the fewer edge reduction nodes. The reduction rule shown in Figure 8-16 can be used to transform graph $G_1$ to $G_2$. Figure 8-17 shows some further examples of possible graph reduction rules.

Figure 8-18 illustrates an interesting transformation. Given the pair of clusters $H_1$ and $H_2$, a further edge reduction is performed on the graph yielding $H_3$. This graph is then reduced using reduction rule $R_3$ to give $H_4$. 
Figure 8-15: ERN-equivalent graphs

Figure 8-16: Example Reduction Rule
CHAPTER 8. CLUSTER DETECTION AND PATTERN MATCHING

Figure 8-17: Further Reduction Rules
Figures 8–19 and 8–20 illustrate other examples of this type of graph transformation. This evidence tends to suggest the existence of a set of rules which might directly merge pairs of clusters, producing the same results as further passes of the edge reduction algorithm and applications of reduction rules. Figures 8–21 through 8–25 illustrate the complete set of such rules for pairs of clusters. In the example transformations, merge rule $equal_2$ corresponds to Figure 8–18, rule $equal_4$ corresponds to Figure 8–19, while rule $subset_1$ corresponds to Figure 8–20.
Figure 8-19: Example transformation 2
Figure 8–20: Example transformation 3
Rules: equal₁: if \( L_1 = L_2 \) then form cluster
\[ R_1 \cup R_2 \]

equal₂: if \( R_1 = R_2 \) then form cluster
\[ L_1 \cup L_2 \]

Figure 8–21: Equal
Rules: if \( L_1 \cap L_2 = 0 \) and \( R_1 \cap R_2 = 0 \) then no merge is possible
- if \( L_1 \cap L_2 = 0 \) and \( R_1 \subset R_2 \) then use no_subset_{1L}
- if \( R_1 \cap R_2 = 0 \) and \( L_1 \subset L_2 \) then use no_subset_{1R}
- if \( L_1 \cap L_2 = 0 \) and \( R_1 \not\subset R_2 \) and \( R_1 \cap R_2 \neq 0 \) then use no_subset_{2L}
- if \( R_1 \cap R_2 = 0 \) and \( L_1 \not\subset L_2 \) and \( L_1 \cap L_2 \neq 0 \) then use no_subset_{2R}

Figure 8-22: No subset 1L and 1R
Figure 8–23: No subset 2L and 2R
CHAPTER 8. CLUSTER DETECTION AND PATTERN MATCHING 183

Rules:
- if $L_1 \subseteq L_2$ and $R_1 \subseteq R_2$ then discard $C_1$
- if $L_2 \subseteq L_1$ and $R_1 \subseteq R_2$ then use subset$_1$
- if $R_2 \subseteq R_1$ and $L_1 \subseteq L_2$ then use subset$_1$
- if $L_1 \subseteq L_2$ and $R_1 \not\subseteq R_2$ and $R_1 \cap R_2 \neq 0$ then use subset$_2L$
- if $R_1 \subseteq R_2$ and $L_1 \not\subseteq L_2$ and $L_1 \cap L_2 \neq 0$ then use subset$_2R$

Figure 8–24: Subset
Rule: if \( L_1 \notin L_2 \) and \( L_1 \cap L_2 \neq \{\} \) and \( R_1 \notin R_2 \) and \( R_1 \cap R_2 \neq \{\} \) 
then use contains_subset

**Figure 8–25:** Contains subset
In general the clearest, most concise reduction node representation of a graph (i.e. the one with the fewest possible edge reduction nodes) seems to best convey the information contained within the graph. The use of merge rules is beneficial in this respect, achieving the same effect as further combinations of edge reductions and reduction rules. This is the basis of the new algorithm.

Ideally *every* possible complete bipartite subgraph should be found for a graph, and the minimum number used to cover the maximum number of vertices such that the graph is displayed to show the greatest amount of sub-structure. In general, the detection of *every* possible complete bipartite subgraph will be NP-complete. However, the cluster amalgamation technique from the previous algorithm can alter the types of graph for which a detection algorithm's best and worst case run-times occur. The use of this technique means that for graphs where edge reduction can be used to good effect the run-time is short, while for graphs where a good result will not be produced the run-time is exponential. Therefore the algorithm can in effect time itself to determine whether or not it will have any beneficial result.

The structure-oriented algorithm is shown in Figures 8-26 and 8-27. Firstly, all complete bipartite subgraphs with two vertices $E \in L$ are formed. Each cluster is associated with a set of vertices $E \in L$ called the *augmenting set*, which the algorithm will serially attempt to add to the cluster. The addition of a vertex $x \in L$ may cause the number of vertices $E \in R$ to remain the same if $x$ is incident with all vertices $E \in R$, or to reduce if it is not. After one vertex has been added to each existing cluster, the cluster set is amalgamated, and the process repeated until the augmenting set of each cluster is empty.

Figure 8-28 illustrates the behaviour of the algorithm when applied to the example graph of Figure 8-2. The result is the same as that of the new approximate algorithm, as shown in Figure 8-7.
Number each vertex $e \in L$ from 1..$|L|$.

for $a = 1$ to $|L|$ do begin

  for $b = a + 1$ to $|L|$ do begin

    form the cluster (if it exists) with $L$ vertices $a$ and $b$,
    and place it in the Cluster-set $CS$

    for $c = b + 1$ to $|L|$ do Aug$_i$ = Aug$_i$ + $c$

  end

end

Amalgamate

while for each cluster $C_i \in CS$, Aug$_i$ $\neq \{\}$ do

  $CS_{new} = \{\}$

  for each cluster $C_i$ in $CS$ do if Aug$_i$ $\neq \{\}$ then begin

    augment $C_i$ with $S_k \in$ Aug$_i$ to form cluster $C_j$

    Aug$_i$ = Aug$_i$ - $S_k$

    if $|R_{C_j}| = 0$ then Aug$_i$ = $\{\}$ else begin

      Aug$_j$ = Aug$_i$

      Place new cluster in $CS_{new}$

    end

  end

$CS = CS + CS_{new}$

Amalgamate

endwhile

Figure 8-26: Structure-oriented Algorithm
Procedure Amalgamate

if |CS| = 1 then exit else

for each cluster $C_i$ in $CS$ do

compare $C_i$ with every other cluster $C_j$

if $R_{C_i} = R_{C_j}$ then begin

$L_{C_i} = L_{C_i} + L_{C_j}$  \{ Add the $L_C$ vertices together \}

$Aug_i = Aug_i - S_{C_j}$  \{ Remove the $S_{C_j}$ vertices from $Aug_i$ \}

Delete $C_j$ from $CS$

end

endfor

---

**Figure 8–27: Amalgamation Procedure**

---

**Best Case Run Time on $N_{m,n}$**

The best case occurs when each $K_{2,n}$ has the same set of vertices $\in R$ i.e. the graph is $K_{m,n}$. Here there are no passes through the augmentation loop: all instances of $K_{2,n}$ are formed in $O(m^2)$ and these are then amalgamated in $O(m^2)$, giving a total process run time of $O(m^2)$.

---

**Worst Case Run Time on $N_{m,n}$**

The worst case occurs when no amalgamation is possible i.e. each cluster has a different set of vertices $\in R$. If the worst case graph contains every possible cluster of this form then there will be

$$E_m = \binom{m}{2} + \binom{m}{3} + \ldots + \binom{m}{m-1} + \binom{m}{m}$$

clusters, and hence from previous results the algorithm will run in $O(2^m)$ time.
In the following explanation, each cluster $C_i$ is represented as:

\[
\{ \text{vertices } \in L / \text{vertices } \in R / \text{Aug}_i \}
\]

1. \( CS = \{ \{ 1,2 / ABCDEFG / \{ 3 \} \}, \{ 1,3 / CDEFG / \{ \} \}, \{ 2,3 / CDEFGHI / \{ \} \} \} \).

2. Amalgamation has no effect.

3. \( CS_{new} = \{ \} \).

4. \( C_j = \{ 1,2,3 / CDEFG / \{ \} \} \).

5. \( CS_{new} = \{ \{ 1,2,3 / CDEFG / \{ \} \} \} \).

6. \( CS = \{ \{ 1,2 / ABCDEFG / \{ \} \}, \{ 1,3 / CDEFG / \{ \} \}, \{ 2,3 / CDEFGHI / \{ \} \}, \{ 1,2,3 / CDEFG / \{ \} \} \} \).

7. After amalgamation, \( CS = \{ \{ 1,2 / ABCDEFG / \{ \} \}, \{ 1,2,3 / CDEFG / \{ \} \}, \{ 2,3 / CDEFGHI / \{ \} \} \} \).

Merging process:

1. \( V[1,2] = 10 ; V[1,3] = 12 ; V[2,3] = 10. \)

2. Select clusters 1 and 3.

3. Apply rule \textit{contains subset}.

After the elimination of the edge reduction node with only one in-going edge (from vertex 2), the final result is the same as that shown in Figure 8–7.

\textbf{Figure 8–28: Structure-oriented algorithm application}
Self-Timing

The self-timing aspect of the algorithm can be implemented in several ways. Firstly, a count might be taken of the number of iterations of the main loop. If this count exceeds some user defined threshold then there would be too many clusters being processed, indicating that there are many complete bipartite subgraphs with independent sets of vertices $E \subseteq R$. Alternatively a more flexible idea would be to use a concurrent timer process – if the cluster formation process takes too long to produce a result then the timer process would cause it to terminate. The set of clusters which had been created could then be passed on to the merging routine described in the next section, or the algorithm could issue a suitable error message and stop.

Merging

The merging routine processes the set of clusters determined by the previous part of the algorithm, as outlined in Figure 8–29. The basic strategy that is adopted is to find the pair of clusters which have some common vertices and which cover the maximum possible total number of vertices. This heuristic was found to be highly successful in all of the example graphs that were tested. The appropriate merge rule is then applied to the pair of clusters. The process is then repeated if there are further pairs of the correct form which do not cover more than $T$ vertices which have been covered already. Again $T$ is a user defined integer value, typically ranging from three to five. Next, any single clusters are added to the set if they also do not cover more than $T$ vertices which have been covered already. An optional final pass replaces edge reduction node clusters with only one in-going or out-going edge by the corresponding $K_{1,q}$ or $K_{q,1}$ subgraph, as this representation uses fewer edges and can sometimes be more straightforward.

Examples of the results of the new algorithm are shown in Figures 8–30 through 8–35. In each case the use of edge reduction nodes makes the graph easier to visu-
1. For each pair of clusters $C_i, C_j$ in $CS$ do
   
   if there are common vertices $\in L$ or common vertices $\in R$ then
   calculate the number of vertices there would be if these clusters were
   merged, storing this value in $V[i,j]$.

2. Select the pair of clusters with $\max(V[i,j])$.

3. Apply the appropriate merge rule.

4. Eliminate $C_i$ and $C_j$ from $CS$.

**Figure 8–29: Merging routine**

---

**Figure 8–30: Graph from Figure 8–4**
alise and understand. In Figure 8–35 Boehm et al.’s primitive and intermediate characteristics for software engineering [BBK+78] especially illustrate the ability of the reduction node to highlight special relationships between the vertices. For example, the lower cluster might be interpreted to mean that software structure and self-descriptiveness combine together to influence testability and understandability. Other interpretations and viewpoints are also possible and will of course be dependent on the specific graph in question.
Structured solution (9 crossings)
Figure 8-33: Composed_view.o function calls relation
Figure 8–34: *Edge-reduced Composed_view.o function calls relation*
Figure 8.35: Boehm's primitive and intermediate characteristics

Before

- Device independence
- Completeness
- Accuracy
- Consistency
- Device efficiency
- Accessibility
- Communicativeness
- Structuredness
- Self descriptiveness
- Conciseness
- Legibility
- Augmentability

- Portability
- Reliability
- Efficiency
- Human engineering
- Testability
- Understandability
- Modifiability

After

- Device independence
- Completeness
- Accuracy
- Consistency
- Device efficiency
- Accessibility
- Communicativeness
- Structuredness
- Self descriptiveness
- Conciseness
- Legibility
- Augmentability

- Portability
- Reliability
- Efficiency
- Human engineering
- Testability
- Understandability
- Modifiability
type vertices = ( a,b,c,d,e,f,g,h ) ;
cluster = record
    L : set of vertices ;
    R : set of vertices ;
end ;

Figure 8–36: Cluster definition

![Cluster representation diagram]

L : 1 1 1 1 0 0 0 0
R : 1 0 0 1 0 1 0 0

Figure 8–37: Example Cluster Representation

8.2.1 Implementation

The efficiency of the implementation will determine the usability and success of
the algorithm. In particular the complex pattern matching process of merge rule
application needs very careful consideration together with the representation of
the graph that is used. Fortunately the required efficiency can be obtained using
the “Pascal Set” implementation technique. If the implementation of a cluster is
defined as shown in Figure 8–36, then each set of vertices will be represented in
a machine word, with the value of each bit corresponding to whether or not the
associated vertex exists. An example is shown in Figure 8–37.

If the maximum number of vertices ∈ L or ∈ R is defined to be a multiple of
eight, then the set representation will fit neatly into a whole number of memory
bytes. Operations such as comparing sets of vertices ∈ R become simple machine
equals comparisons, and the amalgamation of sets of vertices becomes a machine or operation.

The identification of the correct merge rule to use for two clusters $C_1$ and $C_2$ is also performed via efficient set operations, for example:

$$\text{if } C_1.L = C_2.L \text{ then use rule equal}_1$$

The full rule determination algorithm is shown in Figure 8-38. Each intersection operation is performed by a machine and. Each subset operation is performed by a machine and followed by a machine equals comparison (an ‘$a = b$’ comparison test is performed at the start of the algorithm and so the equivalence $a \subseteq b \equiv a \& b = a$ can be used here).

The representation makes the cluster manipulation operations very easy and efficient, giving the overall merge rule application process a run time of $O(n^2)$, where $n$ is the number of clusters found by the cluster detection algorithm (each such pair of clusters must be tested).

8.2.2 Conclusions

The new algorithm seems to produce a more interesting graph, one which reveals more about the sub-structures contained within it. In presenting these structures, the connectivity information contained within the graph is clearly and concisely expressed. A good result may not be produced when there are a great many complete bipartite subgraphs to be displayed which are not independent. However the clustering technique in general does not work well in this case.

Figure 8-39 shows that the relative ability of each algorithm to reduce the number of edge crossings in a set of example graphs does not form a sound basis for comparison, since vertex placement and hierarchical layout will bias the results produced. In fact, every graph in this thesis which contains edge reduction nodes
if $C_1.L = C_2.L$ then $equal_1$ else
if $C_1.R = C_2.R$ then $equal_2$ else
if $C_1.L \cap C_2.L = 0$ then begin
  if $C_1.R \cap C_2.R = 0$ then no merge possible else
  if $(C_1.R \subset C_2.R)$ or $(C_2.R \subset C_1.R)$ then no-subset$_1L$ else no-subset$_2R$
end
else if $C_1.R \cap C_2.R = 0$ then begin
  if $(C_1.L \subset C_2.L)$ or $(C_2.L \subset C_1.L)$ then no-subset$_1R$ else no-subset$_2R$
end
else if $C_1.L \subset C_2.L$ then begin
  if $C_1.R \subset C_2.R$ then discard $C_1$ else
  if $C_2.R \subset C_1.R$ then subset$_1$ else subset$_2L$
end
else if $C_2.L \subset C_1.L$ then begin
  if $C_2.R \subset C_1.R$ then discard $C_2$ else
  if $C_1.R \subset C_2.R$ then subset$_1$ else subset$_2R$
end
else if $C_1.R \subset C_2.R$ or $C_2.R \subset C_1.R$ then subset$_2R$ else contains-subset

Figure 8-38: Merge rule Determination
## Figure 8–39: Comparison of Edge Crossing Reduction Performance

<table>
<thead>
<tr>
<th>Figure</th>
<th>Originally</th>
<th>Newberry</th>
<th>New Approx</th>
<th>Structured</th>
</tr>
</thead>
<tbody>
<tr>
<td>8–2</td>
<td>54</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8–4</td>
<td>87</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>8–5</td>
<td>28</td>
<td>48</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>8–11</td>
<td>896</td>
<td>18</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>8–13</td>
<td>33</td>
<td>3</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>8–14</td>
<td>45</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

is planar, and may be drawn with no edge crossings. However in doing so the information conveyed by vertex adjacency and hierarchy is usually lost.

Figures 8–40 and 8–41 show the equivalent comparisons of the number of edges and vertices respectively for these graphs. With all three comparisons, the structure-oriented algorithm performs as well as or usually better than Newberry’s algorithm. However, these simple metrics do not capture the essential difference between the approaches, namely that the structure-oriented algorithm attempts to impose more order on the presentation. The regular patterns of linked edge reduction nodes which are produced by the merge rules help the reader to understand the graph. The graphs of Figures 8–30 and 8–32 show some evidence to suggest that the algorithm can reduce the number of edge reduction nodes which are required in some cases, and thereby improve graph readability. As a side effect, this can often reduce the number of edge crossings.

The cluster merge heuristic seems to be very useful for reducing the number of edge crossings. It is unlikely that a single heuristic will work well in all possible cases, and so it might be tailored to produce a more relevant or appropriate result if required.
### Figure 8-40: Comparison of Number of Edges Produced

<table>
<thead>
<tr>
<th>Figure</th>
<th>Originally</th>
<th>Newberry</th>
<th>New Approx</th>
<th>Structured</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-2</td>
<td>23</td>
<td>16</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>8-4</td>
<td>30</td>
<td>21</td>
<td>21</td>
<td>18</td>
</tr>
<tr>
<td>8-5</td>
<td>12</td>
<td>24</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>8-11</td>
<td>91</td>
<td>54</td>
<td>53</td>
<td>53</td>
</tr>
<tr>
<td>8-13</td>
<td>20</td>
<td>19</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>8-14</td>
<td>22</td>
<td>14</td>
<td>16</td>
<td>12</td>
</tr>
</tbody>
</table>

### Figure 8-41: Comparison of Number of Vertices Produced

<table>
<thead>
<tr>
<th>Figure</th>
<th>Originally</th>
<th>Newberry</th>
<th>New Approx</th>
<th>Structured</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-2</td>
<td>12</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>8-4</td>
<td>17</td>
<td>19</td>
<td>20</td>
<td>19</td>
</tr>
<tr>
<td>8-5</td>
<td>8</td>
<td>14</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>8-11</td>
<td>47</td>
<td>52</td>
<td>50</td>
<td>51</td>
</tr>
<tr>
<td>8-13</td>
<td>13</td>
<td>17</td>
<td>15</td>
<td>17</td>
</tr>
<tr>
<td>8-14</td>
<td>12</td>
<td>14</td>
<td>15</td>
<td>14</td>
</tr>
</tbody>
</table>

*Figure 8-40: Comparison of Number of Edges Produced*

*Figure 8-41: Comparison of Number of Vertices Produced*
The self-timing aspect of the algorithm is interesting in itself. This new algorithm class is distinguished by the characteristic that when a good solution exists it will be found quickly, and when there is no good solution then it takes exponential time to determine any solution at all.

The structure-oriented algorithm uses self-timing combined with the definition of the quantity $T$ to prevent the production of a solution such as that shown in Figure 8–5, in the case where the graph is of the type that is not amenable to the clustering technique. Newberry’s approximate solution makes no attempt to deal with this problem although it could perhaps be modified in some way to do so. The new approximate algorithm relies solely on the definition of $T$ to generate an appropriate result for this case.

The structure-oriented algorithm represents a more intelligent application of the edge reduction node clustering technique. If computation time is a relevant factor then the user now has the choice of either an approximate solution or a structural, informatic solution that attempts to show the most interesting clusters in the graph, but which may take longer to compute. Clearly it would be possible to run both solutions in parallel in this instance, selecting the appropriate final product according to any computation time constraint.

In summary, the structure-oriented algorithm uses the connective properties of the reduction node to a greater extent than that of the approximate algorithms, yielding a more regular result. Also, there is some evidence to suggest that using fewer edge reduction nodes improves graph readability and may allow a vertex placement which reduces the number of edge crossings. In general, a graph seems to be easier to understand if it contains the minimum number of edges, edge reduction nodes and edge crossings. On some occasions a trade off between these factors may be necessary in order to achieve the best possible presentation, while in other cases it may be necessary to reduce the readability of the graph in order to present the information that the user requires.
For general graphs, the complete bipartite subgraphs may be formed between vertices on different 'levels', and hence a straightforward application of any level-based algorithm may not produce the best possible result. In this case a more complex analysis of the graph as a whole is required. The extension of the algorithms to cope with this difficult case is left for future research.

An Optimal Solution?

The definition of optimal must be carefully considered. Newberry [New89] gave the definition of an optimal set of edge concentrations to be the set of complete bipartite subgraphs which have the fewest number of edges in the final graph, and suggested that Figure 8–2 showed the optimal solution with sixteen edges. The different approach of the work presented here suggests that in fact Figure 8–7, with fifteen edges, is the optimal solution by this definition (or at least this author cannot find a better solution). As previously described, the definition of 'optimal' might be to produce the minimum number of edge crossings in the final graph, to produce the best structural summary, to detect the largest sub-structures in the graph or to give the best summary in a given time. In conclusion, there may well be a different optimal solution for every graph depending on the user's requirements, and the algorithms presented in this work should simply be viewed as tools to assist users in attaining their goals.

These simple definitions of 'optimal' do not consider all the issues that may be relevant such as vertex placement, graph planarity, requirement for hierarchy etc. For a particular application, the low level goals such as attaining the minimum number of edge crossings may interact and hence interfere with the higher level goals of revealing hierarchy. These are the areas that should be addressed by future research.
Chapter 9

Conclusions

The introduction presented a simple scheme to broadly categorise the types of support that a computing system might provide for program monitoring and debugging. Chapter 2 briefly examined each level of the framework by describing research on previous tools, techniques and systems which realise monitoring and debugging support. The success of the systems which use a hybrid of software and hardware techniques, measured in terms of low cost, ease of use, low overheads and minimal probe effect, provide evidence to support the proposition that an integrated approach has significant advantages. Gorlick's conclusions [Gor91] regarding the future of wafer-scale chip production indicate that, in future high performance systems, an integrated approach for program monitoring and debugging will be essential.

The production of a complete system based on the Fully Integrated Debugging-Oriented System Architecture is a major undertaking, even for a large research group. Extensive simulation of the interactions between the hardware and software components will be required, to validate techniques and explore options before a realistic system could be built. However, some of the possible components of such a system can be researched in isolation, and much of the work presented here could be extended within the framework of the architecture as will be shown presently.
This thesis described work which spanned the top layers of the model. The results produced in each significant area are summarised below, together with directions for future research.

Unassigned Variables

Chapter 3 discussed the methods that language designers have used to prevent and detect unassigned variable errors. The use of formal verification and static analysis techniques to eliminate the possibility of errors of this type was described. The issues of required and default initialisation of variables were considered, and some problems with the techniques were discussed. The approaches used by languages that do not automatically initialise variables were described. The methods for physically representing the unassigned state presented in [KW90] were summarised and extended. The functions of unassigned variables and variables that are never unassigned were discussed, along with the issues of dynamic valuation and propagation of the unassigned value. Finally, methods for adding more useful ways of coping with unassigned variables to the C and Ada languages were considered.

Chapter 4 considered the more complex problems of unassigned pointer variables. Three systems were presented for detecting dangling references at runtime. In the first system, referent objects were never deallocated and were instead marked as deallocated using a flag. The second system used double indirection to keep track of objects that had been deallocated, while the third associated a key with every pointer and referent object in order to do this. Each of these methods relies upon the run-time system to carry out special actions when creating and destroying heap objects and when performing pointer operations. This imposes a run-time overhead on every pointer operation, which can give rise to performance problems.
The double indirection and the key techniques were extended using the smart pointer construct to add dangling reference detection to the C++ language. It was shown how the new facilities do not rely upon any specialised run-time system and hence can be easily added to any given C++ language implementation. As users can precisely target the code that they wish to analyse, it may not be necessary to incur checking overhead for every pointer operation. This can help to alleviate run-time performance problems.

A different kind of C++ smart pointer was then presented that provides a bounds checking facility for pointers which are used to access arrays. Address arithmetic can be performed on this type of pointer. Also, it can be extended to provide other monitoring and diagnostic information in program code where pointers are used to increase array element access speed, with little impact on the syntax of the code that uses the facility.

Chapter 5 examined the problems involved in defining what it means for a composite object to be unassigned, and demonstrated some techniques for determining the state of composites at run-time. The approaches used by several popular programming languages for coping with unassigned composite objects were described. These ranged from having no notion of unassigned for composites, to insisting that all of the components must be assigned before a composite-level operation can take place.

It was shown that the "dual view" of composites as collections of other objects and as objects in their own right leads to inconsistencies which require resolution. The simplifications of this view presented by Winner [Win84] were examined, and it was determined that they create different problems and extra burdens for the programmer. Instead, language constructs were defined which give complete control over the strictness of the definition of unassigned status that is used, and also facilitate selective changes in the level of antipropagation. In this way the most appropriate strategy can be adopted at the point at which it is required.
Run-time unassigned variable errors are often not trapped due to the overhead that is incurred by doing so [KW90]. However some program faults are so difficult to locate that programmers may (temporarily) be prepared to endure even very large overheads in order to track down the cause. Some relatively efficient techniques for monitoring pointer behaviour in C++ have been presented. Winner's work has been extended to place the actions taken, and overhead incurred, when detecting unassigned scalar and composite objects under direct programmer control. In this way the resources available for run-time checking can be efficiently targeted to the regions of code in which they are needed.

Future work in this area might attempt to experimentally validate the techniques of Borie et. al. [BPM93] as presented in Section 3.5.5, in order to determine their usefulness in practice. The techniques might be extended to C++ and hence permit the use of inheritance and operator overloading to give a less intrusive programming style. A comparison with the results of implementing standard methods of detecting unassigned variables in C++ should yield valuable insights into this aspect of language design.

Other work in this area could include a quantitative evaluation of the overhead of tracking composite object states. Simulation could be used to determine the type of software and hardware architecture that best supports the required debugging functionality for a specific language. The treatment of unassigned objects interacts with many other features of a language design. The overhead of supporting the detection of unassigned composite objects may sometimes be too great, and so compromise implementations that support unassigned checking only for scalar objects should be investigated. The work on composite objects could be extended to consider variant records or union types, and the issues of equivalence at the object, component and physical representation levels. The results of simulating many different approaches and architectures should hopefully reveal the subset of unassigned variable checking strategies that provide the most function-
ality and usefulness, for the least overhead and cost. This will form one of the foundations of a fully integrated system architecture.

Graph Display

Chapters 6 to 8 assumed the techniques of Chapters 3 to 5 to ensure that any data structure selected for rendering was “real”. The simplification of complex structures using the *Edge Reduction Node* graph clustering technique was then considered.

Chapter 6 presented an elementary theory of the type of graph for which the clustering technique is useful, the *mvc-bipartite graph*. A calculation for enumerating mvc-bipartite graphs was determined. Some simple results concerning planarity and rendering bipartite graphs under the Y-drawing constraint were derived.

The *Type I* and *Type II* edge reduction node clustering techniques were then defined for directed bipartite and undirected general graphs respectively. The real information content of bipartite graphs was considered along with some possible motivations for edge reduction algorithms. The requirement to detect the complete bipartite subgraphs present within a graph was identified. Detection by exhaustive searching was shown to be intractable.

Chapter 7 demonstrated that the standard graph theoretical method for detecting the largest complete bipartite subgraph does not work in the sense required for edge reduction purposes. Some shortcomings of a possible vertex labelling algorithm which would be required to actually find the vertices of this largest subgraph were determined. Several interesting results regarding bipartite graphs containing perfect matchings were derived using a graph enumeration engine.

Chapter 8 described the idea of an *approximate solution* to the detection problem, which uses heuristics to determine a “reasonable” set of complete bipartite
subgraphs that can be edge reduced, producing a result which approximately satisfies a user's goal. An algorithm independently developed by Newberry [New89] was described, and shown to produce unacceptable results for a certain type of bipartite graph. Another approximate algorithm was presented that attempts to cope with this class of graph. This led to the idea of a structure-oriented algorithm, which takes advantage of the connective properties of the reduction node in order to display the structures contained within the graph in a clearer way. The notion of equivalence under the edge reduction node graph transformation was formalised, and a set of rules for merging pairs of clusters was derived. A self-timing algorithm based on these merge rules was then constructed and analysed. Examples of the results of the algorithm on graphs taken from the field of software engineering were given. The results yielded by the structure-oriented algorithm appear to be more concise, and show some evidence to suggest that using fewer edge reduction nodes allows better vertex placement, and as a side effect produces fewer edge crossings.

Finally, the possibility of an optimal solution was considered, showing that there may well be a different optimal solution for every graph which depends on the user's exact requirements. The algorithms presented in this work are based on heuristics, which means that they may not produce a suitable solution for every given case. In conclusion, these techniques should be viewed simply as tools which may assist users in achieving their goals.

In future work the purely theoretical study of bipartite graphs containing a perfect matching could be extended, by improving the functionality of the enumeration program to allow the investigation of different graph properties. Ideally, this would involve the formulation of a query language to specify both the hypotheses to be tested, and the sets of graphs to be generated (which would in effect form a database for queries). The results which have been generated by the current program have shown that the general technique of graphical enumeration is a useful tool for disproving hypotheses, by providing a counterexample, and for
proving hypotheses for a finite set of graphs. It has also been found that viewing the sets of graphs can suggest the presence of relations which can be formally proved.

The practical application of edge reduction techniques to graphs where the vertices of the complete bipartite subgraphs are positioned on different levels requires further study. Facilities to abstract subgraphs of this form into a separate display, or to use different colours to draw attention to them might be investigated.

Newberry [New89] conjectured that the problem of finding the set of complete bipartite subgraphs which have the fewest number of edges in the final graph might be NP-complete. In the same manner this author suspects that many of the more useful definitions of an "optimal" set of edge reduction node clusters may be computationally intractable. Formal proofs of these hypotheses remain elusive.

Another area that requires future investigation is the study of the trade-offs between the different aesthetic principles used to render a graph containing edge reduction nodes. Different levels of priority could be determined for each aesthetic, so that all of the relevant issues such as hierarchic vertex placement, reduction of total area etc. are considered in producing the final result. The effects of a modification to the new algorithms to replace the quantity $T$ with two new values, $T_{\text{upper}}$ and $T_{\text{lower}}$, representing the maximum upper and lower vertex overlaps in the cluster formation process respectively, might be considered. Finally, procedures for incremental graph update in the presence of edge reduction nodes could be determined.
Bibliography


Bibliography


Bibliography


