Methodology Independent CASE Tool

A Prototype.

A thesis presented in partial fulfilment of
the requirements for the degree
of Master of Arts in Computer Science at
Massey University

Paul Clark
1994

Abstract

The object-oriented (OO) movement is at present split into many factions and as a result no standard has been defined. A direct consequence of this lack of consensus is that there are no mature CASE tools available. Current object-oriented CASE tools are methodology dependent. They are not driven by the need to enable the successful construction of OO software but rather to sell a particular methodology (at this time there are more than 30 different OO methodologies. Forcing the developer to use a particular methodology constrains his/her ability to select the most appropriate problem representation. To address these problems a research project aiming at the development of a methodology independent OO CASE tool has commenced. The thesis is the first stage in the tool development. It addresses two main problem areas of this research project:

- The development of a customisable user interface which utilises an abstract notation definition language.
- Support of the impementation phase of OO software.

A language which facilitates abstract definitions of graphical notations and the human computer interaction with them has been developed. The implemented graphical user interface uses the developed language to allow arbitrary notations and dialogues for OO models to be described and modified without recompilation of the CASE Tool.

Re-engineering facilities have been designed to allow a user to generate an OO model from existing OO source code. Automatic layout generation has been investigated and several auto-routing algorithms applied. The current tool generates Coad and Yourdon diagrams from C++ source code. The user can update and navigate the C++ source via the created OO model.

Based on the thesis results two papers were presented at the International conference on Object Oriented Information Systems in London, December 1994. A collaborative research has also commenced with the University of Technology, Sydney.

Acknowledgments

I would like to thank

Stephanie first because she wanted to be at the top of the list.

Daniela Mehandjiska-Stavreva for being my supervisor.

The following people for checking my work:
Shamus Smith
David Page
Daniela Mehandjiska-Stavreva
Sarah Stock

Wolfgang Tietz for lending me some OO literature.

Table of Contents

Introduction

Most CASE tools available on the market today do not always deliver the type of functionality that would best suit the organisation using it. The users have to adopt the methodology supported by the CASE tool they use. If a new methodology is required the costs for the new software and retraining can be crippling (Beynon-Davies 1993).

If future CASE tools do not solve this restriction then users will have problems representing data in the most meaningful way. This may produce ambiguous information or a model that only shows a partial understanding of the data present.

To address these problems a research project aiming at the development of a methodology independent OO CASE tool has commenced. The tool will use definition files that will describe the layout and interaction that characterise a particular methodology. Once the file is loaded the CASE tool will act as a diagramming tool for that notation.

The configuration and customisation features of the tool will allow the user to specify a new methodology or adapt an existing one. This will allow an organisation to progressively evolve a methodology to keep up with new and existing demands. The only requirement is that a new definition file should be specified outlining the new methodology or the necessary adaptation.

As the object oriented approach becomes more widely used so does reuse. This can provide greater recycling of code and a saving in development times and costs. To facilitate this emerging technology the CASE tool needs to support re-engineering technologies.

The re-engineering facilitates allow the user to load in existing code and see the hierarchical and non-hierarchical graphs visual representation of the program. The user may then either manipulate the diagram or the code. At this stage only the C++ programming language is supported. The resulting information will be saved back to the files with the code in either the original format or in a user specified format.

The graphs use specific heuristic's to increase the aesthetics of the diagram layout. This will produce a more readable and easier to understand representation. Much detail has gone into the design of the heuristic's so that a quality diagram can be produced.

This thesis is the combination of two related areas of research: a methodology independent CASE tool and automated re-engineering facilities for that tool. The large scope of the research and the time constraints did not allow the re-engineering stage to reach the level of completion that was originally planned.

The division of work is approximately sixty five percent for the CASE tool and thirty five percent for re-engineering. Initially it was hoped for a fifty fifty split to allow a fair appraisal of each segment. Regardless of the split it was never envisioned that the writer would be able to complete either section as the sole worker. The split therefore is only an indication of the work involved.

The initial idea of building such a tool was based on work of the Object Oriented Development Research Group, mainly Daniela Mehandjiska-Stavreva, David Page, Wolfgang Tietz and myself. A trial system was built by David Page to test the feasibility of describing graphical notations. The object hierarchy for this initial layout is shown in appendix A. The system was very simple with few features implemented. There was no research done into the feasibility of this functionality being combined into a CASE tool. The tool at that stage consisted of:

- A hard coded template definition. The template definition file that is now used was not available.
- Basic template classes for arcs, lines, text and list text. These preformed only the very minimal functionality necessary to test the theory.
- No connections definition
- A basic windows interface.
- Basic expressions some of which were not fully functional.

The systems purpose was to test the initial concept of using templates for the construction of graphical forms. The large thesis scope proven by this initial trail could have restricted the thesis topic only to the development of a language which facilitates abstract definitions of graphical notations and the human-computer interaction with them. The intention of the whole OOD research group to build an initial prototype and check the feasibility of the construction of such tool , as well as my determination, commitment and desire to see the tool completed determined the final version of this thesis. The outcome and the results of this research have allowed us to submit a three years proposal to the Foundation for Research, Science and Technology for building a fully functional production tool. In parallel to my work some additional work on the user interface design utilising Tcl and Tk had been done by David Page. Jonathan Ham has developed a template generator which automatically generates methodology notation description files in the proposed, designed and implemented in my thesis abstract notation definition language.

The methodology notation used throughout this document is from Coad and Yourdon (1991a, 1991b). It is assumed that the reader has an understanding of the following aspects of this notation shown in figure 1.1.

Symbols	Relationships
Class	Gen-Spec
Class-&-Object	Whole-Part
	Instance
	Message

Figure 1.1 - Areas of the Coad and Yourdon notation readers should be familiar with.

Some of the diagrams drawn in the Coad and Yourdon notation are generated by the CASE tool described within this thesis. These are marked with the text "MOOT" after the figure number, eg "Figure 666 MOOT -". At this stage the tool is not able to produce one to many relationships. Any such relationships seen in diagrams created by the CASE tool are done by overlaying many one to one relationships.

1.1 What is Object Oriented ?

The organisation of software as a collection of discrete objects that incorporate both data structure behaviour has been termed 'object oriented'. Unfortunately the characteristics that accurately define an object oriented approach are not clearly defined, sometimes causing dispute. (Rumbaugh, Blaha, Premerlani, Eddy and Lorenson 1991).

Goor, Hong and Brinkkemper (1992) identify the two fundamental elements upon which the object oriented approach was built upon. These were abstraction and encapsulation.

> Abstraction denotes the use of only essential information about a particular object. This provides a precise boundary between objects.
>
> When one uses abstraction, one admits that a real-world artifact is complex; rather than try to comprehend the entire thing, one selects only part of it (Coad and Yourdon 1991b).
>
> Encapsulation is the grouping of data and code within a module. Encapsulation does not guarantee information hiding (explained below) although the converse is essentially true (Hendersen-Sellers 1992).
>
> Encapsulation is a principle, used when developing an overall program structure, that each component of a program should encapsulate or hide a single design decision. The interface to each module is defined in such a way as to reveal as little as possible about its inner workings (Coad and Yourdon 1991a).

Other key concepts of the object oriented paradigm are:

Objects

> An abstraction of something in a problem domain, reflecting the capabilities of a system to keep information about it, interact with it, or both; an encapsulation of attribute values and exclusive services. (Wirfs-Brock, Wilkerson and Wiener 1990).
>
> At runtime, a single instantiation of the class template (Hendersen-Sellers 1992)

Classes

> A class is an uniform description of one or more objects. This description include the operations and data associated with those objects. In addition it may describe how to create new objects from that class (Coad and Yourdon 1991b).
>
> A class is a set of objects that share a common structure and a common behaviour (Booch 1994).

State

> An objects state is its data (Coad and Yourdon 1991a).
>
> A list of the attributes of an object (Hendersen-Sellers 1992)

Behaviour

> Behaviour is how an object act and reacts, in terms of its state changes and message passing (Booch 1994).

Method

> An operation upon an object declared as part if a class; all methods are operations; but not all operations are methods (Booch 1994).
>
> The implementation of an operation for a specific class (Rumbaugh, Blaha, Premerlani, Eddy and Lorenson 1991).

Message

An action is initiated in object oriented programming by the transmission of a message to an object responsible for that action. The message is accompanied by any additional information (arguments) needed to carry out the indicated action. In response to a message the receiver will perform some method to satisfy the request (Budd 1991).

A request to perform an operation of an object. It includes both the operation name and arguments (Wirfs-Brock, Wilkerson, and Wiener 1990).

Inheritance

Inheritance is a relationship between classes where a class shares the structure or behaviour defined in one or more other classes (single or multiple inheritance). The sub-class is a specialisation of one or more super classes. A sub class generally specialises its super class by augmenting or redefining existing structure and behaviour (Booch 1994).

A mechanism for expressing similarity among classes, simplifying definition of classes similar to ones previously defined. It portrays generalisation and specialisation, making common attributes and services explicit within a class hierarchy or lattice (Coad and Yourdon 1991b).

Information Hiding

The Process of hiding all the details of an object that do not contribute to its essential characteristics; typically, the structure of the object is hidden, as well as the implementation of its methods. The terms encapsulation and information hiding are usually interchangeable (Booch 1994).

Information hiding consists of separating the external aspects of an object, which are accessible to other objects, from the internal implementation details of object the object, which are hidden from other objects (Rumbaugh, Blaha, Premerlani, Eddy and Lorenson 1991).

Polymorphism

Wirfs-Brock et al (1990) state that polymorphism is two or more classes that respond to the same message each in its own way. Each object does not have to know whom it is sending a message to. It only needs to know that there are different kinds of objects that can respond to the message.

The object oriented community is currently split into many fractions. As a result no standards have been defined. A direct consequence of this lack of consensus is that there are no mature CASE tool available. Currently there are over twenty seven different methodologies.

Some organisations are setting up standards and guidelines in an attempt to standardise the industry. Published in 1992 The Common Object Request Broker: Architecture and Specification (COBRA), as described by Object Management Groups Object Request Broker (ORB) provides the mechanisms by which objects transparently make requests and receive information. The ORB provides interoperability between applications on different machines in heterogeneous distributed environments and seamlessly interconnects multiple object systems (FIRST CLASS 1994).

1.1.1 Benefits

There are a large number of benefits that the object oriented approach offers. The general areas all author's have a consensus on are: re-useability, designer thinking, reliability, easier programming, managing complexity and maintainability (Martin 1993, Firesmith 1993, Booch 1994 and Rumbaugh, Blaha, Premerlani, Eddy and Lorenson 1991).

Re-useability is often described as the single biggest factor in sustaining the viability of software engineering organisations. Perhaps the single most important thrust in software engineering in the years ahead is achieving the highest level of re-useability (Coad and Nicola 1993, Lorenz 1993 and Jacobson, Christerson, Jonsson and Övergaard 1994). As is so often the case programmers are constantly reinventing algorithms and methods. What they are developing may have already been built, painstakingly debugged and improved upon countless thousands of times (Booch 1994).

The reason object oriented facilitates the advantages of re-useability so well can be attributed to inheritance and encapsulation. Inheritance allows the developer to build upon existing classes, extending their functionality to accommodate the new needs of the developer (Coleman 1994).

In the past the developer may have either rewritten all of the code they needed or scavenged various parts from existing code. With reuse the developer can inherit from a class that has functionality close to what they desire. Any methods that do not conform with the new class can be overridden.

If the only visible part of an object is the interface then the developer will not design a program around the inner workings of that object. When changes are made to the internal operations of that object the countless systems that have been developed with this object will remain unaffected. Hiding the inner workings in this way is termed encapsulation.

The object oriented approach is generally considered as a natural way of thinking. Outside of the programming world everything is seen and treated as objects that have a particular state and behaviour. For example, a television could have the state of colour, four channel specifiers and the selected channel. The behaviour would be to visually show the information being transmitted on the selected channel.

With the traditional structured approach information in the analysis and design stages of a project involved thinking about details. With object oriented the developer thinks of situations as they would in the real world.

It should be noted that not all examples of object oriented analysis and design are easy to quantify into classes and objects. There are the occasional situations where the area being modelled will not fit the object oriented approach. However these circumstances are rare.

Software reliability is a direct result of re-useability. When a class has been used twenty to thirty times it should be completely bug free. At this stage the value of that particular class increases because it is seen AS a totally reliable part of the system, ie a part that should not need to be checked or tested.

Separating a program into multiple class classifications optimises programming. Each class is treated like a module that can be designed, written and tested independently. The capabilities of the programmer are increased when they only have to deal with a small segment of the overall program at a time.

Building complexity into a system can be a time consuming process for both coding and testing. Object oriented systems overcome this problem by using inheritance to build upon existing complexity. This allows for a much smaller amount of additional code to be built onto an already tested class hierarchy. One example of this involves the re-engineering section of this thesis. Instead of building a single class that will parse C++ and turn this into hierarchical and non-hierarchical graphs, the job is the combined effort of four different classes.

Figure 1.2 shows the C++ Parser object model. The combined complexity of the non hierarchical and hierarchical graph generators are available in the graph generator. Likewise the C++ Parser encapsulates the structure of the parser and has access to the combined complexity of the graph generator classes.

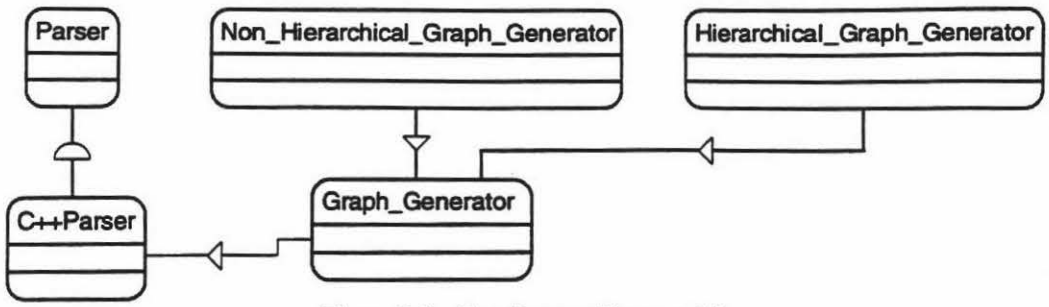


Figure 1.2 - C++ Parser object model.

Encapsulation greatly decreases the amount of maintenance needed on most systems. Since none of the internal workings of classes are visible to the developer, constructing a system will involve only the classes interface. When changes are needed to one or more internal methods the interface stays the same. This makes changes localised and the rest of the program need not be changed.

Sometimes a developers search for benefits is too narrow and they over look the wider scope of benefits. For example, a conversation had with another student who was about to embark on CASE tool development of their own, they wanted to know what benefits object oriented programming could offer them for building a CASE tool. The answer was none. Object oriented provides no discernible benefits directly related to CASE tool development. It does however offer a large number of benefits to software engineering in general, some of which were mentioned earlier.

1.1.2 Warnings

Using an object oriented programming language or environment will not guarantee good results. Like any other human endeavour, software design is an art: discipline, hard work, inspiration and sound technique all play their parts (Wirfs-Brock, Wilkerson, and Wiener 1990).

Wirfs-Brock *et al* note that just using an object oriented programming language is not enough. There have to be sound analysis and design techniques behind the approach. For example a number of attributes for Turbo Vision libraries included with Borland C++ 3.1 are public. This allows the user to change the information held within the class, possibly affecting the functionality provided. This breaks the encapsulation rules set out by the object oriented approach. Even worse is the fact that the user must rely upon some of these attributes for retrieving and setting information.

1.1.3 The Future

"I sincerely believe that the successful software developers of this decade will be the ones that learn the secrets of productivity. One of those secrets is object-oriented development, not just programming, but analysis and design as well." (Brumbaugh 1994).

Not all outlooks for the future are as confident as Brumbaugh. Already one major magazine has pronounced the death of object oriented and there will probably be others. Some people feel that object oriented has been around for a good number of years and has not taken a large share of the market so it will undoubtedly die.

However the object oriented approach is still in its infancy. There are a large number of obstacles that are preventing organisations from taking on this approach. Two problem areas are changing to the new technology and managing this technology. Object oriented programming (OOP) has to contend with changing the way programmers think. Traditional programmers may face a steep learning curve adopting OOP as it takes time to adopt to the OOP style of programming. (Miller 1990). The second problem area, managing the object oriented technology, is of crucial importance to an organisations

success in an industry. For example measuring productivity of workers. In the past this has been measured by the number of lines of code written. This technique can not be used when OOP is involved.

When using the object oriented approach reuse should account for more than half the code required for a system. That makes the old measure of productivity obsolete. There are many other problems, such as the costing out of reusable code when first written, that have not been solved to everyone's satisfaction.

As this decade progresses the object oriented approach will undoubtedly take a firmer grip of the software engineering industry. Organisations can not be expected to throw away years of work just for the possibility of saving time and money. Long term gains have to be proven more viable rather than short term gains.

2. Problem Definition

2.1 CASE Tools

2.1.1 History

CASE is an ancroyn for Computer Aided Software Engineering which is a generic term that refers to the automation of software development (Finkelstein 1989). Another name that can be used is CAISE or Computer Aided Information Systems Engineering.

The tools described as being CASE are a subset of those belonging to CAISE. CASE encompasses only those tools that are concerned with software production. CAISE incorporates these tools plus a large range of other system design tools, such as executive information systems (Beynon-Davies 1993).

"CASE tools have the potential to become the most technological advance in software engineering development. The keyword in the previous sentence is potential." (Pressman 1992). Presently CASE is at the same stage that computer aided design, computer aided engineering and computer integrated manufacturing was in the mid nineteen seventies.

The primary goal of CASE at the present time is the accomplishment of taking design specifications and automatically generating code. Previously it was generally considered that the analysis and design phases were sufficient.

2.1.2 Deficiencies

There are existing CASE tools and methodologies, so why not use these? The reason is that new CASE tools might improve upon flexibility, integration of information and the coordination of work efforts. New methodologies can provide better methods of modelling information that are: more cost effective, easier to use or more likely to produce a well-engineered system meeting the real requirements.

A large proportion of methodologies in place today are adaptations of pencil and paper methods. These were designed before the transition to computers and are not always suitable for today's systems (Brough 1992). Particular pitfalls associated with CASE tools can be identified:

- New methodologies can not be handled by existing CASE tools.
- Users are forced to adopt the methodology supported by the CASE tool they use.
- Existing methodologies do not always allow the user to represent the data in its most understandable form.
- If a change in CASE tools is needed then the price for retraining and information exchange can be crippling.
- No information interchange is available between the analysis and design stages.
- They lack integration between specification and construction tools, typically focus on a narrow portion of the system development process (Sumner 1992).
- Lack support for specifying methodology knowledge.

The main flaws with most CASE tools are the restrictions placed upon the user. The tools are methodology dependant. Instead of choosing the most applicable methodology the tool forces the user to adopt the tenets of the methodology supported (Beynon-Davies 1993, Nilsson 1990, Papahristos and Gray 1991). This results in deficiencies, extra complication in the model or the need to purchase additional software.

Deficiencies in a model occur when the methodology used is unable to express a particular concept. An example of this would be to try to represent parallel operations on a state transition diagram. Large amounts of additional information may have to be added to simulate what has to be modelled thus adding extra complication.

The only other alternative is to purchase another CASE tool that supports a methodology that can best present the information required. This has repercussions in staff training, budgetary constraints, information reuse and software / hardware compatibility.

If more than one tool is used then information interchange is often near impossible. Existing information usually can not be used on the new tool. If the transformation is a simple upgrade then databases and existing information can usually be updated to the new system. However this is not always the case. It is a major step to forfeit all previous work and change to a new methodology. This is one of the reasons that many companies have not changed to an object oriented approach in software engineering.

2.1.3 Problem Area to be Addressed

To compensate for the deficiencies addressed, a CASE tool is required that is independent of any specific methodology. Changing between methodologies will be facilitated without any large cost to an organisation of purchasing software, retraining staff or transferring data.

Users will be presented with a standard interface allowing easy transference of skills, reducing the retraining costs. Information interchange will be possible between different methodologies so previous work may be reused.

This is the concept of the methodology independent CASE tool, called MOOT (Massey Object Oriented Tool). The goal for the development of MOOT is to build an intelligent modifiable CASE tool. MOOT is an intelligent environment which supports the specification of new methodology knowledge

MOOT incorporates some of the characteristics referred to as features of an ideal CASE tool (Gibson 1988, Papahristos 1991). It supports a range of methodologies, has an open architecture and is configurable. Methodology independence avoids conflicts resulting from:

- Having to choose one and only one methodology.
- Becoming CASE vendor dependent.
- Being unable to extend the methodology to meet the special requirements of an organisation, such as having additional annotation for hardware and people related activities.
- Information interchange between analysis and design results expressed in different methodologies.

The resulting system is open so that it may be altered and extended to verify ideas about different methodologies and techniques. The activities of an organisation will not be restricted by the architecture and notation of third party CASE tools (Mehandjiska-Stavreva, Page and Clark 1994).

To facilitate such features the tool will use a notation definition file which allows the specification of conceptual and graphical information. The information directly determines the characteristics for the construction and manipulation of a given methodology notation.

The tool itself is an expert system shell. All knowledge pertaining to a particular methodology including its notation is obtained from the definition file. All that is required to use another notation is loading another definition file.

This thesis addresses some of the problems related to the development of a notation diagramming environment, primarily the creation of a template definition language. Due to this certain features have been considered out of the scope of this thesis and hence left out.

2.1.4 Existing CASE Tools

In the past few years the research into CASE technology has been concentrated in two main areas. The first one addresses the development of software environments with an open architecture aiming at the integration of independently developed CASE tools.

Attempts have been made to create an open environment in which different methodologies and their supporting CASE tools coexist. This approach provides re-useability of information. For example, communication among diverse methodologies is addressed in the Federated CASE Environment (FCE) by a common data dictionary (Papahristos and Gray 1991).

The FCE employs an open methodology environment that gives the user the choice of a number of methodologies. This is not, however, a super methodology where all available notations can be used in the same diagram. Instead a common data dictionary acts as a co-existence mechanism for the different methodologies.

Independent CASE tools could then use this common data dictionary for general storage and exchange of information. Local data dictionaries would be used to maintain the autonomy of the individual CASE tools. Whether the common or local data dictionaries are used is up to the user of the CASE tool. This environment overcomes some of the problems outlined in the deficiencies sub-section but it does not address them all.

If a situation occurred where a particular methodology was unable to completely describe a problem two separate notations would be needed. What is required to solve the deficiencies listed above is a tool that will allow either a variation on an existing or a completely new methodology.

A lot of research has also been done to address the problems of CASE tools methodology dependence. A metamodelling approach has been utilised to allow the generation of customised software specification environments.

The primary purpose of a meta system is to automatically generate the major parts of a software development environment. Meta systems are used to develop software environments in the same way that compiler writing systems produce compilers (Sorenson, Tremblay and McAllister 1988).

Meta system specifications rely upon a CASE shell for the final product. The tool provides the knowledge on how to model a methodology while the CASE shell itself is without knowledge of any particular model. By loading the modelling information into the CASE shell it is transformed into a workbench or modelling support system for that specific process (Verheof, Hofstede and Wijers 1991).

Research prototypes adopting this approach include Metaview (Sorenson, Tremblay and McAllister 1988), MetaEdit (Smolander, Lyytinen, Tahvanainen and Marttiin 1991), MetaPlex RAMATIC (Smolander *et al* 1991) etc.

The environment for a given methodology is a combination of two distinct parts: conceptual and graphical definitions. The conceptual definition is based on different data models. For example, MetaEdit is based on the object-property-role-relationship (OPRR) model.

Metaview is based on the entity-aggregate-relationship-attribute (EARA). RAMATIC is based on the set-oriented data model. The development prototypes support mechanisms to express the mapping between the metamodelling concepts and corresponding graphical representations.

Most of the literature found for this section has exclusively discussed the implementation of meta systems. There have been a wide variety of systems discussed with features ranging from simple diagramming through to software metrics (Boloix, Sorenson and Tremblay 1993).

Most of the past development has been in the area of knowledge production systems for CASE shells. The motivation for this research has been the often clumbersome nature of CASE shells. Most shells end up having only a limited number of people that are capable of using them (Rossi, Gustafsson, Smolander, Johansson and Lyytinen 1992).

Most of these systems have their origins in the public domain. It is unclear whether or not any of these systems actually made it to commercial production. More recent articles have included development tools that are available in the commercial environment. These tools are :

- Virtual Software Factory made by Systematica.
- Ipsys Tool Builder's Kit made by Ipsys.
- ABC Toolkit from Micrografx.
- ABC Flowcharter also from Micrografx.

The latter two tools are for general modelling that are relatively removed from software engineering. They are mentioned here because they allow re-definable methodology notations that could be used as part of a software engineering life cycle.

There is little information available for the first two tools mentioned above. The articles that describe these tools do so in an abstract way. These tools are definitely meta CASE but the extent to which they solve the problems set out in the deficiencies sub-section is unclear. Both of these tools grew out of work done for the Alvey Program in the nineteen eighties. The overall aims of the project was for the improvement of software engineering process (Haine 1992). There is only one report on the success of using this type of tool. In this case it was the Ipsys Tool Builder Kit.

Waldin and Ricketts (1994) state in their case study that choosing a meta CASE tool was the best decision they could have made. They were able to modify and extend the tool at very short notice to meet changing requirements. This was also a key factor in getting the tool accepted by the project members.

The CASE tool proposed in this thesis fits firmly into the second approach set out above. The tool is basically a CASE shell. It requires a definition file to specify the modelling information. The work done by Ham (1994) provides a tool that allows the developer to graphically describe the modelling knowledge needed to operate the shell.

Another similarity to the forementioned environments is that of the common data dictionary. The CASE tool can allow more than one methodology to share a single piece of information. An example is a class definition where the class name is used for both a Coad and Yourdon (1991a) and Booch (1994) class.

If this name is changed through the Booch class then the Coad and Yourdon class is automatically updated. Other than this method the tool requires filters that transfer information between methodologies. More information is given on filters in the next sub-section.

2.2 Methodology Independent CASE Tool Structure

The overall structure of the CASE tool is shown in figure 2.1.

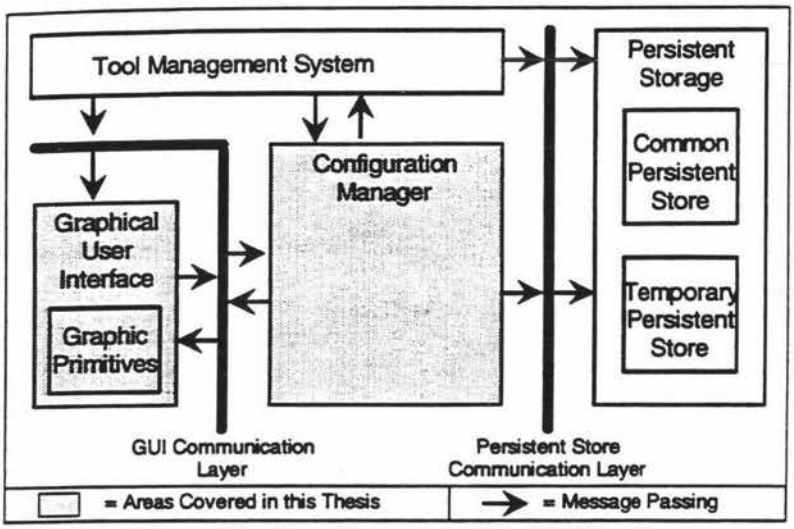


Figure 2.1 - Overall structure of the CASE tool.

2.2.1 Tool Management System

The tool management system is the system care taker. It contains utilities to coordinate, manipulate and uphold the integrity of any data in the system. Some of its functionality is:

- Importing new information into the common persistent store.
- Exporting information from the common persistent store to a temporary persistent store.
- Printing information.

The need for this part of the CASE tool has been identified but no research has yet been done. The exact specifications for this will need to be made at a later stage.

2.2.2 Graphical User Interface

A CASE tool which can support different methodologies and whose graphical user interface (GUI) has a consistent look and feel reduces the amount of effort required to fully utilise these methodologies. Each methodology may use a distinct notation but the interaction mechanisms and interaction items are the same.

The GUI encapsulates all information on how to display, manipulate and control the interface. The GUI communication layer offers a standardised front end for the rest of the system. The front end is made up from graphic primitives that perform operations such as: drawing lines, popping up dialogues and prompting the user for information.

Any methods that communicate with the GUI have no way of knowing how a particular operation will be carried out. This allows the GUI to be changed, upgraded or replaced by a new system without any interference to the rest of the system. The GUI is covered in more detail later in the interface section.

2.2.3 Persistent Storage

The Persistent Storage area of the CASE tool is a database management system. This is made up of two different sub-areas: common and temporary persistent store. The first sub-area encapsulates the bulk of diagrammatic and coding information that is available.

The common store can be used on multiple projects and / or by several users at the same time. This sub-area facilitates the reuse of information by providing access to data obtainable by every user of the CASE tool. As new information that may be required by other users is produced it is added to this store.

The common persistent store is maintained as the standard repository for all information that is to be freely available within the organisation. Various projects use this repository during and after the development of their systems to store all normal data.

The temporary persistent store is used for isolated storage and transference of data. Data placed here is completely separate from the common persistent store. This allows the manipulation of data in an environment that would otherwise cause disruption to users of the system.

For example, consider a situation where a flaw has been found in the structure of a class hierarchy, the entire structure may require reworking. To avoid disruptions to other users, the structure can be worked on independently of the rest of the system. When the alterations are complete the common persistent store can be updated with a minimum of inconvenience.

The temporary persistent store can also be used for the transference of information between CASE tools. Information produced on one CASE tool can be saved, as a duplicate, to a temporary persistent store. This store is then transferred to another system, say for an overseas or local branch, and uploaded into their system common persistent store.

This type of transference greatly increases the accessibility of data between systems. However during the uploading of information, thorough checks have to be made in the systems common persistent store to maintain the integrity of the database. This feature offers increased accessibility to data that can boast the amount of work reused.

Since this part of the system has not been implemented the following views are only what is perceived to be necessary. When it comes to choosing which type of database to use there are three different possibilities: relational, extended relational and object oriented. The first two database types are controlled by database management systems (DBMS) while the latter is controlled by a object database management system (ODBMS).

There are many reasons for using database management systems: abstracting details, database integrity and data sharing, etc. By abstracting out details, the users are shielded from the actual physical details of storage. Users can concentrate on what they wish to do and not how to do it (Date 1990).

Data integrity guarantees the accuracy, completeness and internal consistency of information stored within the database (Rothwell 1993). Data sharing, or concurrency, limits simultaneous reads and updates by different users of a system so that all users have a consistent view of data (Cattell 1991).

If a system is written in an object oriented language then it is sometimes perceived that an object oriented or possibly an extended relation database should be used. However this is incorrect since there are distinct advantages and areas of applicability for each approach. The selection of a database should be dependent upon the specific needs of the application.

The object oriented approach is better suited to complex data structures than a relational database. Application areas where information structures which are very simple can be stored more efficiently in tables. For these it is best to use a relational databases (Jacobson, Christerson, Jonsson and Övergaard 1992).

The category that the CASE tool falls into has to be determined. The information to be stored can be categorised into two different forms: text strings and numerical information. This lends itself heavily towards the relational approach because this is ideal for storing information in tables.

A disadvantage of this type of approach is that each class that will need to be written to the database has to have a specialised method for writing this information. However a large number of available public domain object oriented databases that may also require a similar feature have to be considered.

Textual information is stored as pointers to a text string rather than the actual string itself because at compile time the strings length is unknown. With the large number of public domain ODBMSs that were examined over ninety four percent of these did not chase up pointers. In other words it was still the CASE tools job to save this information.

A front end object is recommended for handling all database operations. This object will have a standard set of primitive commands, in the same style as the interface. The database object will encapsulate the actual DBMS or ODBMS used.

Each viewable thing that needs to be placed into the database is given the database object and told to write itself. The object will pass all its information to the database object which will inturn write the data physically to the database, through the DBMS.

An extended relational database is a combination of both relation and object oriented databases. It is capable of storing information as tables or objects. That type of database can also be ruled out because since the object oriented features are not needed then a relational database need only be used. The only exception is if the most suitable database is an extended relation. In this case it would be the logical choice.

2.2.4 Configuration Manager

The structure of the configuration is a combination of five essential parts: tool inference, interpretation manager, methodology manager, notation manager and filter manager. The interaction between the various parts is shown in figure 2.2.

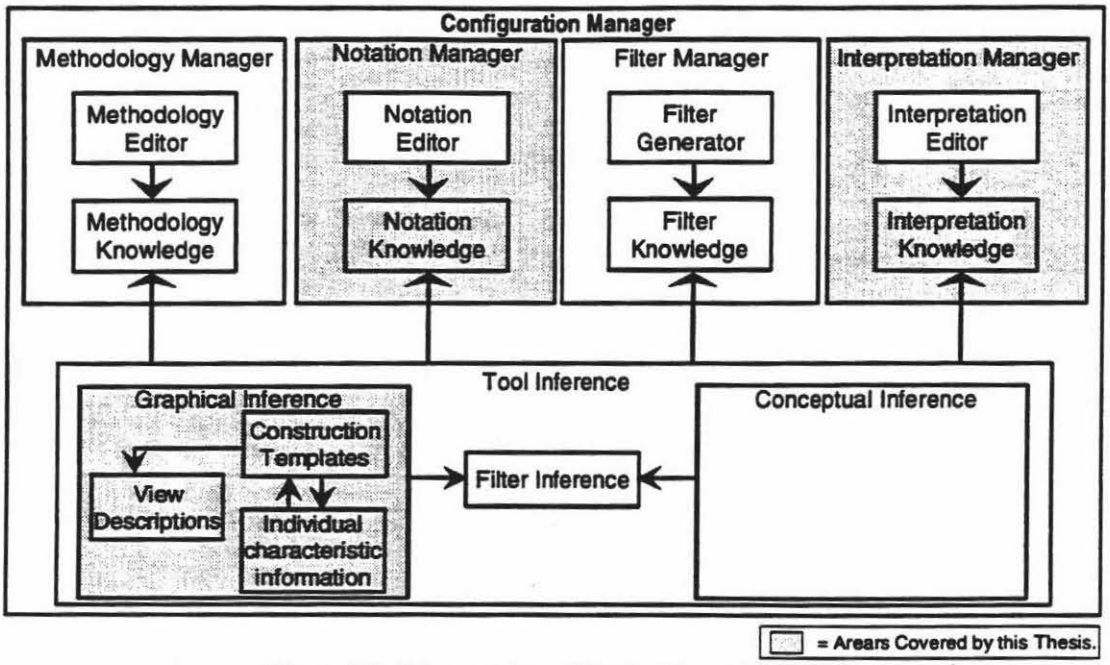


Figure 2.2 - The structure of the Configuration Manager.

2.2.4.1 Tool Inference

The information inside of the tool inference can be separated into three different categories, conceptual, graphical and filter inference. Filter inference will be discussed in the filter manager sub-section below (2.2.4.1.3).

The conceptual information specifies how a particular methodology will react to various forms of data within the diagramming environment. For example if there is a class 'Template Object' in the model can another class, also called 'Template Object', be placed in the same model?

Whereas the methodology and notation editors can be used to generate the conceptual and graphical information the tool inference interprets and uses this information. The tool inference is an expert system shell that uses heuristic information to achieve the appropriate methodology environment.

Conceptual Inference.

At this stage no work has been done on the conceptual side of the CASE tool. To write and test the conceptual inference the graphical inference part of the CASE tool has to exist. Because of this the graphical area was implemented first. Minor discussions on the functionality and layout of the conceptual knowledge has been carried out.

From these discussions it has been ascertained that a generic knowledge base of common methodology features and characteristics will exist. The conceptual knowledge in a newly created methodology can utilise these features by inheritance or implement their own.

An inheritance structure between files will also be available so that common features can be shared by several methodologies. This will be made possible by allowing one definition file to include the contents of another. The information that is require for inheritance will then be specified by the original file. Any information not declared in this manner will be ignored.

By having an inheritance structure, common elements of particular groups or variations of methodologies can be built up. For instance one file could contain core object oriented principles and concepts. Other object oriented methodologies can share these without having to re-specifying them. This core object oriented knowledge would include:

- Class and object diagrams showing:
  - generalisation and specialisation (is-a).
  - part-of.
  - associations.
  - message passing.
- Subject / category diagrams, grouping of classes.
- Attributes belonging to each class and object.
- Services (operations) offered by each class and object.

Figure 2.3 show a possible inheritance structure.

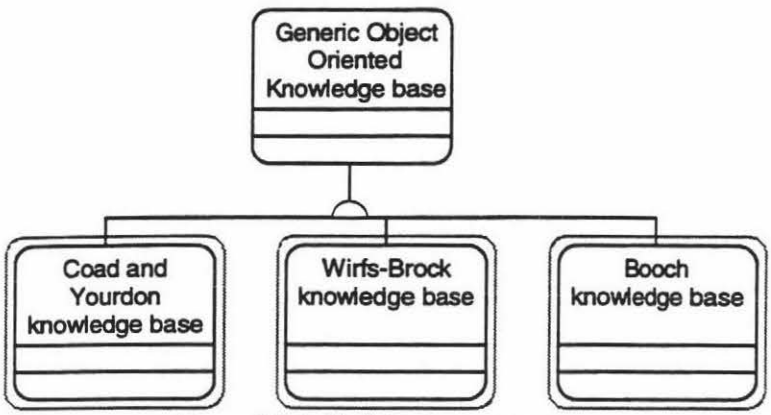


Figure 2.3 A file inheritance structure.

Methodology Manager.
All the information used by the conceptual inference is built using the methodology editor in the methodology manager. This thesis has not considered how such information will developed just that there is a need for it.

Graphical Inference.
The graphical knowledge describes how a methodology is constructed. The three parts shown in figure 2.2 are illustrated in their class form in figure 2.4 (Construction Templates = *Template*, View Descriptions = *View* and Individual Characteristic Information = *Viewable_Thing*).

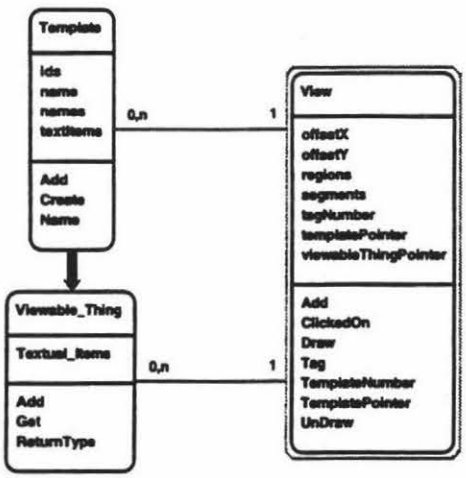


Figure 2.4 [MOOT]- The relationship between the classes Template, Viewable_Thing and View.

The class View is the graphical representation of either an object, connection or composite object on the display canvas. The word object in this document is not used in its standard object oriented meaning. Instead it refers to, unless otherwise specified, a singular item in a methodology notation that does not encompass other objects and is not a connection between objects. Examples of an object are : process bubbles, Coad and Yourdon classes, Booch classes, data stores, etc. The view is constructed from textual information gathered from the class Viewable_Thing.

When a template (a graphical design specification) is used to construct an object, connection or composite object not all information needed is supplied in the template file. The one exception is the textual information that will be in the text segments. Viewable_Thing contains this textual information which has previously been extracted from the user, figure 2.5.

Each piece of textual information is stored with an identification number. When the text segment with the same identification number is constructed it prompts the Viewable_Thing class. Given the identification number the Viewable_Thing class returns whatever corresponding information is available.

Viewable_Thing also holds information pertaining to connections with other Viewable Things. This is usually in the form of relationships. Examples of this are the relationship between a connection and the attached objects or an object and the attached connections. If a new class is created and the user wants to save it in a persistent store then Viewable_Thing would be the only information saved.

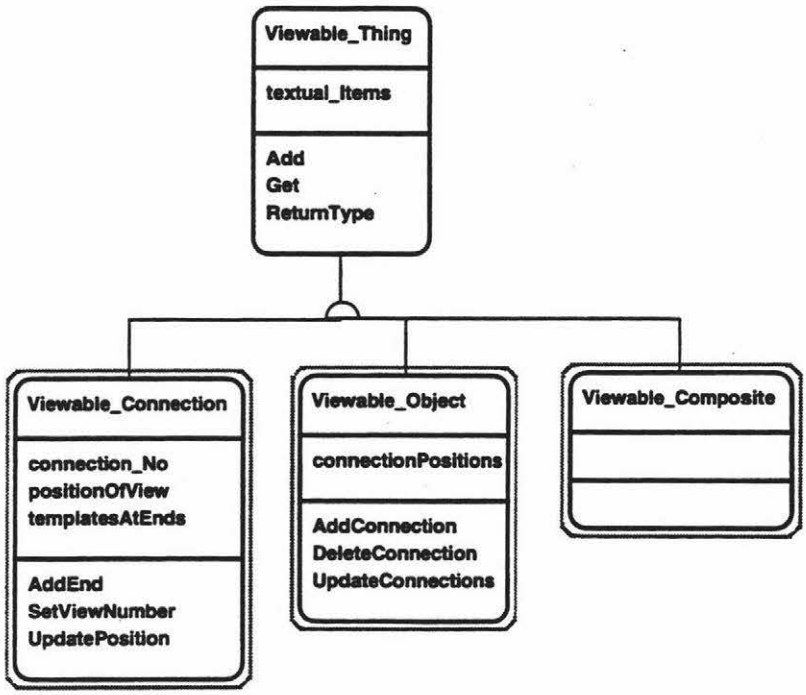


Figure 2.5 [MOOT]- Viewable_Thing class hierarchy.

Templates are specifications on how to build different views. To construct an instance of itself the template needs a Viewable_Thing and the context (the interface window relevant to the view). The Viewable_Thing supplies all the textual information and the context gives access to the graphical user interface, for calculating attributes like text height.

Template has three sub-classes to accommodate connections, objects and composite objects (figure 2.6).

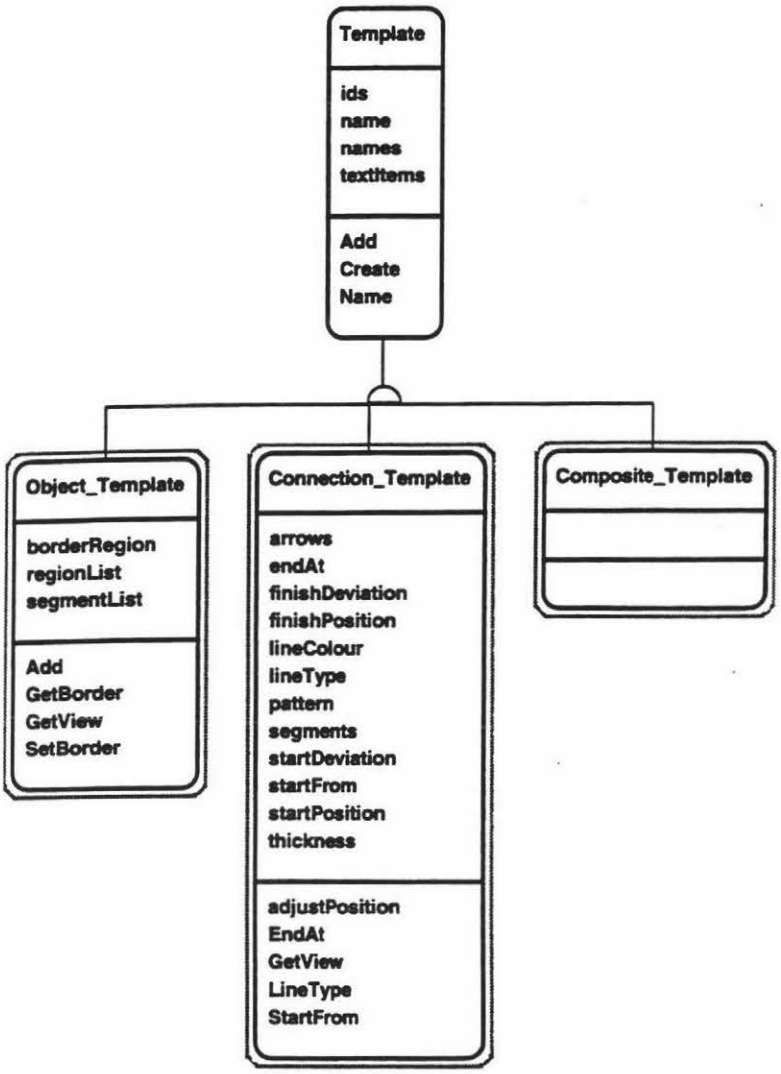


Figure 2.6 [MOOT]- Different template types.

Template is the super class in the hierarchy and provides all the standard methods and instance variables required for the base classes. GetView is mentioned in both Object_Template and Connection_Template but not Template for a good reason. The parameters passed to the method for both classes are quite different making inheritance of this method impractical.

The Template class contains:

- A list of identification numbers and names (*ids* and *names*) to facilitate the acquisition of textual graphical segments. There are two types of textual segments: a single line of text and a list of text lines. These provide information on what question to ask when prompting the user for data to fill these segments. This only occurs when the text segments are initially created.
- The name of the template (*name*). Each time a new object, connection or composite object are created there can be several types to choose from. These names are those types. Examples of names for new object could be "Class" and "Class-&-Object".
- A list of text items (*textItems*). These are pointers to any textual segment contained within the template. These pointers help to facilitate the adaptation of the template being built to cater for varying sizes of the text segments.

Object_Templates are singular items within a methodology that do not encompass other objects and are not connections. This class consists of:

- An encompassing border region, to identify whether the object has been selected (*borderRegion*).
- A list of regions containing function references, to add pre-defined functionality to the diagram (*regionList*).
- A list of segments, to graphically describe the construction of a view of the object (*segmentList*).

This information is quite straight forward and doesn't require the complex sequence of definitions required for the Connection_Template class. These are:

- Information about arrows on the connection. If they are at one, both or neither ends (*arrows*).
- The type of line: single, poly line or poly line constrained to horizontal and vertical (*lineType*).
- A list of graphical segments along with location information (*segments*).
- A list of object templates that a connection can start from and end at (*startFrom* and *endAt*).
- Where the connection ends in relation to the objects (*startPosition* and *finishPosition*) and if there is any deviation how much (*startDeviation* and *finishDeviation*).
- Attributes about the lines such as thickness (*thickness*), colour (*lineColour*) and a pattern that will be applied to the line (*pattern*).

The final type, Composite_Template, refers to the encapsulation of zero to many objects within an encompassing area. This can be in the form of a Coad and Yourdon (1991a) subject area, an exploded process bubble or some other form of information hiding.

The need for this type of template has been identified but nothing has yet been implemented. However a general outline of what can be available in this class are:

- A list of template objects that can be inside of the composite object.
- Whether, when opening the composite object, to explode the information in the current window, open a new window or query the user on which one to do.
- A graphical description of how the composite object looks before and after exploding.

The hardest part in implementing composite objects are the encompassing border of the exploded view. Since the size, shape and number of internal object is unknown then the finished look can vary quite substantially, figure 2.7.

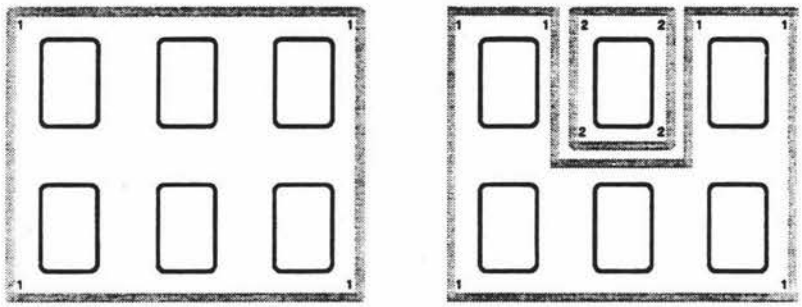


Figure 2.7 - How easily border descriptions can become complex.

The sub-class components, for object and connection templates, will be explained in greater detail further in the Template Language section.

Notation Manager.
The notation editor outlined in the notation manager, figure 2.2, is not covered in this these. Work has been done in this area by Ham (1994). The only part of the notation manager that will be discussed is the notation knowledge, section 3.

Interpretation Manager.
The interpretation manager translates a given language into a format understandable by the CASE tool. This is then processed by the tool inference to extract the relevant information for the formation of a particular methodology.

An interpretation editor is used to create the knowledge necessary for the translation of a particular language. It is necessary to have a different knowledge base for each type of language. The interpretation knowledge provides an exact specification on how to parse a given language.

The information extracted from the language is converted to a format that will suit one or more methodologies. Since the interpreted knowledge has to be processed into a format suitable for a single methodology it is very easy to add the extra information that can be used by several methodologies.

This will allow an organisation to do a single language translation or produce information that will cater for every methodology they use. This will provide a powerful re-engineering capability within the organisation. The retrieved information can even greatly decrease the work required when transferring from a structured to an object oriented approach.

The converted information will have to comply with the heuristic's set out in the conceptual knowledge for the methodology in question. If the rules are broken then the interpretation file must specify how the situation will be handled.

The re-engineering features outlined later in this thesis were implemented in a simplified form to ascertain the feasibility of the concepts for this part of the tool. A single language is converted into a single methodology. The level of success will indicate whether a generic system would be implemented and what changes may need to be made.

Filter Manager.
The filter manager translates information between methodologies. It has not been decided whether this will happen as each new object is created, when the user specifies they want the information filtered or a combination of both. The implementation and specification of the filter manager is not supported in this thesis.

2.4 Selection of Tools

This project is implemented on a Sun Sparc Station LX running Solaris 2.3. This hardware was chosen because of the intention of building the initial prototype as close as possible to a commercial version.

The language chosen to implement the tool is C++ because:

- It is object oriented.
- It is available on multiple platforms, allowing some of the initial work to be done on an IBM PC before porting it to a Solaris based machine.
- C++ has more public domain interface development tools and persistent stores than other object orientated languages like Smalltalk.
- The debugging capability on C++. It allows single stepping through code and indication of the exact line an error has occurred on. The other alternative, Smalltalk, isn't up to the same standard, with some of the error messages being quite cryptic.

There was a choice of two C++ compilers that would work under Solaris, Gnu C++ and AT&T C++ from the Sun Sparcworks package. The latter was chosen because of its strict adherence to the standard. The Gnu compiler had deficiencies in its implementation of templates.

Interface development could be handled in a variety of ways from using XLib routines to using one of a large range of interface development tools. It was decided to implement the interface using a development tool and forgo the arduous task of starting from scratch. As this is an initial prototype the interface will change for the commercial system.

The interface development tool selected was Tcl7.3 and Tk3.6. Tcl is a general purpose interpretive programming language while Tk is the graphical front end. These were first considered because they came highly recommended from researchers at several other New Zealand Universities.

They are public domain software. One other benefit is the tool xf2.3 which coordinates the features of Tcl and Tk into a graphical development environment. This combination offers an inherently powerful tool that allows updates or changes with only minimal effort.

However there are some disadvantages to using Tcl and Tk. Firstly the underlying system of the two tools is not object oriented, it is written in 'C'. To get around the problem of a globally available screen, all the interface specific information is handled through a front end, the Context class.

The front end enforces encapsulation so that if the interface is changed to an object oriented system in the future the changes will be transparent. The last problem is that Tcl and Tk are only interpreted. This may seem like a major problem at first, especially for an interface, but its overall performance was very fast so it was not considered much of a predicament.

3. Notation Definition Language

The environment for a given methodology is specified in two parts: conceptual and graphical definitions. A notation definition language is used to support the definition of the graphical representation of the underlying object model, called templates. This chapter addresses issues and problems involved with the development of a language which facilitate the abstract definitions of graphical notations and the human interaction with them. The language structure, layout and implementation are described.

3.1 Template Language

The template language is a sequential pseudo-English definition of the graphical characteristics for a particular methodology notation. Even though the language has an English-like syntax it is not meant for the developer of a new methodology to use it. A special notation definition specification tool has been developed by Ham (1994). This tool allows the user to specify visually the notation. The template definition file is automatically generated from this definition.

3.1.1 Language Structure

Figure 3.1 shows a top down hierarchical layout of the template file of a given notation. This is only an abstract model to give an overall impression not a specific definition. The rectangular boxes represent an area of definition. For example the "Notation" box specifies that the area being defined is for a notation.

The connections, which are only followed from top to bottom, represent that the boxes connected are a part of that boxes definition. For example a "Notation" definition is made up of "Group Template", "Template Object", "Template Connection" and "Template Composite" parts.

The connection "⊢" specifies that the top box can be made up of many sub-parts. The crossing of connections, like "+" specify that the two connections have no relationship and are simply passing each other.

The numbers above each box represent the number of times that the particular item can be defined in the parent definition. For instance '0,m' signifies that the item may be defined zero to many times whereas a '1' says that the item must have only one occurrence.

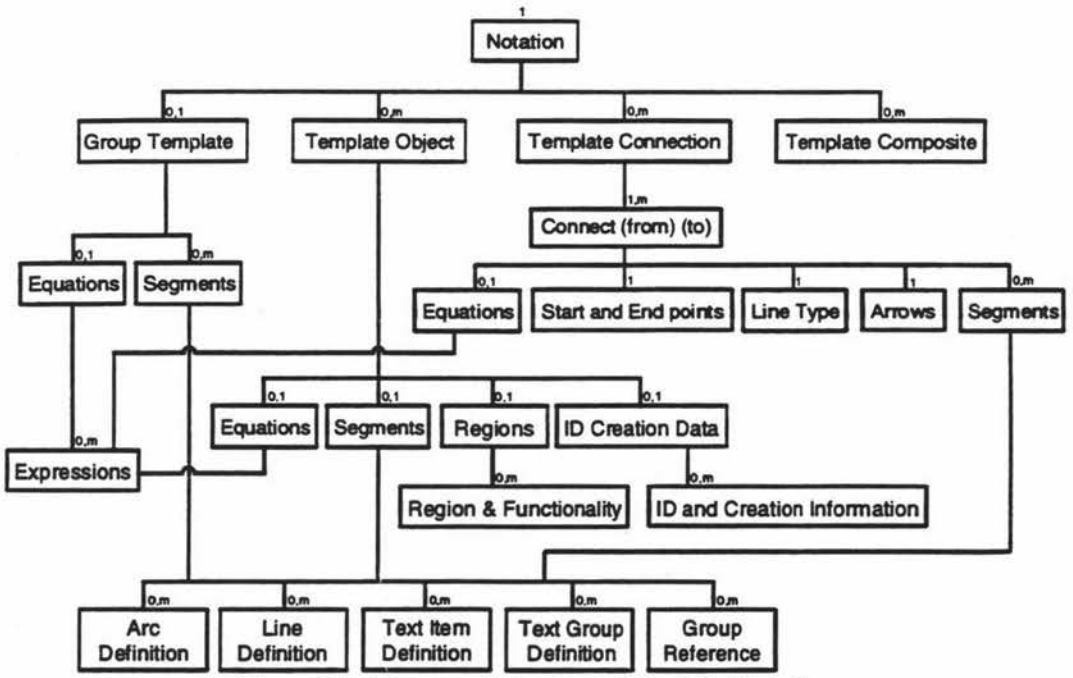


Figure 3.1 - The overall structure of the definition file.

All definitions are set out in the same format in the definition file, although not all parts need to be incorporated. All keywords used in the template language are not subject to case restrictions. They can be upper, lower or mixed case.

Throughout this section each different definition will be shown as both examples of what could appear in a template file and in a language grammar, Backus Naur form. For a complete list of the grammar refer to appendix B.

An example of a grammar is shown below. There are five different possible specifications: 'go north', 'go east', 'go south', 'go west', or do nothing.

```
<Language>      ::= go <Direction> | ε
<Direction>     ::= north | east | south | west
```

<Language> for this example, is the start of the grammar. Proceeding this is the "::=" symbol. The words on the left hand side, which can contain the "_" symbol (eg "Days_Of_The_Week"), are non-terminals. This specifies that the forementioned non-terminal is made up of the following definition, 'go <Direction> | ε'.

Both lines of the grammar now contain a string of text. This indicates that this text has to be present before continuing along this path. The string of text does not include any words surrounded by the "<>" symbols. The "<" symbol indicates that the following word is a non-terminal, unless the "<" symbol is surrounded by single quotes. After this, on the first line, is another non-terminals.

If the text string 'go' is found then the parse will proceed to definition of the following non-terminal <Direction>. This moves the focus to the specification on the grammar line identified by <Direction>. On the right side of the "::=" symbol there can be any number of non-terminals and strings in any order necessary to specify the grammar.

The symbol "|" indicates an 'or' relationship. If the second line of the grammar is taken as an example then there are four options: north or east or south or west. Any of these are valid but only one can be used, unless the identifier is called recursively. If the "|" symbol is enclosed in single quotes it is treated as the character it represents.

The 'ε' symbol at the end of the first line indicates that nothing is to be done. The first line represents two options. The first is 'go', a direction and then stop, since no follow on is specified. The second option is to do nothing and stop. The second line does not require one of these because it forces a selection of one of the four options.

3.2 Notation Definition

The first, compulsory, line in the template language identifies the name of the notation being described. The grammar shown below describes how the notation details are specified. This includes all possible situations.

```
<Template>              ::= Notation <String> <Template_Collection> End
<Template_Collection>   ::= <Group_Templates> <Object_Templates> <Connection_Templates>
```

The following example shows one possible combination of the forementioned grammar as it would appear in the template notation definition file.

```
Notation "Coad and Yourdon"
    .....
End
```

Three different types of templates: groups, objects and connections are defined. Composite templates have been omitted because their use and definition is outside the scope of this thesis.

3.2.1 Group Templates

The first section within the methodology definition is for group templates. Their purpose is to stop redundant information being produced. This can be illustrated using the two diagrams presented in figure 3.2.

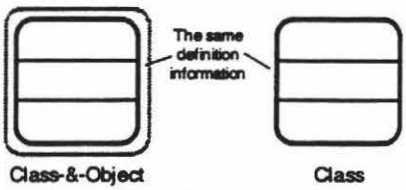


Figure 3.2 - Group templates.

Figure 3.2 shows Coad and Yourdon Class and Class-&-Object graphical icons. Both of these symbols are distinctly similar in their appearances with the Class-&-Object having additional information. Group template allows the common parts of these two diagrams to be defined only once. These common parts are then referenced during the specification of both objects.

The Group Templates section is not compulsory and can thus be omitted if there is no subsequent use for it. The grammar is shown below followed by an example.

```
<Group_Templates>    ::= Template Group <Group> End | ε
<Group>              ::= Shape <Equations> <Segments> End <Group> | Shape
                         <Equations> <Segments> End
```

```
Template Group
    Shape
        ....
    End
    Shape
        ....
    End
End
```

One or more group definitions can be specified. Each definition has two subsections: equations and segments. The definition is started with the keyword 'Shape' and ended with 'End'. These will be referenced by their sequential order of definition in the segments section and in connection definitions.

It is possible for one group template to use the information defined in a previous group template definition. This allows the specifications to be progressively build upon each other. All references to other group templates have to be for previously defined specifications. It should be noted that at this stage in the development of the tool the need for group templates has been identified but not yet implemented.

3.2.1.1 Equations

Equations, which are optional, are lists of predefined expressions that can be referenced in the Segments section, to avoid duplication of equations. Expressions form the basis of any mathematically defined values. These can be classified as numerical constants, complex expressions, text related functions and expression related functions. The class hierarchy that represents the different types of expressions is shown in figure 3.3.

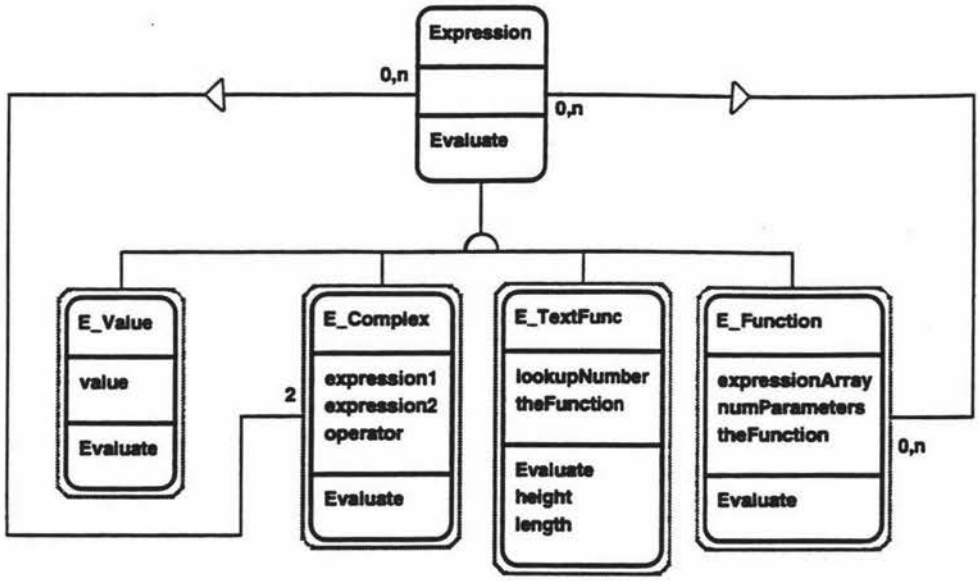


Figure 3.3 MOOT- Expressions Hierarchy.

All these types of expressions can be used in conjunction with parentheses to force the evaluation of any part of a given expression. The first type, numerical constant, is the simplest type of expression which contains either an integer or floating point number. The numeric precision, both left and right the decimal point, is restricted to that of the C++ compiler being used and what it supports.

The complex expression merges two expressions together using one of '*\+-' mathematical symbols. The ordering for these expressions are infix. The precedence for these symbols and parentheses are shown below.

- ( )
- * \
- + -

The last two types of expressions are both function related. The difference in these two types is that an expression function is the application of a function to zero or more expressions whereas the text function applies specific functionality to a text item, after gathering information from a Viewable_Thing.

For text functions only two unique operations have been identified, height and length. This is the only information about a text template that can not be derived when generating the template file. The text functions depend upon the text that is entered and the context in which it will be displayed.

The E_Function expression is an application of a particular function, like maximum or minimum, to a list of expressions. Currently only maximum and minimum have been implemented but there are a large number of other functions that could be implemented. Examples are the trigonometry functions cosine and sine and tan.

To utilise these expressions a backward only reference is used. This can be a part of any expression although it is not categorised as a type of expression. When the program is building up the expressions, any reference is replaced by a pointer to the actual instance of the expression that was referenced. Trying to reference an expression not yet defined will cause an error and the definition file parsing will halt.

The grammar definition and some equation examples are shown below.

<Equations> ::= Equations <Each_Equation> End | ε
<Each_Equation> ::= = <Expression> <Each_Equation> | ε

```
<Expression>        ::= <Expression> + <Term> | <Expression> - <Term> | <Term>
<Term>              ::= <Term> * <Factor> | <Term> / <Factor> | <Factor>
<Factor>            ::= <Float> | ( <Expression> ) | <TextFunc> | <Function> |
                        <EQ_Ref>

<Float>             ::= <Digit> <Float> | . <Integer> | <Digit> | .
<Integer>           ::= <Digit> <Integer> | <Digit>
<Digit>             ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<TextFunc>          ::= Property( <Integer> , <FuncToDo> )
<FuncToDo>          ::= Height | Length

<Function>          ::= Max( <ListOfExpr> ) | Min( <ListOfExpr> )
<ListOfExpr>        ::= <Expression> | <Expression> , <ListOfExpr>

<EQ_Ref>            ::= EQ( <Integer> )
```

```
Equations
    = 10+Property(1,Length)     // ten plus the length of the text item with the ID 1
    = (10.3 * 37 / (4 + 5) + 64)  // A complex expression
    = max(Property(1, Length), Property(99, Length))  // The maximum length of two text items
End
```

3.2.1.2 Segments

Segments are the definition of the graphical parts that make up an object on the screen. There are five different types of graphic definitions: arc, lines, text line, list text templates and group references. Figure 3.4 shows the template segment class (TSegment) and its subclasses.

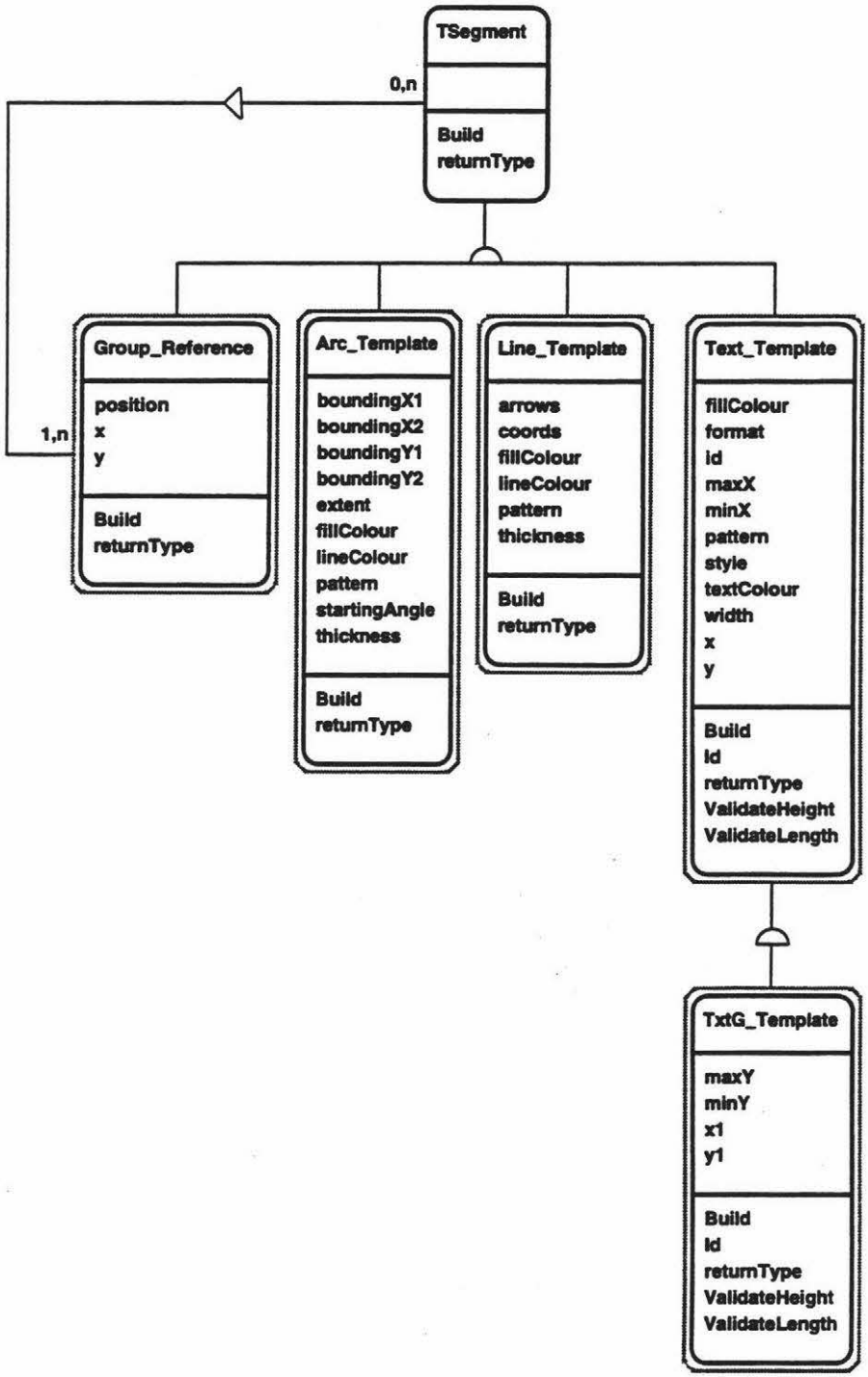


Figure 3.4 [MODT]- Template segment class hierarchy.

All the numerical instance variables, except *position* for Group_Reference and *id* for Text_Temp and TxtG_Temp, are in the form of expressions, as described in the previous sub-section. The grammar for the segments definition is as follows.

<Segments> ::= <Arc_Template> <Segments> | <Line_Template> <Segments> |
<Text_Template> <Segments> | <List_Template> <Segments> |
<Group_Reference> <Segments> | ε

The template segment Group_Reference is for referencing a predefined group of template segments, as discussed in the Group Template section. The description for this includes a reference number to indicate its sequential position in the group template definition and a set of coordinates to represent its position on the screen.

The grammar for a group reference is shown below along with several examples of group references.

<Group_Reference> ::= Group(<Integer>) at <Expression> , <Expression>

```
Segments
   Group(4) at 10*13.5 , 11
   Group(2) at Property(1,Length),10
End
```

Arc_Template is used to describe anything from a small arc through to a circle or an ellipse. A bounding box is specified to encompass the entire ellipse or circle, not just the shape. The quarter, half and full circles in figure 3.5 overlap the bounding rectangle because all measurements are taken from the middle of the arc, not the outside.

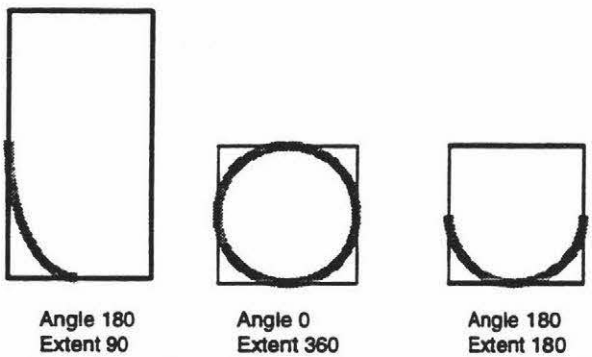


Figure 3.5 - A bounding rectangle encompassing different icons.

The arc is specified with two different types of angles. The first is the start position (*start*) and the second is the extent (*extent*), or angle of the arc. Figure 3.6 shows how the angles are calculated for the arcs in figure 3.5.

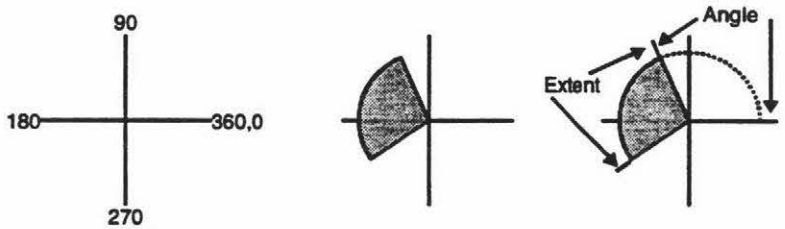


Figure 3.6 - Calculation of the angles on an arc.

Normal arcs can be described by having an extent of less than three hundred and sixty degrees or exactly three hundred and sixty degrees for a circle. The circle or arc can be warped into a more elliptical shape by changing the bounding box from a square to a rectangle.

The grammar and some examples of arc templates are shown below.

```
<Arc_Template>        ::= Arc in box <Expression> , <Expression> , <Expression> ,
                          <Expression > Starting <Expression> Extent <Expression>
                          Thickness <Expression> Colour Line <Colour> Fill <Colour>
                          Pattern <Pattern>
```

```
Segments
    Arc in box 0,0,20,20 Start 0 Extent 87 Thickness 1 Colour Line "red" Fill "blue" Pattern ""
    Arc in box EQ(9)-10,0,EQ(10),20 Start 90 Extent 180 Thickness 2 Colour Line "red" Fill ""
            Pattern "gray25"
End
```

Thickness refers to an offset from side of the projected arc border. For example, a thickness of three indicates the border line plus one pixel either side. There are two different colour specifications: the arc border and its internal fill. Either of these may be blank, eg. fill "", which results in that particular specification not being drawn.

The last configuration option for the arc template definition is pattern which refers to a bitmap that is logically anded to the border as well as the fill. There are a variety of predefined bitmaps that can be used, some being available through Tk (the language the graphical user interface is built from) and others in X11. An example of the application of pattern, using gray25 (only twenty five percent of the pixel's are shown), is shown in figure 3.7.

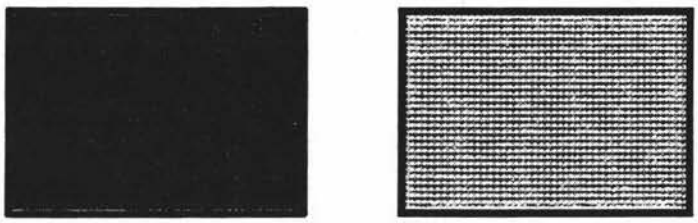


Figure 3.7 - The application of the gray25 pattern.

The Line_Template segment describes either a single line between two points or a poly line, containing multiple line segments. The main difference between these two types of line, apart from the number of coordinates listed, is that the poly line allows the specification of a fill colour. This will fill, with the specified colour, either a fully or partially enclosed area, as shown in figure 3.8.

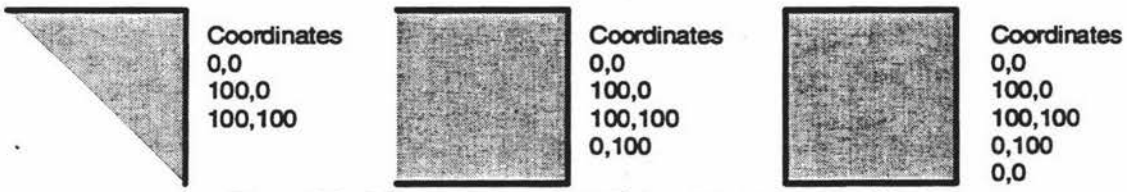


Figure 3.8 - Different ways of specifying objects and their fill.

For a single line the Colour option describes the colour of the line. There is no possibility of having an internal fill. Likewise with the pattern, it only has an effect on the line. Arrows may be placed at either, both or none of the ends. This was implemented as an option because the specification of an arrow on a line, by describing with segments, is difficult and it is a feature, especially for connections, that may be used frequently.

For a poly line the colour has to be specified for both the line and the internal fill. Because of these extra parameters the Pattern option now alters the line and the fill. Both sets of examples are shown below along with grammar describing the line template.

```
<Line_Template>      ::= Line <Expression> , <Expression> , <Expression> , <Expression>
                         <Line_Variations>
<Line_Variations>    ::= <MoreExpr> Thickness <Expression> Colour Line <Colour> Fill
                         <Colour> Pattern <Pattern> Arrows <Arrows> | Thickness
                         <Expression> Colour <Colour> Pattern <Pattern> Arrows
                         <Arrows>
<MoreExpr>           ::= , <Expression> , <Expression> | , <Expression> , <Expression>
                         <MoreExpr>
<Arrows>             ::= Beginning | End | Both | None
```

```
Segments
    // Simple lines with varying attributes.
    Line 10,0, EQ(9),0 Thickness 2 Colour "black" Pattern "" Arrows none
    Line 0,EQ(7)-10, 20,EQ(8) Thickness 1 Colour "red" Pattern "gray50" Arrows beginning
    // Lines creating a rectangle and triangle, respectively.
    Line 0,0, 100,0, 100,120, 0,120, 0,0 Thickness 1 Colour Line "blue" Fill "red" Pattern ""
            Arrows both
    Line 0,EQ(1), 100,EQ(1), 100,EQ(1)+100, 0,EQ(1) Thickness 2 Colour Line "black" Fill ""
            Pattern "gray25" Arrows end
End
```

The Text_Temp and TxtG_Temp segments are for the rendering of text strings on the canvas. Text_Temp allows for the specification of a single line of text, eg. the name of a class. The first part of the definition is the location coordinates to anchor the text field.

Next is the Width option which is needed for the placement of a scrollbar, if the text exceeds the maximum allowable width. The MaxX and MinX options place constraints on the physical size of the text in measurement of pixel's. If the maximum 'X' value is zero then there is no limitation to the size of the text but the minimum specification still applies.

The grammar for the definition of a single line of text is shown below followed by two examples.

```
<Text_Template>      ::= Text at <Expression> , <Expression> Width <Expression> MaxX
                         <Expression> MinX <Expression> Style " <StyleType> " Format
                         ' <FormatType> ' ID <Integer> Font <String>

<StyleType>          ::= N | <StyleAll>
<StyleAll>           ::= B <StyleUI> | U <StyleBI> | I <StyleBU>
<StyleUI>            ::= U <StyleI> | I <StyleU> | ε
<StyleBI>            ::= B <StyleI> | I <StyleB> | ε
<StyleBU>            ::= B <StyleU> | U <StyleB> | ε
<StyleB>             ::= B | ε
<StyleU>             ::= U | ε
<StyleI>             ::= I | ε

<FormatType>         ::= L | C | R

<String>             ::= " <String_Parts> "
<String_Parts>       ::= <AlphaNum> <String_Parts> | ε
<AlphaNum>           ::= <Letters> <AlphaNum> | <Integer> <AlphaNum> |
                         <Special_Characters> <AlphaNum> | ε
<Letters>            ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q |
                         r | s | t | u | v | w | x | y | z | A | B | C | D | E | F | G |
                         H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V |
                         W | X | Y | Z
<Special_Characters> ::= ` | ~ | ! | @ | # | $ | % | ^ | & | * | ( | ) | { | } | [ | ] | ; |
                         : | < | > | , | . | / | ? | - | _ | + | = | \ | ' | '
```

```
Segments
    Text at 10,10 Width EQ(9)-10 MaxX 300 MinX 10 Style "BU" Format 'R' ID 1  Font ""
    Text at 5,EQ(3)+10 Width EQ(6) MaxX Property(1,Length) MinX 0 Style "N"
            Format 'C' ID 991  Font ""
End
```

Style refers to the attributes of the text. Different font styles 'N' for normal, 'B' for bold, 'U' for underlined and 'I' for italics have been defined. All of these attributes except normal can be used together, eg 'BU' and 'IUB'. Format specifies the justification of the text with: 'L' for left, 'C' for centre and 'R' for right, being the only variations.

The templates definitions include the specification of the style, format and font attributes but this information is not supported in the graphical user interface prototype. There are also no facilities for the specification of point size.

The last parameter is the ID number associated with each text definition. This facilitates the isolation of the text field so that it can be used for various actions described within the template file. These actions, which are discussed in more detail later, are tied to a particular region.

Clicking on this region will call the specified action with the parameters given in the definition file. These parameters can be either a text string or an ID number. If an ID number is defined then the current value held in this text field will be passed and the action, if designed to, can update the information.

The TxtG_Temp definition is just a variation of Text_Temp allowing for a list of text instead of only a single line. There are only two major differences: the specification of a bounding box instead of a single anchor point and width. This is required because at any one time there can be up to two scrollbars, one horizontal and one vertical, attached to the text list. The bounding box allows for these scrollbars to be placed at the outer perimeter of the list, as shown in figure 3.9.

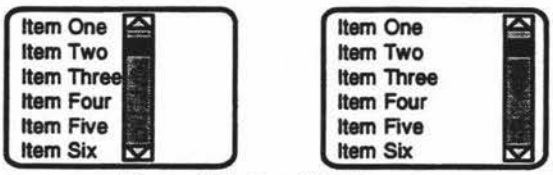


Figure 3.9 -Scrollbar layout.

The second difference is the additional constraints placed on the minimum and maximum vertical size of the list of text. The new constraints behave in the same manner as those for the maximum horizontal size detailed earlier. A value of zero for the maximum vertical size sets no maximum constraints.

The grammar and two examples of a text list box are shown below.

```
<List_Template>          ::= List at <Expression> , <Expression> to <Expression> ,
                             <Expression> MaxX <Expression> MinX <Expression> MaxY
                             <Expression> MinY <Expression> Style " <StyleType> " Format '
                             <FormatType> ' ID <Integer> Font <String>
```

```
Segments
    List at 0,0 to EQ(1),max(Property(1,Length),Property(3,Length)) MaxX 0 MinX 0
            MaxY 132 MinY 95 Style "U" Format 'L' ID 2 Font ""
    List at 0,0 to 30,30 MaxX 30 MinX 0 MaxY 30 MinY 0 Style "B" Format 'R' ID 46 Font ""
End
```

A single line of text that wraps around onto one or more lines has not been identified, or implemented, at this stage. It has been left out at this stage because it was not considered one of the essential segments needed in this stage of the prototype. The usage of this type of text field is illustrated in figure 3.10, which shows a process bubble.

Figure 3.10 - An example of single line wrap around text.

This functionality can not be fulfilled by the list text because it treats each line of the list as an individual item. Manipulation of this information would prove tedious and there would be no way of identifying which was meant to be list or single line wrap around text. A better alternative would be another attribute for a single line of text that specifies line wrap.

If an individual segment component does not create the desired graphical item then several segments can be used to build up the necessary graphics. Figure 3.11 shows one example of the usage of this technique. The position of declaration is the important factor here since the segments are drawn in the order they appear in the template file.

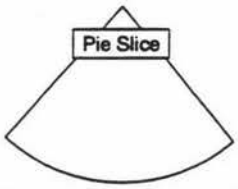


Figure 3.11 - An arbitrary graphical icon.

3.2.1.3 Future Enhancements

Iterative Declarations

At this stage of the tools development there is no capability to define an iterative definition for a group of segments. This type of definition would facilitate objects that had either a set number of the same type of segments or a varying number of segments.

The latter specification could be used to draw better Booch bubbles. Currently the user would have to specify the number of curves along the top bottom and sides of the bubble. When the bubble is expanded in size to accommodate the given text its shape can become warped as shown in figure 3.12.

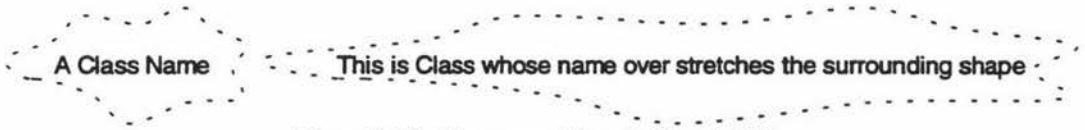


Figure 3.12 - The warped Booch Class bubble.

To address this problem an *iterate* segment is suggested. This would provide a simple loop in which a common element can be recursively defined. Figure 3.13 illustrates a Booch object bubble with the iterate segments shaded.

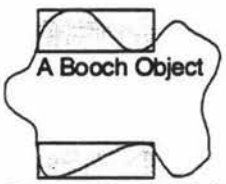


Figure 3.13 - A Booch object with iterate segments.

If the text inside the Booch class is extended then the iterate segments can insert another copy of themselves. The end result, shown in figure 3.14, is more natural without any of the warping that was produced in figure 3.12.

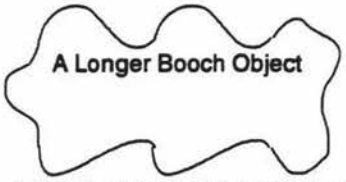


Figure 3.14 - A Booch object with minimised warp effect.

To define this within the template language one new construct would have to be added along with another function. The example of the iteration functionality being used is shown below.

```
Template Object "Booch Object"
        Equations
            = IterateNumber() * .....
            .....
        End
        Segments
            Arc .....
            ..............
            Iterate ( Property(1,Length) - 150 ) / 100
                Arc .....
                .....
            End
        End
End
```

The *Iterate* statement specifies that the following enclosed segment definitions will be repeated zero to many times. The exact number of iterations can be calculated by the expression that proceeds the *Iterate* statement. Any value to the right of the decimal point is truncated. If the user does not want the information to be truncated rounding functions can be used to alter the number.

The length of the text field is used to calculate how many iterations are necessary. One hundred and fifty pixel's are removed to allow space at either end of the bubble. The remaining value is divided by one hundred, the length of the iterate segments. With a few more calculations the iterate segments can be slightly stretched to take up any slack.

The *IterateNumber* function returns the number of the current iteration and can be used in any part of the definition file. If the *IterateNumber* function is used before an *Iterate* statement then a value of zero will be returned. If it is used after the iterate statement it will return the total number of iterations plus one. For example after the statement 'Iterate 3' the *IterateNumber* function will return four.

Curved Lines

Specifying the characteristic shape of some objects by arcs might be a very tedious job. One such example are Booch Class or Class-&-Object bubbles shown in figures 3.12 and 3.13. It would be much simpler to specify the border region of the icon by a group of points and join them with curved lines. Figure 3.15 illustrate how a Booch object can be drawn using fixed points joined together with line segments. Once these segments are specified as curved lines the diagram takes on a new appearance.

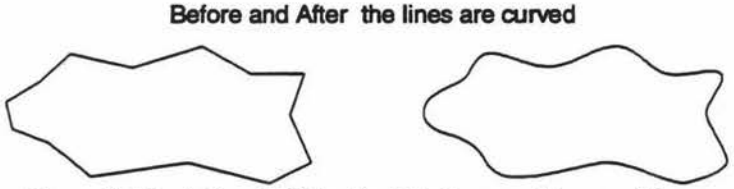


Figure 3.15 - A Booch Object bubble drawn with curved lines.

This feature could be incorporated into the current tool by adding another parameter to the line segment, eg a parameter 'curves', followed by true or false. The current interface tool, Tcl and Tk, support this type of specification for lines.

3.2.2 Template Object

In this section, which is optional, individual object templates are defined. Definitions of objects such as process bubbles, Coad and Yourdon class-&-object's, Booch classes or DFD data stores can be specified. The start of the definition includes the name of the object.

The grammar for the template definition along with an example is shown below.

```
<Object_Templates>      ::= Template Object <String> <Object_Collection> End
                            <Object_Templates> | ε
<Object_Collection>     ::= <Equations> <Segments> <Regions> <Redefine_IDs>
```

```
Template Object "Coad and Yourdon Class"
    .......
End

Template Object "Class-&-Object"
    .......
End
```

There are four sub-sections within the object template definition: equations, segments, regions and ID redefinition. The first two have already been discussed in the group template section.

3.2.2.1 Regions

Regions relate specific functionality to a rectangular region of the screen. If these areas are clicked upon a specified action with predefined parameters will be performed. An example of several region definitions and the grammar are shown below.

```
<Regions>               ::= Regions <Each_Region> End | ε
<Each_Region>           ::= = <Expression> , <Expression> , <Expression> , <Expression>
                            Action <Integer> With <TextOrID> | ε

<TextOrID>              ::= <String> <TextOrID> | <String> | ID( <Integer> ) <TextOrID> |
                            ID( <Integer> )
```

```
Regions
    = 0, 0, 20, 20 Action 23 with "Enter class name" ID(1) ID(1)
    =10, 10, EQ(2), EQ(6) Action 12 with "List of all class information" ID(1) ID(2) ID(3)
End
```

The specified region is only a rectangular shape. In the future other shapes could be supported by referring to a predefined group template instead of actual coordinates. The action number given refers to a position in an action list. These actions may include pop up menus, change of windows or programming code windows.

After the "With" option a list of parameters are passed to the action. The result of clicking a mouse in the example region definition is shown in figure 3.16. In this example the first parameter is the name of the question, the second is the highlighted information in the entry box and the last is what to update. The "ID(X)" refers to the text item, either single line or list, with the corresponding identification number. If any regions overlap then the first matching one found is used.

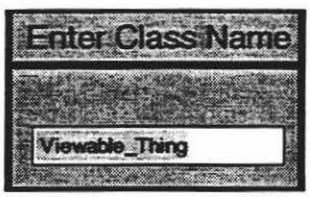


Figure 3.16 - Region example definition result.

3.2.2.2 Redefining IDs

This sub-section is used to supply information for text item templates. When the user specifies they wish to create a new instance of an object the tool must know what textual information to ask for. For instance in a Coad and Yourdon class there are three text fields. If a new instance of this class is created then the user would have to be prompted for the information in these fields.

Redefining IDs allows a name or question to be placed in the prompt box. An example of the Redefine ID section for a template file and the corresponding grammar is shown below. After this is Figure 3.17 which shows how the example below is used in the case of a Coad and Yourdon class.

<Redefine_IDs> ::= Redefine ID <Redefine> End | ε
<Redefine> ::= <Integer> Named <String> <Redefine> | ε

```
Redefine ID
    1 Named "Name of the Class"
    2 Named "Instance Variables"
    3 Named "Methods"
End
```

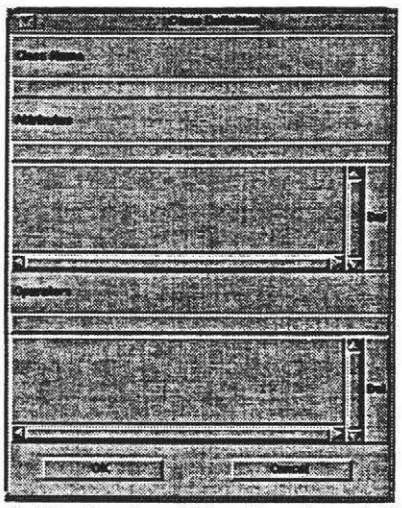


Figure 3.17 - Coad and Yourdon class definition.

The ID redefine specification can be considered a form of predefined action. An action that can not be changed and one which is performed by the CASE tool when textual information is required.

3.2.2.3 Future Enhancements

Docking Zones

An area that has been investigated but not implemented is the location where a connection is to join onto an object. The easiest way would be to allow a connection to join to any part of the object it connects to. However this does not cater for methodologies that want to restrict the connections to set areas around the object called docking areas.

Take for example the Coad and Yourdon Gen-Spec relationship shown in figure 3.18 (A). The hierarchy is always shown top down. To accomplish this a docking area is specified, shown in figure 3.18 (B).

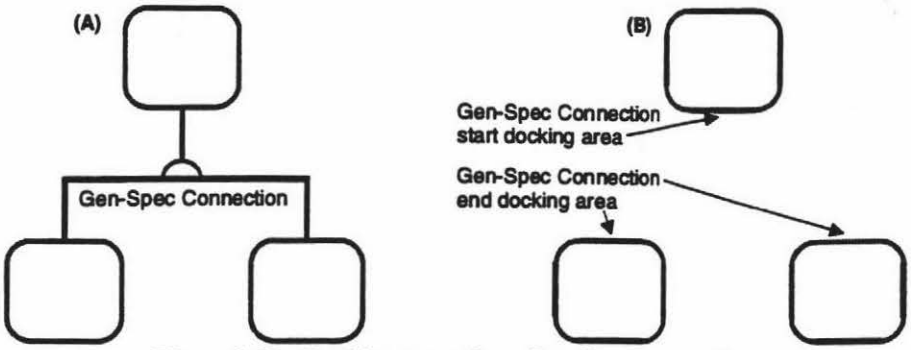


Figure 3.18 - Docking areas for a Gen-Spec connection.

If the user tried to specify a connection to a point on the object that is not valid, the outcome depends on the line type. For a single line the connection will be moved to closest docking area that will not cause the line to touch any other part of the object or cause an error. The result will be to inform the user that they have tried to connect to an illegal area on the object and that the connection has been refused to be drawn, figure 3.19.

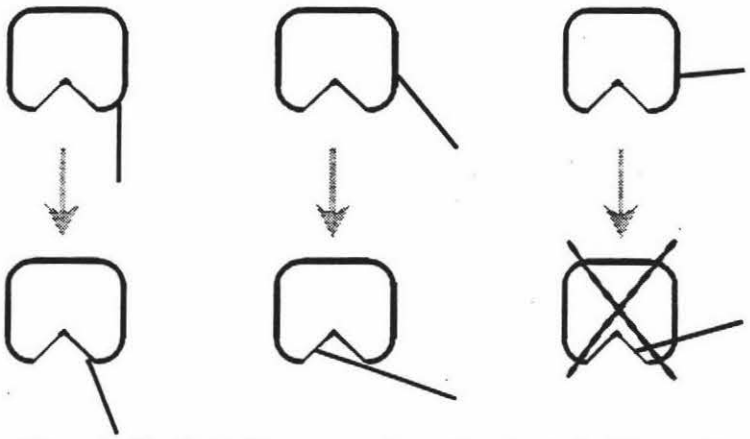


Figure 3.19 - Single line connection redirection to docking areas

A poly line connection would either be moved across to the docking area or additional segments would be added onto the line to intersect the docking area, as shown in figure 3.20.

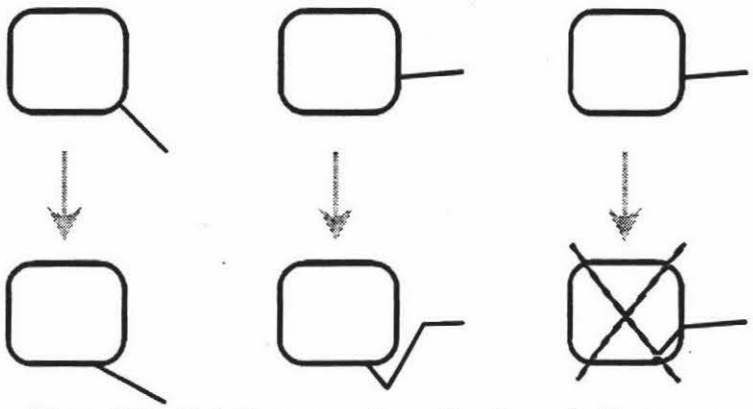


Figure 3.20 - Poly line connection redirection to docking areas.

The constrained poly line connection has the most complicated method for redirection. Connections can not just have one end point moved to the docking area since this would be a violation of the horizontal and vertical constraints that are specified for this line.

The priorities for this type of line are: move either the horizontal, vertical or both of the last two segments so they now allow the end point to touch a docking area, figure 3.21. The other alternative is to do the previously stated instruction and add one or more line segments to finish the docking procedure.

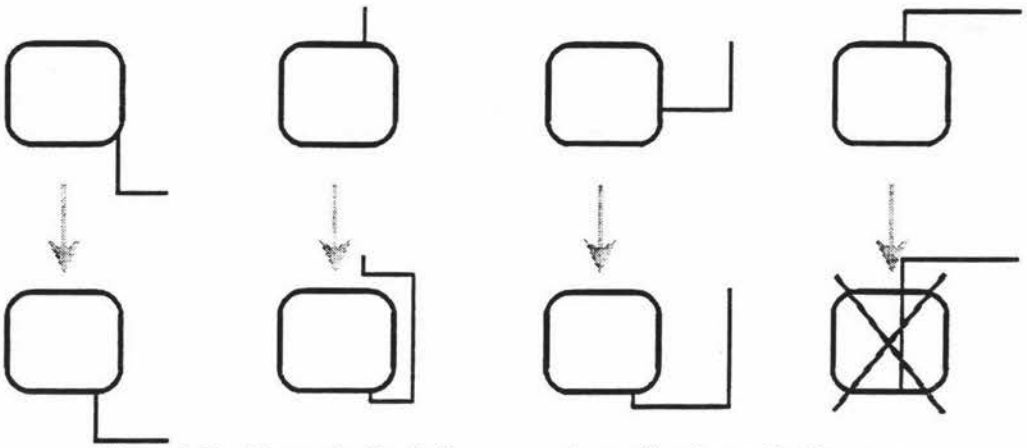


Figure 3.21 - Constrained poly line connection redirection to docking areas.

3.2.3 Template Connections

This template definition contains the specifications for connecting any two objects created from a template object definition. This type of template, which is optional, is the most complicated one because of the enormous amount of information that has to be available to facilitate the creation of connections.

The template definition allows a connection to have a specific name to allow the user to easily identify the functionality of each possible type. The first sub-section in a template connection definition is the optional equations definition.

```
Template Connection "Instance"
    Equations
        = 10 + Property(1,Height)
        = 10 + EQ(1)
        = 10 + EQ(2)
    End
    ......
End
```

A specific connection can vary in its look depending on which objects it is connected to. To facilitate the correct connection representation, more than one variation of the connection needs to be specified. Figure 3.22 shows how a Gen-Spec connection can have slight alterations depending on its target template object. Diagram (A) shows a Gen-Spec relationship that touches the outside of both abstract classes. Diagram (B) shows the connection penetrating the side of Class-&-Object.

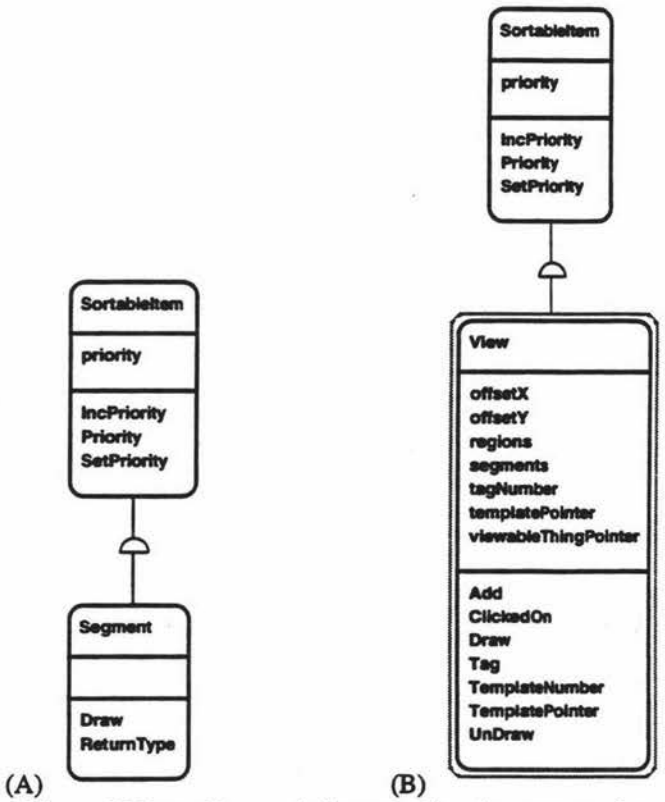


Figure 3.22 [MOOT]- Two variations of a Gen-Spec connection.

The grammar and an example are shown below.

```
<Connection_Template> ::= Template Connection <String> <Connection> End | ε
<Connection>          ::= Connect from ( <ListOfInteger> ) to ( <ListOfInteger> ) using
                          <ConnectLine> End <Connection> | ε
<ListOfInteger>       ::= <Integer> , <ListOfInteger> | <Integer>
```

```
Template Connection "Instance Connection"
   Connect from (1) to (1) using
      ......
   End
   Connect from (1,2) to (2) using
      ......
   End
End
```

The example shows two different ways of drawing a connection for an "Instance Connection". The numbers after the keywords "from" and "to" indicate which template object they can connect to. A one would indicate the first object template definition in the file, a two would be the second, etc.

Next the position of the start and end points need to be specified, the type of line, connection attributes and whether there are arrows on the connection. Re-positioning start and end points accommodates the type of model shown in figure 3.23.

In this case the line on the left does not just touch the edge of the template object but passes through the outer border to stop inside the object. Because of these problems the start and end points of a connection can stop at the side or deviate by a given value.

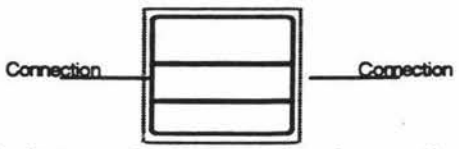


Figure 3.23 - An example of a repositioned connection end point.

The next specification option is for the type of line that is to connect the two template object views. At present there are three different types of line: a single line segment, a poly line made up of one or more segments and a constrained poly line. The constrained poly line has only horizontal and vertical segments. Figure 3.24 shows each of these line types.

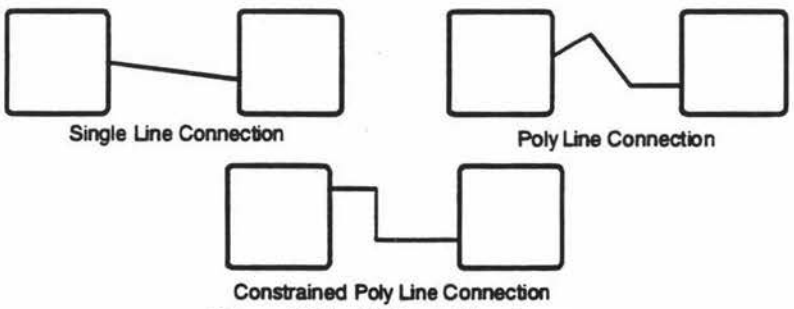


Figure 3.24 - Different line types.

The next part in the definition is a combination of three options that have a specification similar to that of a line. The three options are line thickness, colour and the pattern that will be applied to the connection line. The last option type, for connections, is for the specification of arrows on one, both or neither end of the connection. This is essentially the same as in the line segment definition. The grammar and an example of these four extra specifications are shown below.

```
<ConnectLine>      ::= Start <SFPoint> Finish <SFPoint> Lines <Lines> Thickness
                       <Expression> Colour <Colour> Pattern <Pattern> Arrows
                       <Arrows> <C_Segments> <Regions> <Redefine_IDs>
<SFPoint>          ::= Normal | <PorM> <Expression>
<PorM>             ::= Plus | Minus
<Lines>            ::= Single | Poly | HVPoly
```

```
Connect from (1,2) to (2) using
    Start Normal
    Finish plus 20
    Line HVPoly  //Constrained poly line
    Thickness 1
    Colour "Red"
    Pattern "gray25"
    Arrows none
    ....
End
Connect from (3) to (1) using
    Start minus EQ(3)
    Finish Normal
    Line Single
    Thickness 2
    Colour "Black"
    Pattern ""
    Arrows both
    ....
End
```

After the parameters have been defined the additional annotations are specified. This allows a large number of variations, some of which are shown in figure 3.25

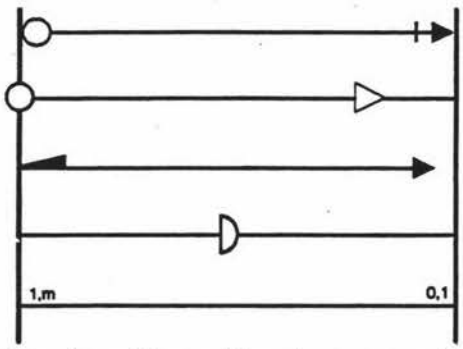


Figure 3.25 - Additional graphics used to enhance connection representation.

These extra graphical icons are composed from the template segments discussed in the group template section. The only change is the addition of location information which indicates the position of each individual icon.

Different annotations may be placed at either end plus or minus a specified value and an offset from the side. The value is either a percentage of the line length plus or minus an expression or just an expression. Figure 3.26 shows how values for the distance along the line and the offset are calculated. The start is always specified to the left and the finish to the right regardless to whether they actually appear this way on the screen. This standardised format is then rotated, except for the text which always remains horizontal, to the desired angle.

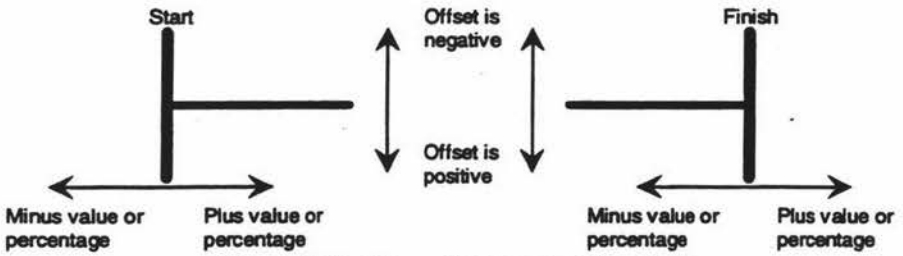


Figure 3.26 - Connection segment locations.

The grammar is shown below along with an example that shows the distance and offset connections drawn in for Figure 3.27.

```
<C_Segments>      ::= <C_Arc> <C_Segments> | <C_Line> <C_Segments> | <C_Text>
                      <C_Segments> | <C_List> <C_Segments> | <C_Group>
                      <C_Segments> | ε
<C_Arc>           ::= <Arc_Template> At <Point>
<C_Line>          ::= <Line_Template> At <Point>
<C_Text>          ::= <Text_Template> At <Point>
<C_List>          ::= <List_Template> At <Point>
<C_Group>         ::= <Group_Reference> At <Point>
<Point>           ::= <EitherEnd> <PorM> <NoOrPercent> Offset <Expression>
<EitherEnd>       ::= Start | Finish
<NoOrPercent>     ::= <Expression> | <Integer> Percent <PercentOption>
<PercentOption>   ::= <PorM> <Expression> | ε
```

```
Template Connection "A made up connection"
    Connect from (1) to (1) using
        Start Normal
        Finish Normal
        Lines Single
        Thickness 1
        Colour "Black"
        Pattern ""
        Arrows End
      Arc in box 0,0,20,20 Start 0 Extent 360 Thickness 1 Colour Line "black" fill "white" Pattern ""
          at Start plus 10 offset -10
      Text at 0,0 Width property(Length(1)) MaxX 0 MinX 0 Style "N" Format 'L' ID 1
          at Start plus 20 offset -20
      Line 0,0, 0,10 Thickness 1 Colour "black" Pattern "" Arrows none
          at End minus 20 offset -5
      Line 0,0, 20,0, 20,20, 0,20, 0,0 Thickness 1 Colour Line "black" Fill "yellow" Pattern ""
          arrows none at Start plus 50 percent minus 10 offset -10
    End
End
```

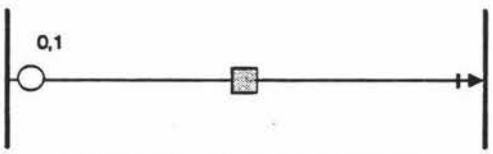


Figure 3.27 - Example connection

3.2.3.1 Future Enhancements

Curved Connections
Some methodologies use curved lines which are not supported by the current definition language. It has been decided that the curved line is going to be supported at this stage, because its functionality is handled basically the same way as a poly line.

Tk, the current interface management system, supports the facility of rounding a poly line by two extra options, to specify whether to round and how course the rounding is. Figure 3.28 shows how these can be easily adapted from the existing poly line definition.

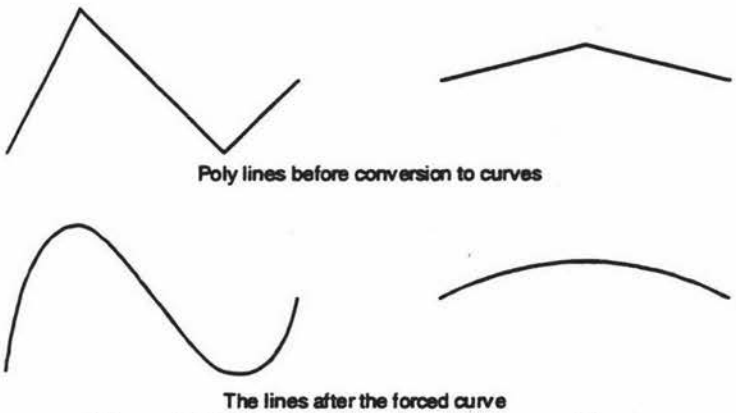


Figure 3.28 - Straight line changed to curved lines.

One to Many Connections
One type of connection that has not been addressed at this stage of the development is a connection from one object to many distinct objects. There are two ways in which a template file can specify this. The first option forces the user to make multiple connections, one at a time, where the connection specified is one to many.

To differentiate between connections that can be joined as one to many or one to one the template file will have to contain two separate definitions. These could be specified, for a Coad and Yourdon, as "Is-a" and "Is-a (1 to m)". Figure 3.29 shows these two different connections.

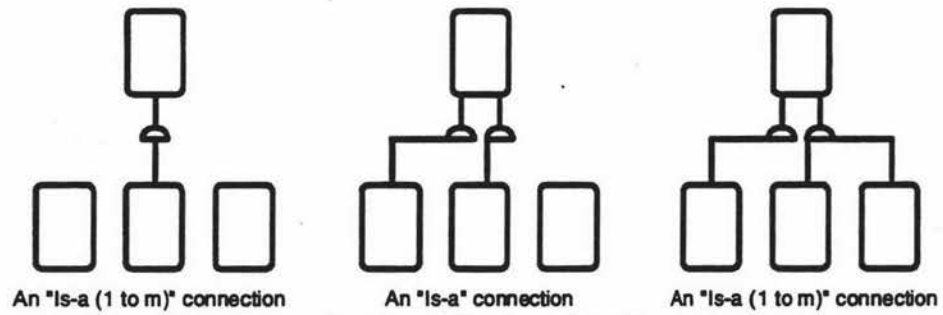


Figure 3.29 - The application of "Is-a" and "Is-a (1 to m)" connections.

However this does not address the problem of having multiple one to many connections, as shown in figure 3.30. To add another connection the user would have to select the connection they wish to add to and then add a new one.

This would eliminate the need to differentiate between one to one and one to many connections. For creating a new one to one connection the user would just add another connection, ensuring that no other connections are selected.

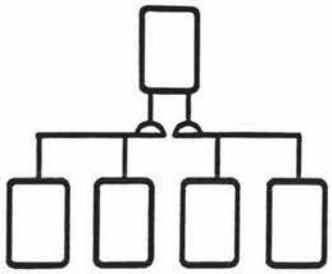


Figure 3.30 - Multiple one to many connections.

The final problem to solve is for deleting one of the many connections without having to rebuild the entire connection structure. The easiest way to do this is to highlight one of the ends of the connection to represent the control point.

When the connection is first selected the one part of the one to many relationship is highlighted with a circular symbol. Selecting again will move the symbol along each of the other connections. Once the symbol has reached the last connection it will move again to the beginning, as shown in figure 3.31.

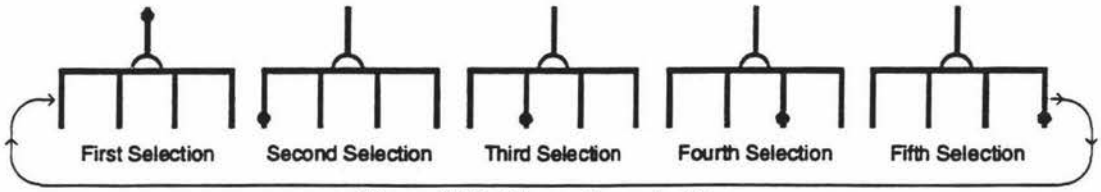


Figure 3.31 -Connection selection.

To delete a connection select it and then press the delete key. If the symbol is on the one part of the one to many relationship then the entire structure will be deleted. Figure 3.32 show the various combinations.

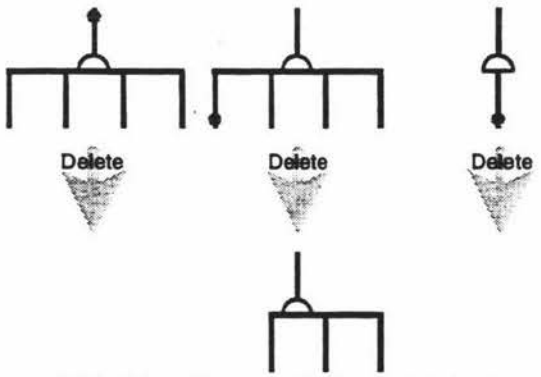


Figure 3.32 - The effects of deleting highlighted areas.

Reconnection mode is activated in a similar way by clicking the mouse on the object the user wants to make a connection to. To create a one to many connection the user first selects the object that will be the "one" part of the relationship and then the connection button. They then click on each of the objects in turn to generate a connection, shown in figure 3.33, and finally a cancel button to indicate they have finished the selection process.

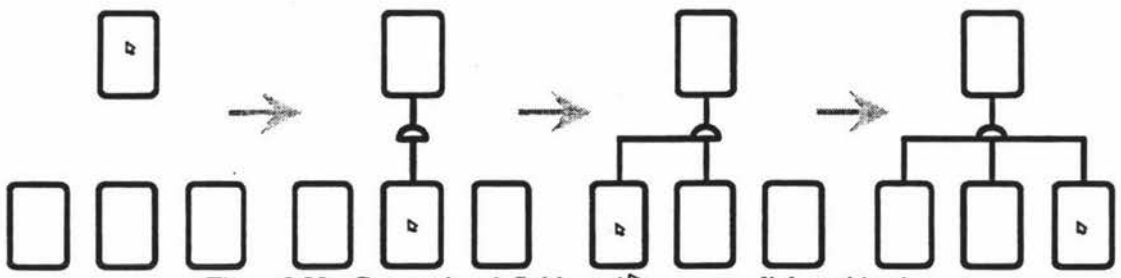


Figure 3.33 - Connection definitions (= mouse click position).

Out of the two different solutions this is the better choice because of the human computer interaction aspects. The user only needs a few button clicks to facilitate a one to many connection, rather than many for the previous.

Multi-Lined connections

All connections specified are built from only one actual line. There are, however, circumstances where more than one line in a connection is required. One such case comes from Booch methodology notation (1991) which is illustrated in figure 3.34.

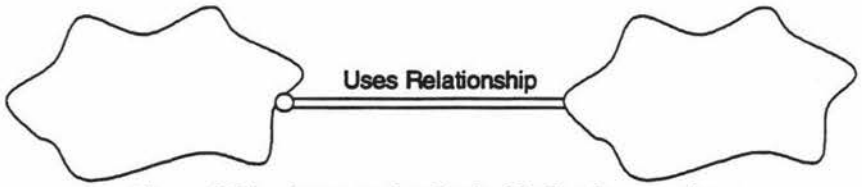


Figure 3.34 - An example of a double lined connection.

This double lined connection was superseded by a single lined connection in the next edition of Booch methodology (1994). Even though this connection type has been superseded there is a possibility using a similar connection in some other methodology notation.

To address this problem it is suggested an extra option is included along with the rest of the attributes. The new option would specify the offset that the connection line would be placed, as shown in figure 3.35. This could be repeated as many times as lines are needed.

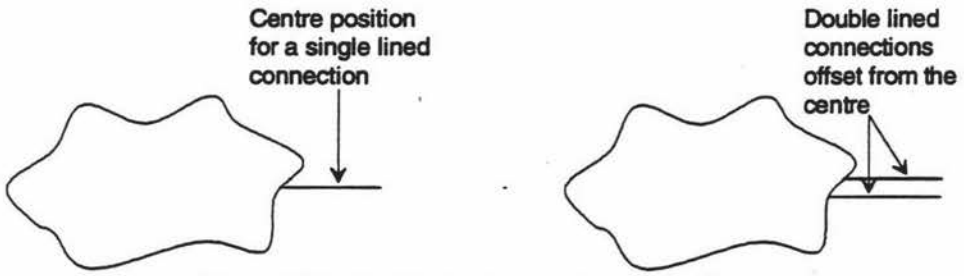


Figure 3.35 - Double lined connection using offsets.

This new option the inner part of the connection definition without including the information where the connection will connect to and from. Instead a list of attributes, including annotations is included, as shown in the example below. The listing below demonstrates an example definition.

```
Template Connection "Double lined connection"
        Connect from (1,2) to (2) using
                Offset plus 5
                Start Normal
                Finish minus 5
                Line Poly
                Arrows none
                ..... // The annotations
                Offset minus 5
                Start Normal
                Finish Normal
                Line Poly
                Arrows none
                ..... // The annotations
        End
        Connect from (3) to (2) using
                ----------
        End
End
```

4. Graphical User Interface

Modern CASE tool development, apart from requiring well formulated methodologies to form the basis for the tool itself, is critically related to issues of human computer interactions. A CASE tool which supports different methodologies and whose graphical interface (GUI) has a consistent look and feel reduces the amount of effort required to fully utilise these methodologies. Each methodology may use distinct notation but the interaction mechanisms and items are the same. This chapter addresses the design aspects and the implementation of the CASE tool's GUI. The interface basics, dialog control, graphical visualisation and object manipulation are further explored and discussed.

The interface for the methodology independent object oriented CASE tool is implemented through a graphical front-end class called Context. This allows the implementation details involved in the graphical interface to be hidden from the rest of the system. The interface to this front end is the GUI communication layer set out in section two, describing the structure of the tool.

The CASE tool has an event driven interface. The system is run by making an instance of the interface executing it. After this, the only time any C++ code outside of the context is used is when an event triggers it. Such encapsulation, which is an important part of the object orientated paradigm, adds great flexibility to the tool. If the interface interaction needs to be changed then only the Context class needs to be updated.

The choice of which interface tool to use was a very important decision. Consideration was given to flexibility, portability, ease of learning and use and speed of implementation. Presently the Context uses the tool command languages Tcl7.3 and Tk3.6 to facilitate graphics control and manipulation. Tcl is a general purpose language that contains looping, evaluation, assignment and system control mechanisms. Tk is the graphics extension to Tcl that allows a wide range of graphical functions to be used with the Tcl commands.

Tk is based around the X protocol with some of the functionality being developed because a similar feature is available in X. This is not a problem when considering porting between different systems. Tcl and Tk already exists a wide range of systems: MS-Windows, VMS, Cray CM5, DOS and Macintosh.

To help in building the interface a visual programming tool was used. This tool, xf2.3, allows the developer to visually place and manipulate the individual parts of the interface. Using this tool save a large amount of time.

Recognition and appreciation for interface related tools and code goes to:

- John Ousterhout from University of California, Berkeley, now working for Sun, who design and implemented Tcl and Tk.
- Sven Delmas from Technische Universität Berlin, Germany who built xf2.3. This tool offers a graphical environment where all aspects of Tcl and Tk can be build through visual manipulation. This tool increased the speed of the graphical development. This is based on work done with BYO developed at Victoria University in New Zealand.
- Stephan Herrmann from Technische Universität Berlin, Germany who supplied the initial C++ interfacing code to Tcl. This code has since been modified to accommodate the functionality of the CASE tool but the general design has not been updated.

These people were mentioned because without their "free" contribution a lot more work would have been involved in getting the interface to the current stage.

4.1 Interface Basics

The CASE tool employs an event driven architecture to control the system during execution. At the initialisation step the Tcl code specifying the interface characteristics is loaded into memory and executed. From this point only expected events will trigger an action.

The CASE tool is implemented with an event driven interface because all actions have to be initiated by the user. Creating an object, quitting the tool, moving an object around the screen, etc are actions causing events. There are no actions that are not initiated by the user.

Figure 4.1 shows the relationships in the event driven interface.

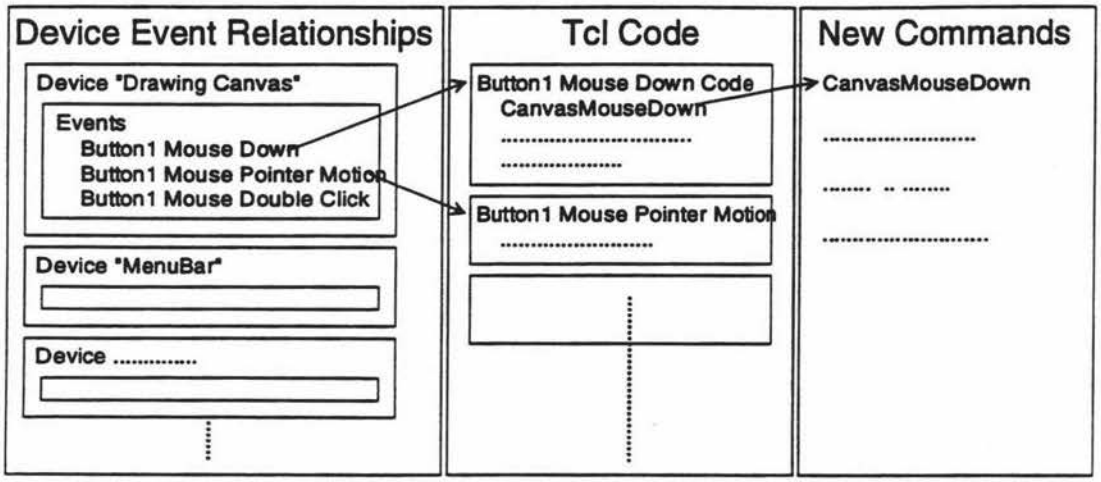


Figure 4.1 - The relationships of an event driven interface.

To facilitate the interaction between Tcl and C++ new Tcl commands had to be written. The execution of these commands call the appropriate C++ code that has been specified in the initialisation phase of the CASE tool. This binding between Tcl and C++ has the following parts:

- A Tcl function call which specifies the name of the Tcl function and the name of the C function that will be called. A C function is used, instead of a C++ method, because the Tcl language is implemented in C. It is therefore only possible to use a function as a parameter and not a member function pointer.
- A C function is written to act as a bridge between Tcl and C++. Its sole purpose is to call the appropriate C++ method in the specified class.
- A C++ method encapsulating the required functionality.

A graphical representation of the Tcl and C++ binding is shown in figure 4.2.

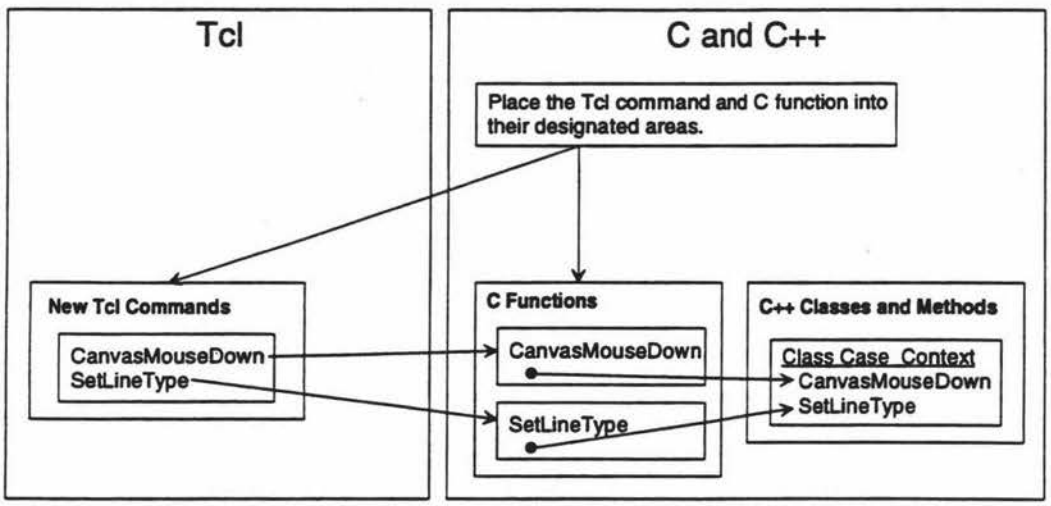


Figure 4.2 - Specification of commands between Tcl, C and C++.

One of the important issues to consider is the transfer of information between Tcl and C++. There are two methods for doing this: a request based service or variable links. The latter provides a link between variables in Tcl and in C++.

When the value of the Tcl variable is required it is fetched from the C++ variable. Likewise when the Tcl variable is altered the C++ variable is updated. When the C++ variable is manipulated or read no communication with Tcl is required since this is the main storage area for the contents of both variables. This approach was not utilised because of the redundant calls required. It is possible for a variable to be changed more than once before a C++ method is called. The variable link would then perform unnecessary communication slowing down the system performance.

A more efficient approach was utilised. The request based service method, which is the one used in the CASE tool, relies on the C commands which transfer information to or from Tcl. Since Tcl treats all variable information as strings the transfer is facilitated through passing string pointers. This method was used because it ensured only the minimal amount of transfer needed to take place, along with other basic features that have an impact on the interface. These are:

frame
: The frame method provides the location of the drawing canvas. When one of the graphic primitives is called this location is used to identify which canvas the segment is drawn on.

UnDraw
: This method is used to remove one or more graphical segments from the drawing canvas. The segments are identified by a tag, a string of alphanumeric characters.

GetWidth
: This method is essential in determining the result of the width text function, in an expression. The context and current font is used to calculate the width of the given string of characters.

GetHeight
: This is the same as GetWidth except that the height is calculated.

NewTag
: Tags are used to uniquely identify each object, connection or composite object in the given context. They take the form of strings containing the letters "TAG" and an identification number. This method supplies such numbers in an incremental fashion. At the current time this number is an integer, giving a possible sixty five thousand, five hundred and thirty six combinations. Depending on future requirements this could easily be changed to accommodate over four billion combinations.

evaluate
: This method is used internally to perform various Tcl commands. The method calls the Tcl function Tcl_GlobalEval which interprets the contents of the given string.

4.2 Dialog Control

Five different dialogs are used to present information to the user. These are:

- The main window the user is always presented with.
- Dialogs for the selection of object, connection or composite object.
- For the acquisition of textual information that is required to produce one of the forementioned selections.
- Dialogs associated with the functionality of regions on an object, connection or composite object.
- Dialogs that show errors.

All Tcl functions mentioned in this section have been implemented from scratch because they were not included in Tcl.

4.2.1 The Main Window

The main window shown in figure 4.3 is the first dialog presented to the user and the only one that will remain on the screen throughout the use of the system.

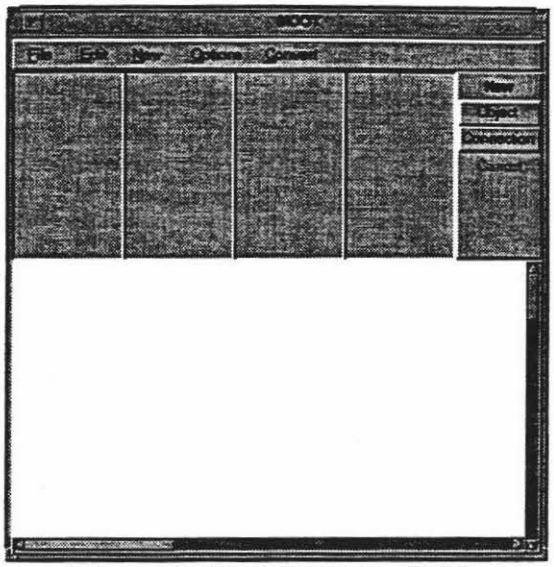


Figure 4.3 - The main screen window.

The window can be resized by the user at any stage, although some size restrictions exist. When the window changes its shape all the internal features will automatically readjust themselves to adapt to the new size. Figure 4.4 illustrates the window resized.

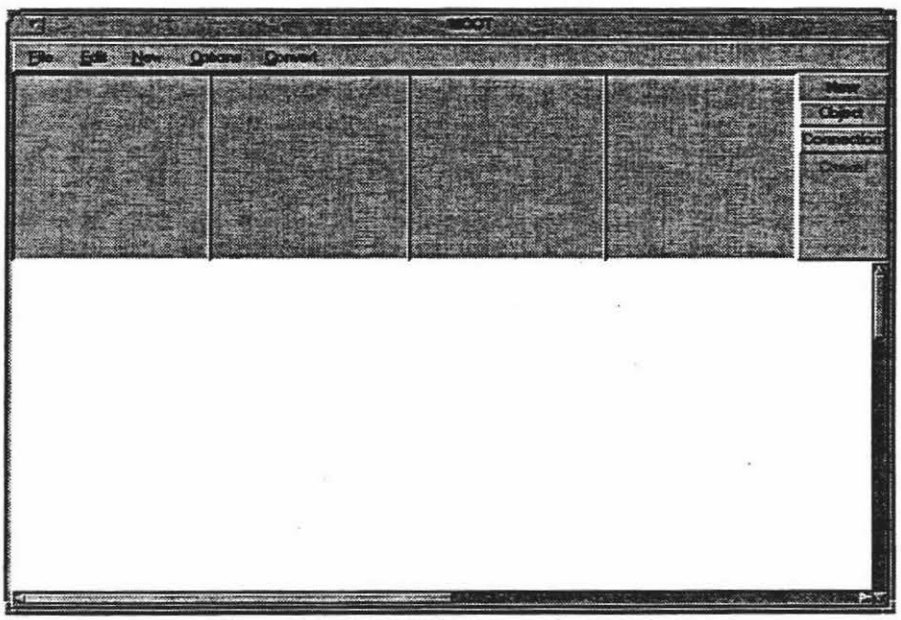


Figure 4.4 - The main screen after resizing.

Four areas can be identified:

- The menu bar.
- The model browser.
- The function and status bar.
- The canvas.

4.2.1.1 Menu bar

All functionality available to the user is facilitated through this menu bar, direct mouse manipulation on the canvas or by using the function and status bar. No help facilities have been added to the system since this is only a prototype. The final system will have this functionality.

Figure 4.5 - The menubar for the CASE tool.

The type of functionality that will be available, when implemented, is:

- The ability to open various project persistent stores and print the canvas contents.
- To handle the standard cut, copy, paste, select all and operations.
- Create new objects, connections and composite objects.
- Customise tool specific information.
- Provide conversion between various methodology notations.

4.2.1.2 Model Browser

The model browser, figure 4.6, provides a hierarchical model of the unused and graphically rendered objects, connections and composite objects. This facilitates reuse by providing the user with all information that has been stored for objects, connections and composite objects.

The general concept of the model browser is an adaptation of the class browser used by Smalltalk. The main difference is that the Smalltalk browser provides less abstract information with details going down to method implementation.

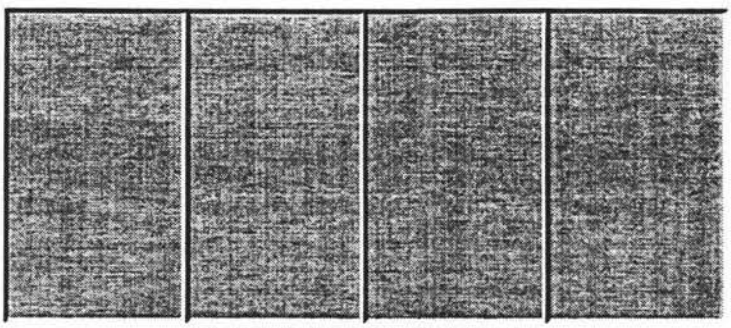


Figure 4.6 - The Model Browser.

The four panes represent distinct categories from an abstract to a specific level of detail. The first pane is a general grouping area representing highly abstract classifications. Figure 4.7 shows an example.

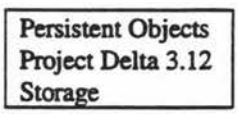


Figure 4.7 - An example of the first pane.

The contents of this pane would encompass an abstract overview of the common persistent store and the temporary persistent store. Information required from this store could be accessed by descending the hierarchy presented in this pane.

The second pane is the group item identification area. Once an item from the first persistent store is selected its individual parts are visualised (figure 4.8). If the grouping classification "storage" is selected then this column could contain items such as array, list or stack.

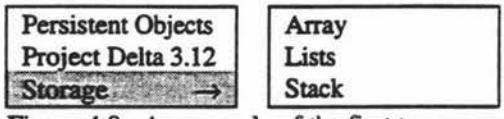


Figure 4.8 - An example of the first two panes.

The other alternative is to have temporary persistent store name in the first pane and the abstract list of classifications in the second pane. What will be visualised in the second pane will be dependant on what the temporary persistent store is describing. The third and forth panes will handle a larger content of information as another level of abstraction.

One type of object that has not been specified at this stage are templates. Even though this is a C++ specific type its unique features should not be ignored if this tool is to be used for commercial organisations. Templates use one definition to specify a multitude of different typed classes. Their ability to take on several types adds complexity to their specification in the model browser.

Simply placing them in as classes would cause problems with names. If the template was for an array of items then not only would the name need to change but so would the internal types of its storage attributes. A character instance, for example, would want to be called something like 'Array_of_Char' with an internal variable containing an array of characters.

One way to address this problem involves an abstract heading specifying the type of templates which would be placed in the second pane. In the following pane would be class instances of the templates that have already been produced along with the prototype (figure 4.9). Any instances that are not present can be created from a template prototype.

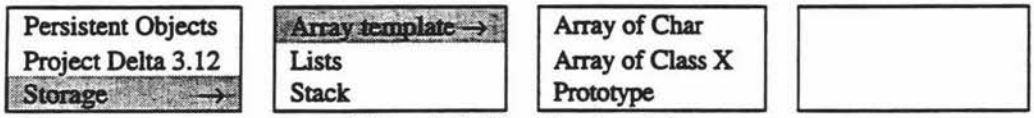


Figure 4.9 - Handling templates.

This method allows for the specification of the classes name and for particular types of its variables. All behaviour within the tool is acceptable. The complications occur when the code is exported, as C++, to files. Each of these would be sent as an individual class definition.

To eradicate this problem these definitions would have to be tagged as a product of a template. When the C++ code is written out to the various files the template prototype is used instead of the other specification.

4.2.1.3 Function and Status bar

The function and status bar, shown in figure 4.10, provides two services to the user of the tool. The first is to allow an easy method of creating objects or connections. When composite objects are implemented in the system another button will be added to accommodate them.

Figure 4.10 - The function and Status bar region of the main window.

These buttons are not the only means for creating objects and connections. They are available because these are extensively used actions. If the user does not know the short cut approach for the creation of objects and connections then these are a fast and easy alternative.

Any information needed for the creation of an object or connection or to print an error is presented in the status window. This is located directly below the 'Cancel' button on figure 4.13. This provides the user with immediate feedback without interfering with the creation process.

4.2.1.4 Canvas

The canvas for the main window, figure 4.11, is a mouse sensitive area for rendering graphics. All functionality in the form of creation and manipulation of objects, connections and composite objects on the screen can be activated directly from the canvas using the mouse.

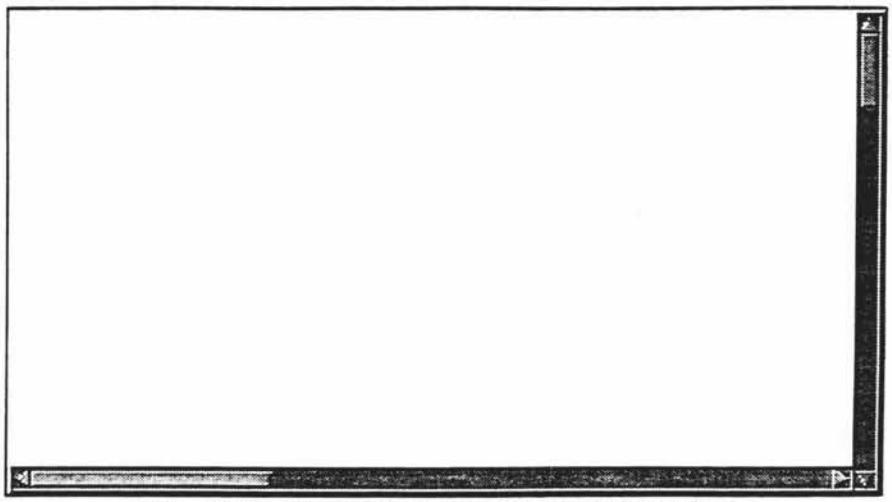


Figure 4.11 - The drawing canvas for the main window.

For example double clicking the mouse in an empty space on the canvas will create a new object. Double clicking on an object, on the other hand, will produce a new connection.

4.2.2 Dialogs for Creation

There are three different categories of dialogs which facilitate the creation of objects, connections and composite objects. Figure 4.12 shows an example of the dialog for a Coad and Yourdon Class&Object and Abstract Class selection.

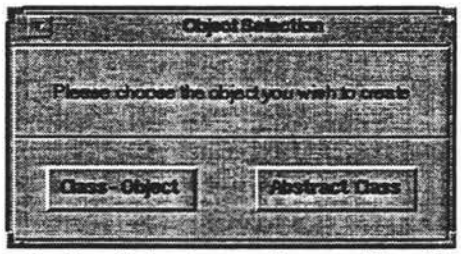


Figure 4.12 - The selection dialog for creating an object (Coad and Yourdon).

The creation dialogs use the same Tcl function which is an adaptation of the one presented in Ousterhout (1994). Figure 4.13 shows another example of the dialog to facilitate the construction of a Coad and Yourdon Connection.

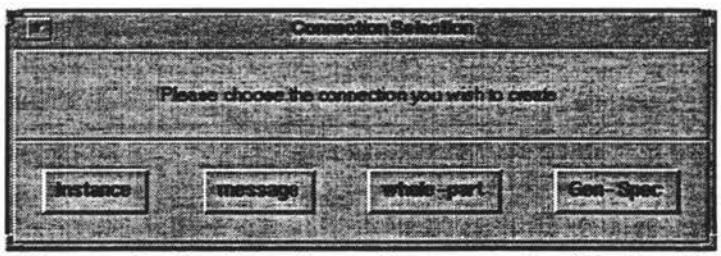


Figure 4.13 - The selection dialog for creating a connection (Coad and Yourdon).

4.2.3 Acquisition Dialogs

The textual information acquisition dialogs are dynamically constructed. To build and execute one dialog requires the functionality from up to seven individual methods. The example given is in accordance to the construction of an input window for textual information in defining a Coad and Yourdon class. The construction stages are outlined. Only the final window is presented to the user.

The first, Window_New, facilitates the initialisation and construction of a top level window. This has the default buttons, 'OK' and 'Cancel', which are positioned on all instances of this window. The method takes one parameter, a name, which is placed on the top of the window.

Window_New in turn calls a Tcl function, make_top_level, passing the window name and construction position. The latter argument is needed because Tcl places all graphical objects on a tree structure with the root being the top level of the existing window. The construction position indicates where the new top level branch will grow from. Figure 4.14 shows an example of the initial top level window.

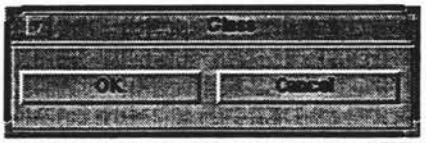


Figure 4.14 - The result of calling the method Window_New.

The next method is Window_Single_Entry. The purpose of this method is to produce a labelled entry field. There may be as many of these as is required to fulfil the information requirements of the template. This method, on execution, is passed the identification number of the text field and the question, or statement, which will be placed above the entry field. Window_Single_Entry then calls the Tcl function make_single_entry with the specified parameters, figure 4.15.

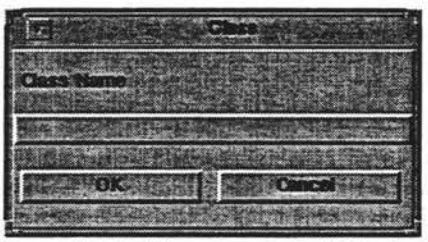


Figure 4.15 - The result of calling the method Window_Single_Entry.

Upon execution of the method Window_Multi_Entry an entry field and label, like those above, are added along with a scrolling list box. Each time the 'Enter' key is pressed in the entry field the contents of that field are added to the list box below.

In addition to the listbox, scrollbars and a delete button are also added. The scrollbars accommodate viewing long lines of text or too many lines. If an entry is incorrect then that item may be selected and the entry deleted, by clicking on the delete button.

The Window_Multi_Entry method takes two parameters, like the identification number and the question (or statement) which are passed to the Tcl function make_multi_entry. Figure 4.16 shows the implementation of the class attributes definition.

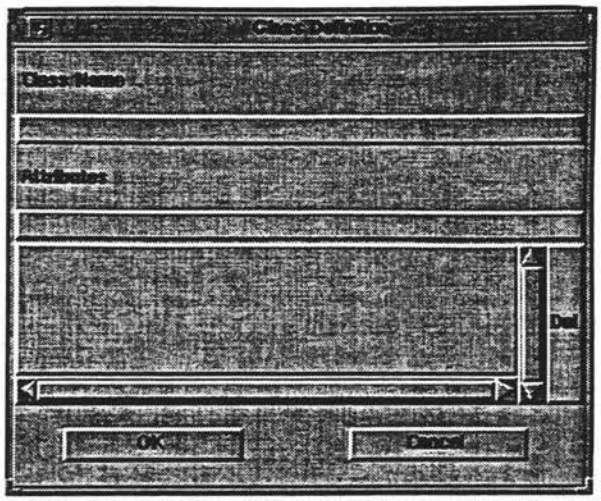


Figure 4.16 - The window with a multiple entry input box added.

To facilitate the complete Coad and Yourdon class another multiple entry field has to be added for the definition of the class operations. Figure 4.17 illustrates the completed entry window.

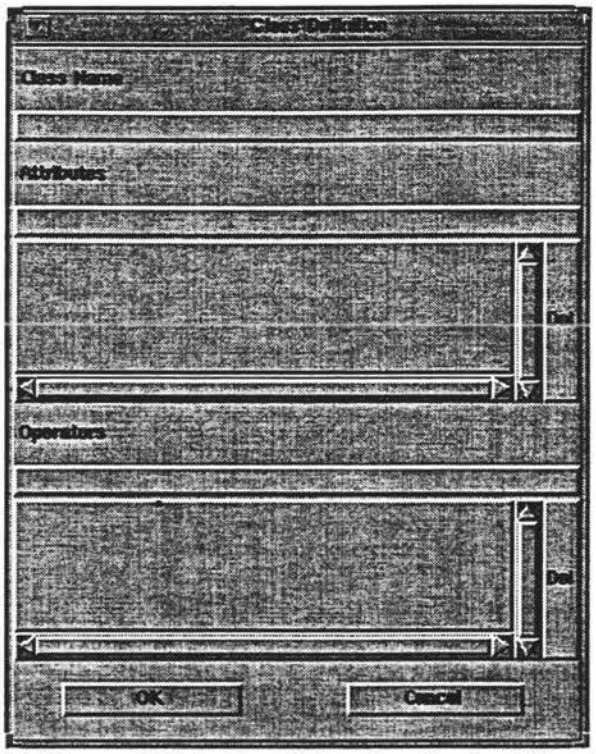


Figure 4.17 - The completed entry window for a Coad and Yourdon Class.

At the next stage control is passed to Tcl, through the method Window_Wait_Button. This allows the interface to respond to the users actions. The Tcl function wait_for_button allows the tab key to move the cursor between the entry fields. This allows the user to visit every entry field without having to select the required field with the mouse in a sequential order down the screen until the bottom most entry field is reached. The next cursor will be moved again to the top most entry field. Tcl function waits until one of either the Cancel or OK buttons have been pressed.

After the input session is completed, the method Window_Destroy is called to delete the interface related graphics. The top level window is destroyed and a cascading effect will destroy all related graphical items. The Window_Button_OK method is called to differentiate between OK and Cancel. The return value from this method is true if OK was clicked on or false if Cancel was selected.

The method load extracts all input from the last dialog session and places this information into the given Viewable_Thing. The extraction is possible because the information given in the input is stored in Tcl data structures. The C++ code identifies, from these structures, how many entry fields were defined, their related identification numbers and whether the field was single or multiple entry. The resulting information is used, along with a template, to construct a new view.

4.2.4 Dialogs for Region Functionality

At this stage no dialogs for region functionality have been implemented. The idea is to produce a set of generic functions for the manipulation of information. These functions would have a specified set of parameters which have to be defined in the definition file.

An example of one possible function would be a single line text edit window. The definition file would specify the function number for this (eg eleven) and the parameters (eg "Change class name to:", ID(15) and ID(15)).

```
Regions
    = 0, 0, 20, 20 Action 11 with  "Change class name to:"  ID(15)  ID(15)
End
```

Three parameters are required: the message presented to the user in the dialog box, the highlighted information that will be placed in the entry box and the position of the updated information. ID(15) refers to the text item identified with the number fifteen. It this case the last two parameters are identical. In case of different identifiers the updated value would be placed into another text item.

There is a large number of additional functions that would be necessary. Some of these are:

- List text manipulation dialogs.
- Adding in symbols to identify characteristics about list text items. An example of this would be to place symbols in front of C++ methods and variables to indicate which are private, protected or public.
- Functions to change the template object that defines the object on the screen, figure 4.18.

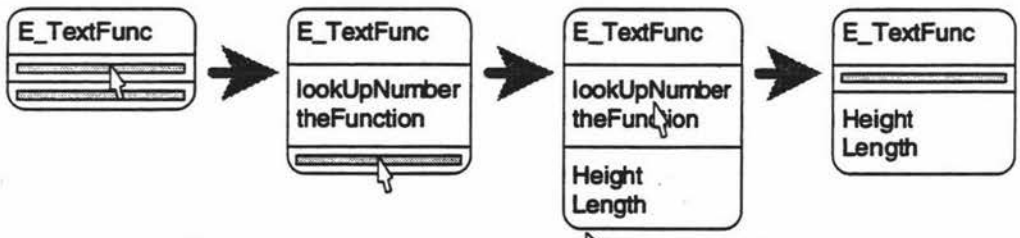


Figure 4.18 - Changing between template objects (= a mouse click at the given point).

- Show which other objects are connected to this object. This would include which connections and what textual information to display.

Specific dialogs have to be defined for the re-engineering module of the CASE tool. These have to display and allow changes to information that is not actually seen in the model on the canvas. The information that will be present is shown below along with mock-ups of the dialogs. The © symbol means the field can be changed whereas the ® symbol indicates the field is read only.

Class Editor

- © The name of the Class.
- © The base classes and their access specifier (public, private or protected).
- © Variables and their access specifiers. The variable editor can be invoked to create or edit individual variables.
- © Methods and their access specifiers. The method editor can also be invoked for this information.
- © A comment with the description of the class.
- © The declaration file.

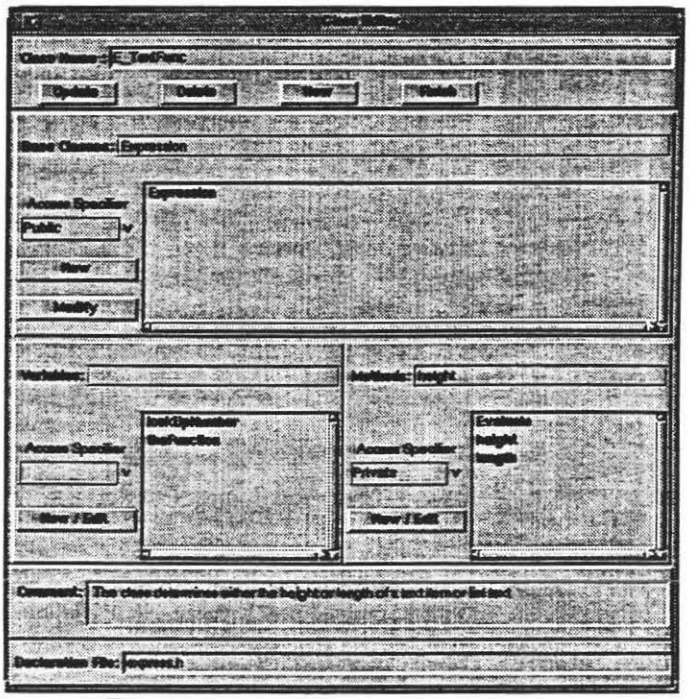


Figure 4.19 - A mock-up of the class editor.

Variable Editor

- © The variable name. Normally one variable is being edited. If the arrow to the right is selected a list of all other variables is shown. The user can selected one from this list.
- © Whether the variable is public, private or protected.
- © Its type.
- ® The actual line parse from the C++ file. This allows the user to see the context in which it was defined.
- ® A list of the methods that use this variable. The user can edit any of these methods by passing control to the Method Editor.
- ® Which class this variable belongs to.
- © A comment about the line the variable was declared on. If there is more than one declaration on the actual line of code then this will be a shared comment.

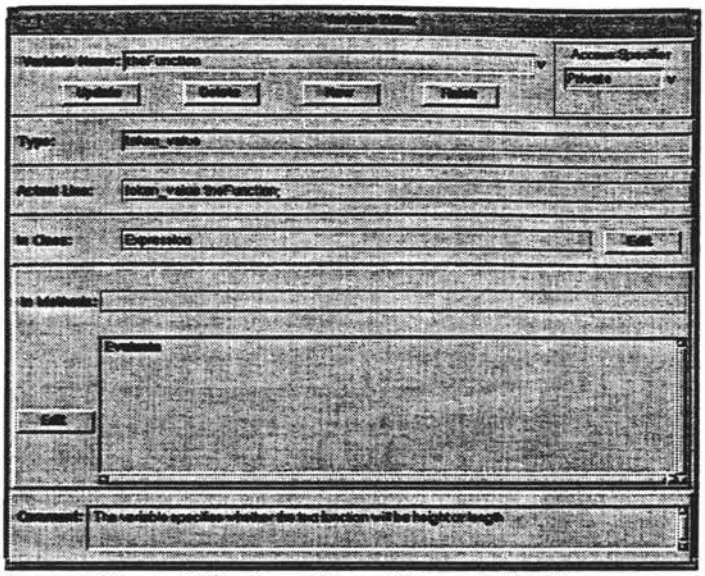


Figure 4.20 - A mock-up of the variable editor.

Method Editor

© The method name. This field operates in the same way as the variable name.
© The access specifier that the method is declared under.
© The return type.
® The actual line of C++ that was parsed.
© The class that owns this method.
® The parameter list for the method. Any changes for this list will be made by passing control to the Parameter Editor.
® A list of class declared variables that are used within the method. Any of these methods can be edited by passing control to the Variable Editor
© The actual code within the method.
© The comment about the method.

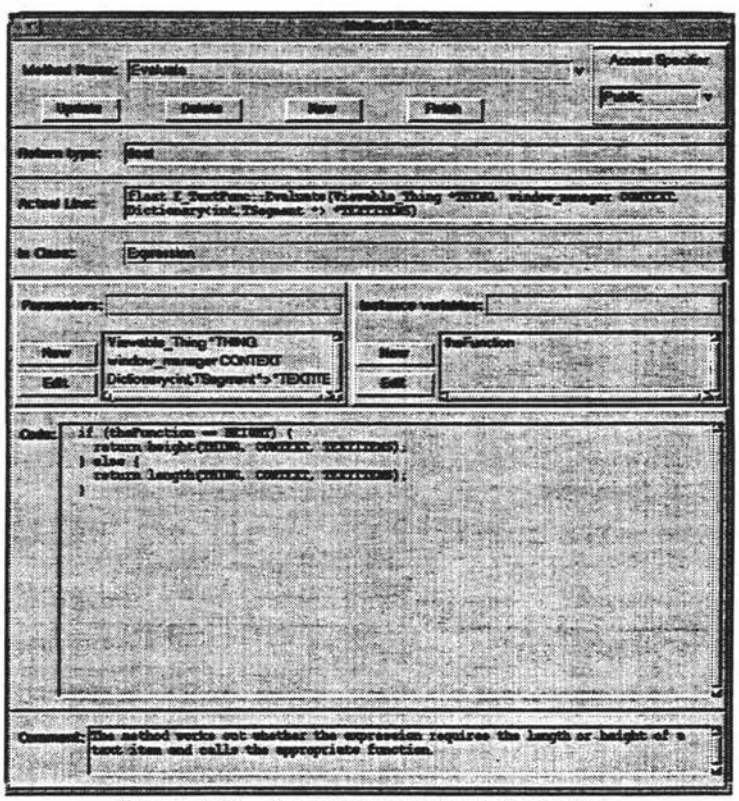


Figure 4.21 - A mock-up of the method editor.

Parameter Editor

© The name of the parameter.
© The data type.
© An optional default value.

4.3.4.1 Future Expansion.

No matter how many different types of dialogs are specified they will never be able to cater for all possible circumstances. The developer, or user, must be able to specify their own interaction dialogs that meet the needs of the methodology notation.

To handle this situation it is suggested that Tcl and Tk be used to implement other functionality. Tcl and Tk would be coupled together with a set of primitive operations that would allow access to all modelled information. The added functionality could then manipulate the current information to produce the desired results.

New functions could be developed and saved as Tcl code files. The CASE tool would be told of this files use during the tool configuration. Using currently available Tcl tools the code can even be examined and modified during use. This feature would greatly enhance debugging.

4.2.5 Errors

The final type of dialogs are for handling errors. These consist of an optional symbol, a message and an OK button. The symbol is either a Tcl or a X windows bitmap. Figure 4.22 illustrate two types of error dialogs.

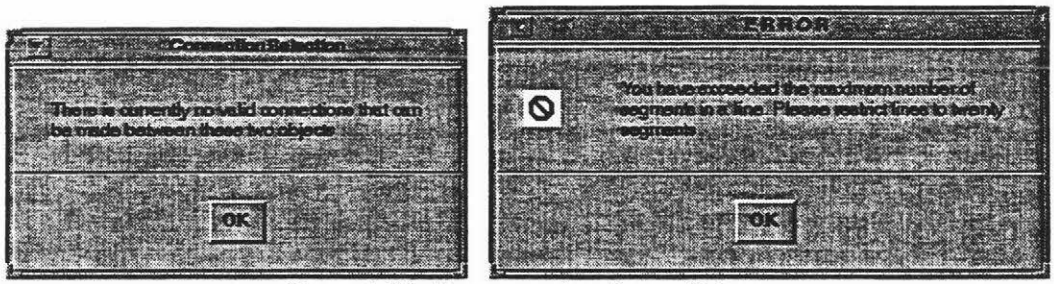


Figure 4.22 - Two examples of error dialogs.

4.3 Graphic Visualisation

All Graphic visualisation is facilitated through the use of predefined generic primitives in the Case_Interface class. This makes all methods for the visualisation of the graphics transparent to the methods that require this functionality. There are four generic graphic primitives: Arcs, Lines, Text items and List text. Each primitive is designed to supply all the functionality needed for a multitude of interface types. This allows the interface to be changed without the primitives needing to be updated

4.4 Object Manipulation

4.4.1 Graphic Creation

Presently the CASE tool supports the creation of two graphical renderings, objects and connections. Creation of a connection requires the selection of an object. All valid connections that can be made are displayed for the user to choose from. If it is a single segment line connection then the user is just required to click on the connecting object. If the connection is valid then there is a check to see if any textual information is required. If text segments are present in the connection then a dialog requests the desired information.

Once the textual information is acquired the connection implementation process begins. If the connection is invalid then an error is visualised and the connect process is cancelled. Poly line and constrained poly line connections are basically the same except a multitude of anchor points may be selected.

The object creation process requires the user to choose a new object type and then selects the anchor point. If there is any textual information required, this is asked for, and then the new object is placed at the given coordinates.

There are several different methods that can be employed for creating connections and objects: using the menu system, using the quick creation buttons (on the function and status bar) and by direct manipulation on the canvas. On the menu bar there is a menu *new* which initiates the creation of either connections, objects or composite objects. The sequence of events needed to create a single graphic rendering is quite drawn out, especially if a large amount of information needs to be created.

The quick creation buttons are implemented because the functionality they represent is frequently used and thus easy access is needed. This performs the same functionality that is present in the forementioned menu except it is more accessible and quicker to use. The first two creation methods are for less experienced user. The expert user requires shortcuts that will speed up the creation process even more. This is facilitated by allowing the user to create whichever graphic rendering they choose without leaving the canvas.

An object can be created by double clicking on the canvas. The coordinates of the mouse depression are where the object will be placed. The rest of the process follows normal object creation behaviour. Connections are initiated by double clicking on the selected object that the connection will start from.

At all stages of the creation process prompting is given through the status and function bar. The messages are kept short and precise.

4.4.2 Graphic Movement

Movement for objects is initiated by firstly selecting the object that is to be moved. By holding down the mouse while the pointer is on the object allows for movement to occur. Releasing the mouse button terminates movement at which time all connections to the object are updated.

The position to which the object is to be moved is shown by moving the actual object instead of just an outline. This real time update gives the user a more accurate indication when positioning for alignment with other object.

At present there are no facilities to move the intersecting points for line segments in poly and constrained poly line.

5. Re-Engineering

Chapter five involves research into the re-engineering stage of the software lifecycle. The programming language C++ and its specific features are examined in order to facilitate its interpretation. Storage structure specification is addressed and defined. The individual interpretation of class headers, methods and attributes are discussed in more detail.

To address the phase some definitions are introduced Coleman (1994):

- *Forward engineering* comprises of moving from high-level abstractions and logical designs to the physical implementation of a system.
- *Reverse engineering* changes the representation of the system to a higher level of abstraction. This combines the areas of re-documentation and design recovery.
- *Restructuring* involves the transference of information from one representation to another while remaining at the same level of abstraction. An example of restructuring would be to remove all 'goto' statements from a program.
- *Re-engineering* is the composition of reverse engineering and forward engineering or restructuring. A difference between restructuring and re-engineering is that the latter may encompass the modification of functionality or implementation techniques

When an organisation transfers to object oriented paradigm they will typically be faced with the problem of re-engineering. The existing code used within the organisation needs to be re-engineered into object oriented code if previous investments in software development are to be recouped (Jacobson *et al* 1994).

About the most well-known early work on re-engineering can be traced back to a Communications article by Bohm and Jacopini in 1966. Their work embodied the removal of 'goto' statements by parsing the code and placing it in a flowchart format, restructuring the flowchart and then converting the flowchart back into code. Two years later Dijkstra's famous Communications letter informed the computing community about the harmful effects of 'goto' statements. This lead to an increase in the number of organisations that embarked on re-engineering (Arnold 1994).

From those days to the present, the numbers of people re-engineering have gradually increased. The biggest factor that has changed over the years is the terminology used (Pressman 1992, Loernz and Kidd 1994). Even today the term reverse engineering is used to encompass all aspects of re-engineering and restructuring.

The question quite often is why do we need to re-engineer code. One of the reasons is that today nearly every programmer is reinventing something that has been built, debugged, and painfully improved - a thousand times before (van Vliet 1993, Sommerville 1992, Pressman 1992, Meyer 1994). Software re-engineering is often not considered because some developers see this as an admission of failure. If the systems did what they were intended to do then it would not be necessary to change it. Building a system from scratch does not have the same connotations (Yourdon 1993, Booch 1994, Coad and Nicola 1992).

There is, however, a flaw in this approach. It is not always possible to build a system that will be permanently correct. There is no way to predict what the system should be able to accomplish five or even ten years after the system was initially designed (Waters and Chikofsky 1994).

What is needed in CASE products is the ability to alter existing applications quickly in response to changing requirements in the marketplace and the new technology that is available (Sorenson 1988). This not only means applications that are developed for customers but also those used for the organisation itself.

Object oriented development is increasing the acceptance and value, or reuse, in the software engineering community (Coad and Nicola 1992, Lorenz 1993). Whether this is code that is written for reuse or re-engineering the object oriented approach provides continuing benefits for both domains.

There are two divisions that re-engineering can be split into. The first involves re-engineering code of a particular style, say object oriented, into the same styled final product. The other type, which is more difficult, involves changing the style of the code. An example of this would be taking 'C' code and re-engineering it into 'C++'.

5.1 Re-Engineering for the CASE Tool.

The focus of this thesis involves re-engineering from an object oriented language, C++, to the same language. The idea is to convert code found in multiple files into a Coad and Yourdon (1991b) style diagramming notation. From this point the code can be manipulated, deleted or added with the resulting code being saved back to the original files.

It was decided to keep the information in the same file format at this stage. By using the original format the amount of information that was needed to be understood and accessed was considerably less. In this technique the parser stores the file positions where useful information is retrieved from. When writing the information back, marked file positions are used for placing the information held within the tool. For example when the position for the definition of class 'X' is found, the CASE tool has to decide whether its information has changed. If so, the data held in the CASE tool will be written.

The written data may be in a different sequence, for private, protected and public areas the methods are written first then the attributes. Generally the overall order is kept because the attributes and methods are written out in the same order they are read in. The program goes as far as keeping definitions that are clustered on one line together. By reproducing the code in basically the same format as it was written the CASE tool provides a structure that will be familiar to the user, if they wrote the original code. Future extensions to this part of the CASE tool will provide facilities for restructuring the entire code into either user defined specification or to a standard template.

To facilitate an easy understanding of this part of the system it will be split into two distinct sections: Language Parsing and Diagrammatic Representation. The first section involves the interpretation of one or more C++ code files. The second section is a combination of two parts that involve producing diagrams. One area that is missing from a large number of existing CASE tools is the ability to generate diagram layouts. If application re-engineering is to be realised in the future then automatic diagram layout is of the utmost importance (Sorenson 1988).

Goals of the Re-Engineering Phase.

The re-engineering stage of the CASE tool was developed because of the large number of benefits it could offer. Analysing and designing the structure of software are not the only parts of a life cycle. Of equal importance is the implementation of code along the design specifications.

One of the goals was to provide support for the forementioned stage of the life cycle. It was inadequate to just provide the entry of code and the subsequent saving of this information to files. The CASE tool also had to be able to retrieve this code in whatever format it was presented. Another goal was to allow for the code to be in any language the user desired. The tool would retrieve information from a language specification file, similar to the template definition file, which would describe how to use the given language. This would allow the user to employ any methodology and generate the code for any language.

However this is an undertaking requiring far more work than could be done in this thesis. Instead it was decided to concentrate on a small subsection of the work. To keep in line with earlier work it was decided to focus on the re-engineering stage of this CASE tool towards the methodology from Coad and Yourdon (1991b) and the language C++.

There were two reason behind the choice of the Coad and Yourdon methodology. The CASE tool was initially developed to encapsulate all the features of object oriented notations, although it should support most notations in the future. All the trial systems were, however, worked around Coad and Yourdon's methodology. Coad and Yourdon was picked for a number of reasons:

- It was the most commonly used notation around the department for object oriented.
- It is a very popular notation, with a number of public domain and commercial CASE tools supporting it.
- It provides a variety of connection types that can be incorporated into hierarchical and non-hierarchical graphs. This will be discussed later.

To keep in line with previous development guidelines the same methodology notation was adhered to. If Booch (1991, 1994) is examined it could be considered a better notation to adopt since it directly supports C++. Some of the reasons for not using Booch's notation have been discussed previously in future enhancements.

The Coad and Yourdon notation is not built to be able to display all of the features of C++. It was still chosen for reasons layed out in the Diagrammatic Representation section. There were two options to compensate for this drawback. The first involved changing the notation, an action that would be very easy to implement with the CASE tool. The second option was to keep the standard look and feel of the Coad and Yourdon methodology but have the code editors display the additional information. Due to time constraints and the fact that the aim of this thesis was not about reworking methodologies the second option was taken. To support the methodology, editing features for classes, methods and attributes had to be very detailed in the information they provided.

More information about the interface designs for the various editors can be found in the Graphical User Interface section.

5.2 Language Parsing

5.2.1 The structure and composition of C++

The declaration of C++ class information can be in two different forms. The first is a forward declaration of the impeding full declaration of the class, several example of which are shown below. This is used in cases where circular references stop a full declaration being admissible.

```
class Expression;
class Viewable_Thing;
class Template;
```

This type of forward declarations are not used for any purpose in the CASE tool. They are mentioned here because at first glance they may look like a full class declaration. Instead the CASE tool has to be able to understand this so it can be ignored. The second form is the full definition of the class, its methods and attributes, or variables. The class header can be in varying degrees of complexities depending on the amount of inheritance. The list below demonstrates a variety of different header definitions.

```
class Expression {
....
}

class AnArc: public Segment {
....
}

class Groupitem: protected Item , public Array<TextItem *> {
....
}
```

The information involved is a class name followed by zero to many inherited base classes. This inheritance can be from any of the three different access specifier types: public, protected and private. All inheritance definitions are separated by commas and may be virtual.

The class header definition is terminated when the '{' symbol is parsed. This indicates that the methods and attributes are about to follow. All specifications of methods or attributes have to appear after the class header. They can not be declared or added to at a later stage.

If the first item after the class header is not a access specifier then the methods and attributes that follow are considered private. The access specifiers take on a slightly different form than in the class header. At this part of the class definition they are followed by a colon.

After an access specifier there can be as many definitions of methods and attributes as required. The access specifiers may also be repeated as often as the writer likes and there is no set order in which they must appear. An example of the use of access specifiers is shown below:

```
    ....
public:
    ....
private:
    ....
public:
    ....
protected:
    ....
public:
    ....
```

Method definitions are more complicated than attribute definitions. They have a standard format of:

- An optional field specifying the type of the method. Some of these are 'static' and 'operator'.
- Return type. This does not appear for class constructors and destructors.
- Normal method declarations do not have to specify the return type if is an integer.
- The name of the method.
- The parameters that the method expects, surrounded by an open and close parentheses. If this is empty then there are no parameters.
- Optional listing of code, which if present, will be surrounded by { } brackets.

Complexity is introduced when a variety of words are used to describe the parameters of the method and the return type. The return type and parameter types can be made up from macros which can have the characteristic appearance of a method.

A macro is a pre-processor directive of the form: #define <identifier> <token string>. The token string specification is stopped when a carriage return is reached. This can be made up from any number of characters and words. The macro is defined at one place in a file and referred to thereafter. A macro can also have parameters. At compile time the pre-processor performs a textual replacement on all macros removing them from the code.

Below are a list of some of the more difficult method declarations. To enhance readability the name of the method has been underlined. A macro can be identified in the second example, GCollection(ClassRef). It has the characteristic appearance of the method definition in the fifth example.

- static int *theClass(const char *NAME, Type T)
  { return bases; }
- const GCollection(ClassRef) operator &() const
- Type type(GCollection(ClassRef) *COMPONENT) const { return (Type)}
- const GCollection(ClassRef) &baseClasses() const
- ~Expression(void);

It is not compulsory to define any method code within the class definition instead it can be specified at a later period. A definition for the method 'theClass', shown above, in the class 'className' would look like:

```
int *className::theClass(const char *NAME, Type T){
    ....
}
```

The first example includes the code that defines the method, surrounded by '{}'. To add extra complication to an already complicated method, declaration extra parentheses may be added. These can surround the method name and parameters as well the entire declaration, up to the semicolon or opening parentheses of the method body. A few examples of this are listed below:

```
int ( type(int ID));
(char * name(void)){ ... }
(float (Evaluate(window_manager CONTEXT));
```

Method declarations are not limited to one per line. There can be as many as the writer wishes as long as they all have the same return type as either a pointer or not. It is also possible to declare attributes within this same definition, as long as they are the same type. An example is shown below.

```
int type(int ID), counter, *GetNumber(void), loop_count;
```

It is acceptable to have overloaded methods. This is quite often used and looked upon favourably in object oriented terms. Methods are disambiguated by the number and / or type of the parameters that the method will receive. Next is the specification of attributes. Attribute definitions are usually in the form of:

- The variable type.
- The name of the variable.

Like the method declaration there may be multiple definitions on one line in the form of the attribute type followed by one or more names, separated by commas. Also, as previously mentioned, method declarations may be interspersed along the definition line. Attribute declarations are defined in a similar style as methods. Many of the problems inherent with method declarations are also present for variables such as: the forementioned multiple definitions on a line, extra parentheses and macros in type definitions.

Listed below are some examples of variable declarations, with the variable name underlined for clarity. The first example gives a classic case of how a variable declarations can appear to be that of a method. No part of a variable declaration can be made outside of the class definition.

- const GCollection(ClassRef) bases;
- Type t;
- int (variable);
- (int (temp));
- (char *name);

The final piece of important information about definitions in C++ is that they are case sensitive. All keywords are in lower case. The compiler has to know the difference between 'class' and 'Class'. However not all compilers will allow the use of identifiers which have keywords partially capitalised.

5.2.2 Internal Representation

To facilitate the easy usage and manipulation of the C++ code it is first necessary to have an appropriate internal representation. Of paramount importance is the ability to easily identify or retrieve the appropriate information.

To accomplish this each class is contained within its own structure. The structure is the combination of four individual areas. The first is the class information. This contains the following elements:

- The name of the class.
- The actual line of code that represents the definition of the class name and its dependants.
- A comment about the class itself. To identify whether a comment is present or not, the area immediately before the class declaration is checked. This is considered the most likely place to find such a comment.
- Publicly inherited base classes.
- Base classes that are inherited as protected.
- Privately inherited base classes.
- For each of the access specifiers above, a list of which classes inherits from this class. These are identified as either public, private or protected.

This structure allows for easy searching either up or down an inheritance hierarchy. This can be very important for such processes as checking semantic correctness or placing the nodes of a hierarchy in close vicinity on a display. The other important characteristic is the identification of whether connections, up or down, are for what access specifier.

The next three areas of the containment structure are for the declaration of variables and methods. These are split into the three access specifier types: public, private and protected. Within each of these areas the following information is stored:

- An array of:
  - Variable names.
  - The actual line of code used to declare the variable.
  - The variable type.
  - A comment about the variable. As default the comment is perceived to be any comment directly before the definition.
- A counter indicating how many variables there are.
- An array of:
  - Method names.
  - The actual line of code used to declare the method header.
  - A list of all the parameters for the method.
  - The return type of the method.
  - The C++ code for the implementation of the method.
  - Room for two comments about the method. It is plausible to comment on the method both during the class definition and with the definition of the code for that method, if they are in separate places.
- A counter contains the number of methods for this access specifier.

It could be ascertained from the information presented so far that it would be easier to clump all the variables in one array, instead of one for each access specifier. Such lists would be beneficial early on in processing but it would cause problems later on. Checking access specifier types for method calls from inherited classes or rewriting the C++ back to the code files are two such problems areas. Clumping together of all methods and variables will occur when the information is changed into viewable things.

5.2.3 Class Parsing

5.2.3.1 Class Header interpretation

The information held within the class header is deemed very important to the soundness of the data presented in the CASE tool. The connections in the hierarchies and the names of each class are of critical importance. Any error in the information could have dire consequences for the production of the graphs. Due to these factors, the C++ grammar set out in Stroustrup (1991) was exactly followed.

Stroustrup's grammar was used to interpret the possible combinations of some of which are listed below. This grammar is shown in Appendix E. It has been converted to Backus Naur form to keep it in line with the rest of this thesis.

```
class Expression {
class AnArc: public Segment {
class Groupitem: protected Item , public Array<TextItem *> {
```

At this stage the definition of template classes is not accepted. A class template specifies how individual classes can be constructed much as a class specifies how individual objects can be constructed (Stroustrup 1991). This will be a future enhancement.

5.2.3.2 Method Interpretation

The method interpretation was handled differently than for the class headers. It was thought desirable to allow for a certain degree of flexibility when retrieving information. To facilitate this type of design the grammar in Stroustrup was not used.

The type of flexibility that is provided in this and the following sub-section could be a major factor in the useability of the tool. If method and attribute interpretation was in strict coherence with the C++ grammar then the CASE tool could not be used for debugging. For example Borland C++ compiler version 3.1. This allows the user to see an overall hierarchy of the classes. The main problem is that the code has to be compiled first. This is some what self defeatist since ninety nine percent of the time, the user requires this facility when the code will not compile.

For both the method and attribute interpretation one definition line at a time was retrieved and then parsed. To identify a method without the strict help of a grammar, the unique characteristics that define a method were identified. Three different approaches were used. The first involved looking through other peoples code. This provided easy access to examples of class declarations that other more write. The second approach involved looking through examples in Stroustrup. Since Stroustrup is the key figure in writing C++, his book was considered a valuable resource. He provides a comprehensive look at C++. Finally the grammar was examined, also from Stroustrup's book.

The pseudo code that explains how a class name is retrieved is as follows:

(1) Count the number of commas in the declaration line that are not embedded within '()' or '{}' parentheses.

(2) For every group separated by commas, or the whole definition if no commas are present, DO

(2.1) If the first character in a definition is an open parenthesis then remove this and the matching close parenthesis and continue. The close parenthesis is the last close parenthesis.

(2.2) If the first token before the last, non embedded, set of parentheses is one of:
C++ types like: void, const, int etc.
A name declared as a typedef or #define, used for macros.
A name of a class, struct or union.
A ')'.
then remove the opening parenthesis and the last close parenthesis.

(2.3) The last set of parentheses that are not within another set of parentheses contain the parameter list. The word or symbol that precedes the opening parentheses is the method name.

All symbols and words from the beginning of the definition line to either the first method or attribute name is the return type. Any '&' or '*' characters that are immediately before the method or attribute name are ignored.

Any '&' or '*' symbols that appear before the name of the method mentioned in the first part of (2.3) are concatenated to the end of the return type.

A set of examples are listed below along with which steps were applied to find the method names and the return type.

Definition line:
const GCollection(ClassRef) &baseClasses() const
Step (1) = 0.
Step (2) = One recursion for the whole line.
Step (3) = Not Applicable (N/A)
Step (4) = N/A
Step (5) = Method Name of 'baseClasses'.
Type of 'const GCollection(ClassRef) &'.

Definition Line:
(Type (type(GCollection(ClassRef) *COMPONENT) const));
Step (1) = 0.
Step (2) = One recursion for the whole line.
Step (3) = Line = Type (type(GCollection(ClassRef) *COMPONENT) const);
Step (4) = Line = Type type(GCollection(ClassRef) *COMPONENT) const;
Step (5) = Method Name of 'type'.
Type Name of 'Type'.

Definition Line:
char *name, GetCharacter(void);
Step (1) = 0.
Step (2) = Recursion number one for 'char *name'.
Step (3) = N/A.
Step (4) = N/A.
Step (5) = N/A.
Step (2) = Recursion number two for 'GetCharacter(void);'.
Step (3) = N/A.
Step (4) = N/A.
Step (5) = Method Name of 'GetCharacter'.
Type Name of 'char '.

The final line for the parsing of the string 'char *name, GetCharacter(void);' represents the special case set out in the last paragraph of step (2.3). Even though the line starts out 'char *' the method has the return type of 'char'. The '*' at the beginning of the sentence is for the variable 'name' only.

A method may be declared inline (inside the class) or outside the class definition. This may or may not be in the same file as the class definition. If the code is defined outside the class then the information pertaining to an individual method may be incomplete at the end of the class specification.

Not only does the parser have to keep looking for any further method declarations but the CASE tool has to handle having incomplete data. It is perfectly plausible for the developer to be working with half finished work.

5.2.3.3 Attribute Interpretation

Method interpretation is performed before attribute interpretation. When a method name is identified it is removed and replaced with a null symbol. If the attribute interpreter finds a null in a block it is checking, it skips this block. This allows the interpreter to only work with definitions that it knows are actual attribute specifications. Unlike for methods, attributes only require one rule to identify the name. The name is always the last word, or identifier, in the definition block.

For the identification of the attribute types a mechanism the same as for finding the return type for a method can be used. The only difference in implementation is that any open parentheses that do not belong to macro references are ignored.

6. Diagrammatic Representation.

To facilitate re-engineering of program code it is necessary to be able to graphically represent the information retrieved. This chapter discusses the different types of connections that are needed to implement the correct rendering of the re-engineered program code.

6.1 Introduction

Before investigating diagrammatic representation it is necessary to introduce some terms. The first new term to expand upon is crossings. This refers to the crossing of two or more connections. Figure 6.1 illustrates two examples of crossings and figure 6.2 a situation where there is not a crossing.

Figure 6.1 - An example of crossings.

Figure 6.2 - No crossings.

'Edge' or 'edges' are commonly used when referring to crossings, ie. edge crossings. An edge is a connection line made up of one or more line segments that travel between two points. In figures 6.1 and 6.2, the number of edges present are, from left to right: two, three and two. An edge is said to have stopped if it comes into contact with a node (black dot). A graph is a finite set of objects called nodes (also called vertices or points) interconnected by a (possibly empty) set of edges (Chartrand and Oellermann 1993). Figures 6.1, 6.2 and 6.3 are all examples of graphs.

The forth term to be introduced is 'planar'. For this section the term is used in the context of graphs, ie. planar graphs. To establish the definition of a planar graph it is first necessary to define a plane graph. A plane graph is a graph drawn in the plane (of the paper, computer screen, etc.) in such a way that any pair of edges meet only at their end vertices (if they meet at all, ie. no crossings). A planar graph is a graph that can be drawn as plane graph, ie. can have crossings (Clark and Holton 1991). Figure 6.3 demonstrates the differences between plane and planar graphs.

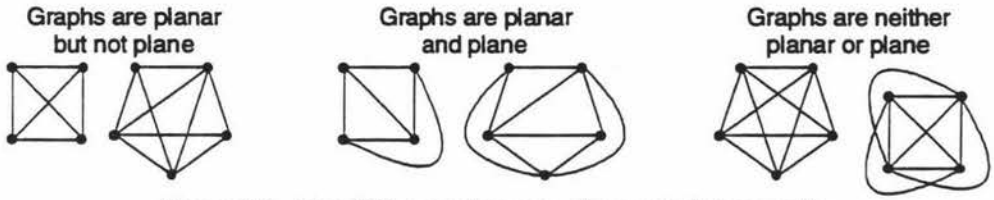


Figure 6.3 - The differences between plane and planar graphs.

When parsing C++ code and showing the resulting graphs, the presentation on the drawing canvas should be of a high quality. By high quality it is meant in a format that is more aesthetic. Increasing the aesthetics of a diagram offers the following benefits for the user:

- A more readable diagram.
- Easier identification of individual parts.
- Higher likelihood of the user distinguishing group and overall relationships for the various parts.
- Saving the user from rearranging the graph.

Readability can be attributed to many factors some of which can not be influenced. Of those factors that can be altered there are usually trade-offs due to either time constraints or the application of one factor having a negative affect on another. One example of the latter trade-off is increasing readability of a graph by spacing out the individual parts. This has a negative effect on the readability factor which minimises the size of the graph so more can be displayed on the screen.

Figure 6.4 demonstrates how increasing the aesthetics of a graph can greatly enhance the readability of the diagram.

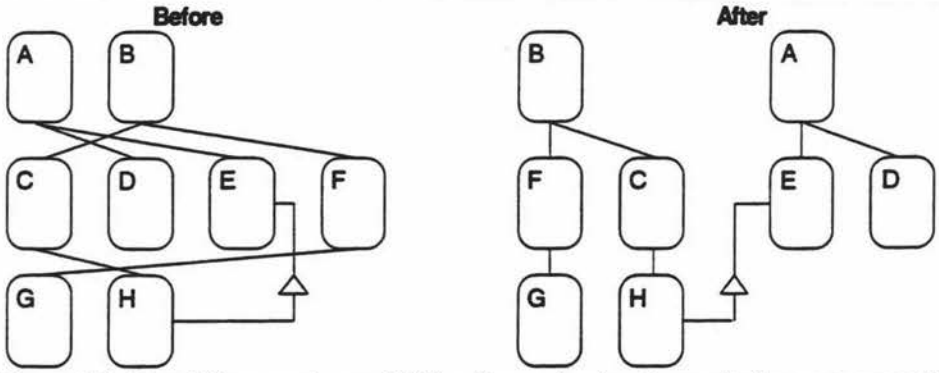


Figure 6.4 - The differences in readability of a graph when the aesthetics are increased.

Research Areas

There are a number of areas where research into the aesthetic placement of nodes is currently being developed. The two foremost leading areas are graph theory and integrated circuit layout, which includes Very Large Scale Integration (VLSI). After studying some of the literature for integrated circuit layout this avenue was considered inappropriate for the current circumstances.

VLSI layout algorithms use different aesthetics than those found in a graph editor. There is much more emphasis placed on finding a good layout than producing a layout quickly. VLSI is also more concerned with the efficient use of space aesthetic than readability (Paulisch 1993). In graph drawings associated with this section it is perfectly acceptable to have non-planar graphs. However in VLSI it is not possible to have any edge crossings at all. To accomplish this it is acceptable to use both sides of the circuit board (Lengauer 1990).

This leaves graph theory to solve the problems that will follow. Graph theory has applicability in both computer science and mathematics. This supplies a wealth of information that is presented from two different view points.

6.2 Diagrammatic Representation Types

When using a Coad and Yourdon's notation there are two different types of connections required. These form hierarchical and non-hierarchical graphs. The hierarchical graphs illustrate a top-down relationship between two or more nodes, Class-&-Object or Class symbols in this case. Figure 6.5 shows some examples of hierarchical graphs.

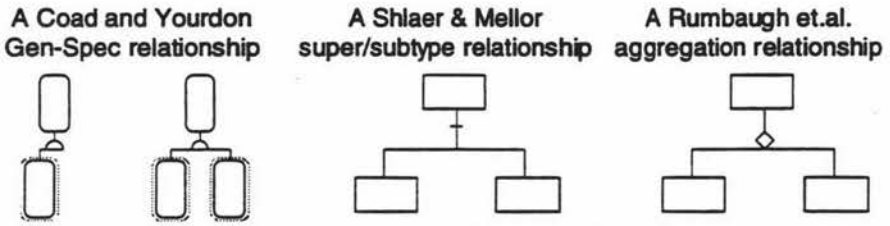


Figure 6.5 - Different types of Hierarchical relationships.

The initial requirements of this type of graph is to produce a result that has:

- A minimal number of edge crossings.
- Takes up a minimal amount of space from one side to another.
- Where possible keeps child nodes under their parent(s).

The first of the criteria requires the individual nodes to be sorted into an order that will place children of a common parent together, thus minimising edge crossings. The second criteria involves separating the individual sub-graph structures so the left hand side is a flush line. Currently all the sub-graphs are being looked at as a single entity (figure 6.6). Instead these have to be treated as individual parts of an overall picture.

Figure 6.6 - Sub-graphs treated as a single entity.

Problems occur when the sub-graphs are separated. When treated as a whole all nodes are kept flush to the left side. This can cause all but the first sub-graph to be concentrated over a larger area than necessary (figure 6.7).

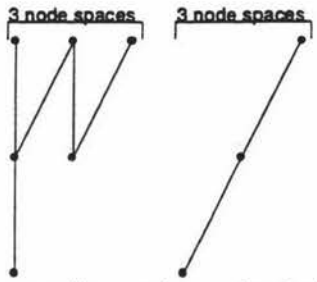


Figure 6.7 - The overall area taken up by the individual parts.

If the sub-graphs are split into individual components then each part can be made to cover less space. Figure 6.8 illustrates two sub-graphs drawn with a flush left hand side. Since these are treated as individual parts they both require minimal space.

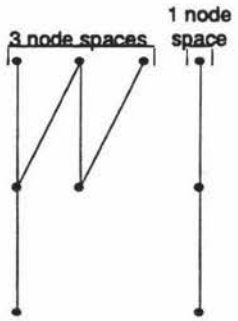


Figure 6.8 - The minimal space taken up by the individual parts.

The third and final criteria specifies the placement of children nodes under their parents. Placing them in this position emphases connectivity and relationships. Figure 6.9 gives an example where a child node is not directly under the parent. Figure 6.10 shows the appearance the graph should have.

Figure 6.9 - Unaltered graph.

Figure 6.10 - Child nodes placed under parents.

Any alterations applied through this criteria have a lower priority than those in the previous criteria. This prevents the nodes being pushed out from exceeding the maximum width of the unaltered graph. Figure 6.11 presents a three layered graph followed by what it can and can not be transformed into.

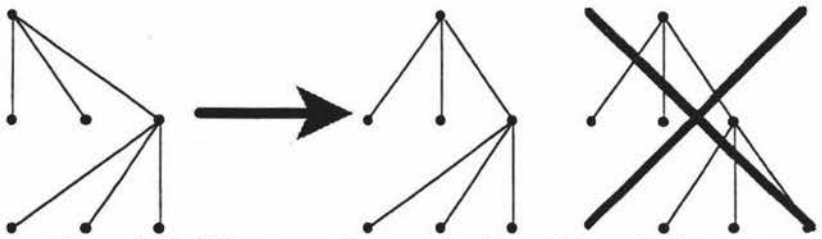


Figure 6.11 - What can and can not be done with graph alignment.

Although this may not seem logical at this stage it will become clearer in the non-hierarchical graph sub-section. In short, the wider the graph is the more room taken up in the final placement. Since there is more blank space at the top of a graph, several graphs placed together would produce large blank spaces.

Non-hierarchical graph connections are the second important type of connection used to facilitate the correct rendering of a multitude of different notations. This type is differentiated from the first by the lack of any hierarchical structure being produced by these connections. Figure 6.12 shows examples of some of this new type of connections along with the methodology.

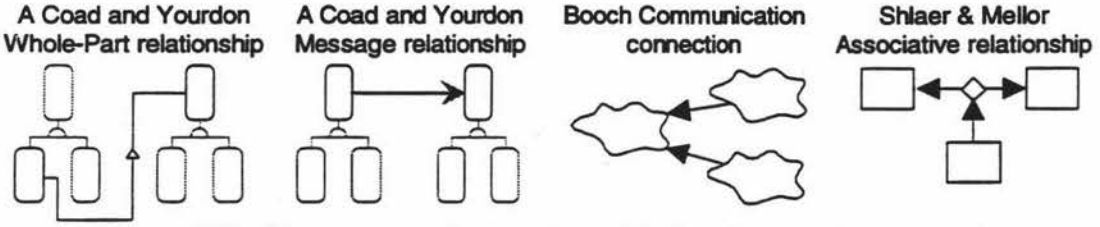


Figure 6.12 - Circumstances where non-hierarchical graph connections are used.

In this case they are used to connect either single objects or objects that are a part of a hierarchical graph. Thus two algorithms are required. The first calculates the relative positions of either the individual objects or groups of objects. The latter routes the connections from one object to another.

There was three criteria set out for placing non-hierarchical graphs. These criteria are:

- The connections should be as short as possible.
- They should have the minimum number of bends possible.
- The overall placement area should as compact as possible.

The first criteria, for shortest possible connection length, is achieved by sorting all nodes from a common parent. The sort places nodes at the second level of the graph close together. This facilitates a minimal distance between nodes and a minimum connection length. Figure 6.13 demonstrates the sort used.

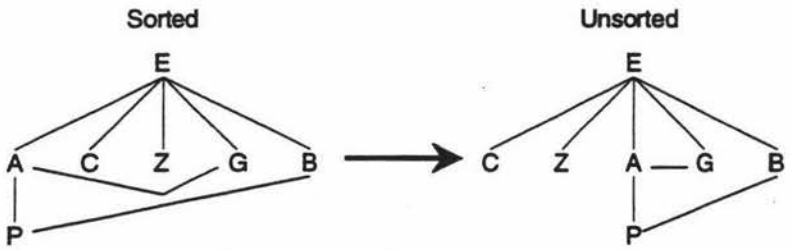


Figure 6.13 - The grouping of connected nodes after sorting.

The next criteria limits the number of bends in a connection. To accomplish this a path finding algorithm was developed that would always try the path that did not require a bend first. This coupled with the rule that existing hierarchies would not be crossed, unless it was the only way to make the connection, produces the required connection.

The final criteria specified that the overall graph be concentrated in a small area. This is an area of specification that can only be followed to a certain extent. Placing nodes so close that they touch would only add confusion to the diagram and lower its readability. Also if nodes are jigsawed together they can force unnecessary bends in connections. By jigsawing, it is meant placing the nodes of one hierarchy hard against the contours of another. Figure 6.14 illustrate an example of jigsawing and what the model should look like.

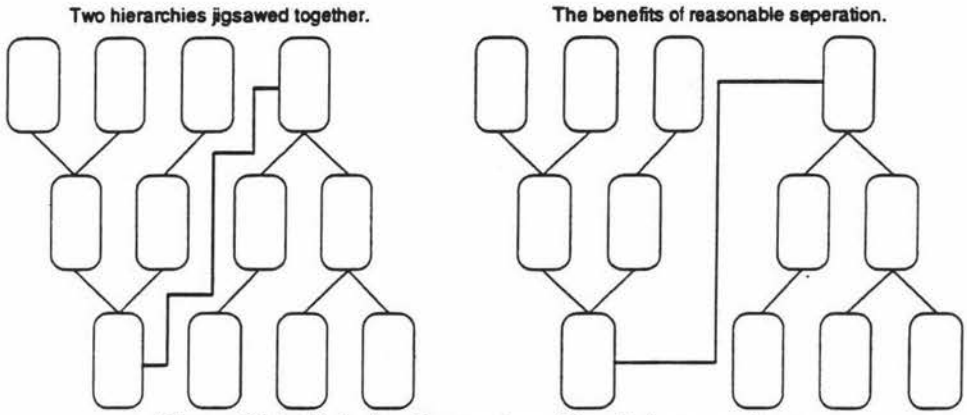


Figure 6.14 - Reducing the number of bends in connections.

If the nodes are placed at a sensible distance in a cluster they will take up minimal space and be more readable. Figure 6.15 shows two diagrams, the first of which is spread out. This requires a large amount of scrolling to view the entire graph.

The second diagram concentrates the graph in a smaller area. This reduces the amount of scrolling required by the user to see the entire diagram. It would also help in gaining an overall understanding of the diagram.

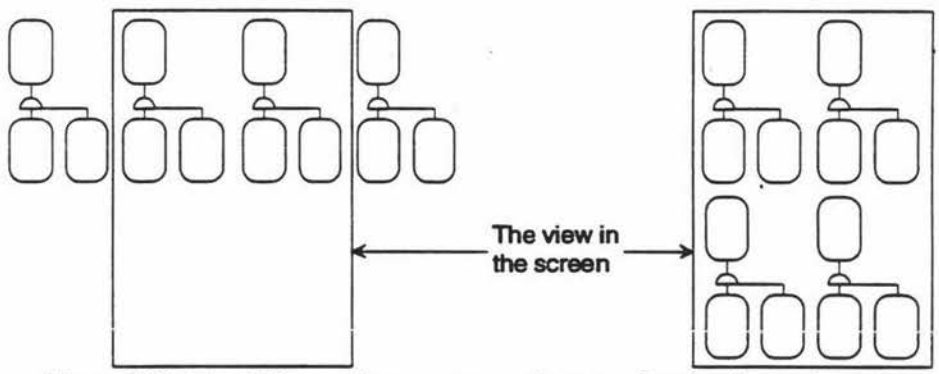


Figure 6.15 - The difference between a spread out graph and a clustered graph.

7. Hierarchical Graphs.

Carpano (1980) defines a simplistic two-level cycle-free graph algorithm. Given two levels of nodes interconnected in a random fashion (figure 7.1), this algorithm will tidy up the diagram. Even though it is not supposed to facilitate the best possible solution it does produce a result that can be considered reasonable.

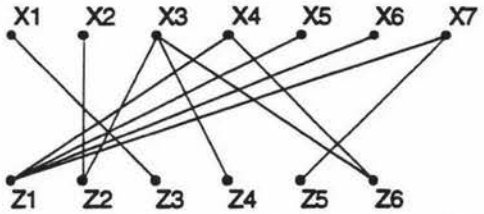


Figure 7.1 - An example of a two level graph before ordering is applied.

To determine ordering, the algorithm calculates the average position of nodes connected to the present one divided by the number of edges. In figure 7.1 this would mean the position of X3 is the average of the positions Z2, Z4 and Z6 divided by three, the number of edges.

Initially the positional information is calculated from the ordering of the nodes. X1 is at position 1, Z1 is at position 1 and X4 is at position 4. Once a nodes position has been altered the new information is used.

The following figures represent the Carpano algorithm being applied to the graph shown in figure 7.1. Figure 7.2 shows the calculations necessary to produce the new positional orderings.

Z'1 =	(X4 + X5 + X6 + X7) / 4	X'1 =	(Z3) / 1
Z'2 =	(X2 + X3) / 2	X'2 =	(Z2) / 1
Z'3 =	(X1) / 1	X'3 =	(Z2 + Z4 + Z6) / 3
Z'4 =	(X3) / 1	X'4 =	(Z1 + Z6) / 2
Z'5 =	(X7) / 1	X'5 =	(Z1) / 1
Z'6 =	(X3 + X4) / 2	X'6 =	(Z1) / 1
		X'7 =	(Z1 + Z5) / 2

Figure 7.2 - The necessary calculations for Carpano's ordering.

These can be expanded one extra step (figure 7.3), to show the inserted values for the new 'Z' positions.

Z'1 =	(4 + 5 + 6 + 7) / 4	Z'1 =	6.5
Z'2 =	(2 + 3) / 2	Z'2 =	2.5
Z'3 =	(1) / 1	Z'3 =	1
Z'4 =	(3) / 1	Z'4 =	3
Z'5 =	(7) / 1	Z'5 =	7
Z'6 =	(3 + 4) / 2	Z'6 =	3.5

Figure 7.3 - The new 'Z' positions expanded.

Similarly, the 'X' values are shown in figure 7.4. Note that replacement values for the 'Z' positions are the new values from figure 7.3 not those from figure 7.2.

X'1 =	(1) / 1	X'1 =	1
X'2 =	(2.5) / 1	X'2 =	2.5
X'3 =	(2.5 + 3 + 3.5) / 3	X'3 =	3
X'4 =	(6.5 + 3.5) / 2	X'4 =	4.5
X'5 =	(6.5) / 1	X'5 =	6.5
X'6 =	(6.5) / 1	X'6 =	6.5
X'7 =	(6.5 + 7) / 2	X'7 =	6.25

Figure 7.4 - The new 'X' positions expanded.

The results can be illustrated in two possible diagrams Figure 7.5 shows the new positional information for the 'Z' nodes used as coordinates upon the X axis. The nodes common to a given 'X' node are clumped together around the base region of its parent(s).

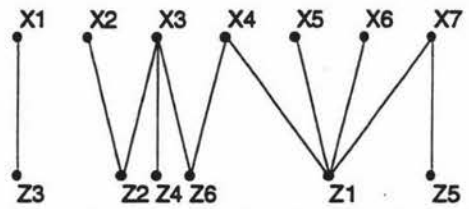


Figure 7.5 - The final result using clumping for the 'Z' nodes.

The alternative method for displaying the nodes is to place them in the set positions outlined in the original diagram, figure 7.1. The 'X' and 'Z' nodes determine their new ordering along these predefined positions.

Note that even though the 'X' nodes in figures 7.1 and 7.6 are identical to the original diagram, their positions were determined by the forementioned calculations. For this example the new 'X' ordering is identical to the original.

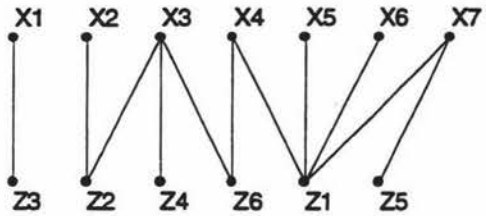


Figure 7.6 - The result without clumping.

However figure 7.6 does represent a graph that has a perfect answer, no edge crossings. Therefore another example is examined that, at the best possible outcome, will produce a result that has edge crossings. Figure 7.7 illustrates a graph with an imperfect ordering of nodes.

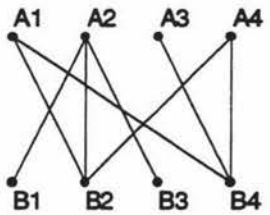


Figure 7.7 - A graph in a less than perfect ordering of nodes.

Figures 7.8 and 7.9 show the calculations involved in determining the new 'B' and 'A' orderings. This time the 'B' nodes produce a double up in two of the values, B1 and B3.

B1′ =	(A2) / 1	B1′ =	2
B2′ =	(A1 + A2 + A4) / 3	B2′ =	2.33
B3′ =	(A2) / 1	B3′ =	2
B4′ =	(A1 + A3 + A4) / 3	B4′ =	2.66

Figure 7.8 - The calculations for the 'B' nodes.

A1′ =	(B2 + B4) / 2	A1′ =	2.5
A2′ =	(B1 + B2 + B3) / 3	A2′ =	2.11
A3′ =	(B4) / 1	A3′ =	2.66
A4′ =	(B2 + B4) / 2	A4′ =	2.5

Figure 7.9 - The calculations for the 'A' nodes.

The resulting graph in clumped form (figure 7.10), presents an unexceptable outcome. Since two of the 'B' nodes have the same value both of their positions are at the same point. This indicates the unsuitability of using the clumping method for the displaying of graphs.

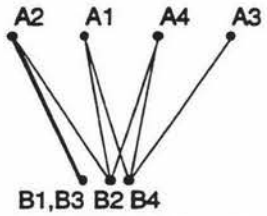


Figure 7.10 - The result of clumping the answer.

However the non-clumped version of the result does produce an exceptable outcome. As figure 7.11 shows, there is one edge crossing in the diagram. It is not possible through any alternative combinations to acquire a better drawn diagram than the one presented.

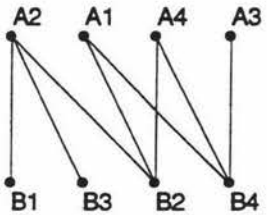


Figure 7.11 - The result in an unclumped format.

Carpano's algorithm produces results that while not always having the minimal number of crossings are close to this point. Through other examples that have been worked on, but not included, no one case has shown a result that has more edge crossings in the ordered graph than the unordered graph.

One graph that does not produce a perfect result after the application of Carpano's algorithm is shown in figure 7.12.

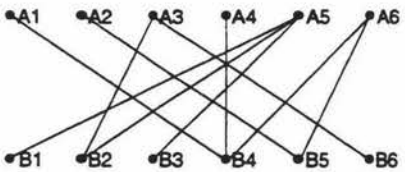
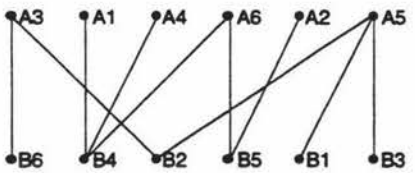


Figure 7.12 - An example of a graph that won't result in minimal edges after re-ordering.

As figure 7.13 illustrates there are five edge crossings in an example that could produce none. If A3 had been pushed to the end of the 'A' nodes and B2 then B6 pushed to the end of the 'B' nodes a zero edge crossing graph would have been produced.



B1 = 5	A1 = 3.66
B2 = 4	A2 = 4
B3 = 5	A3 = 3.5
B4 = 3.66	A4 = 3.66
B5 = 4	A5 = 4.66
B6 = 3	A6 = 3.83

Figure 7.13 - The resulting graph after Carpano's algorithm has been applied.

The initial conclusion upon revising the information presented in figures 7.12 and 7.13 indicate the anomaly may have been caused because two subgraphs were present. To test this theory an extra edge was added between nodes A1 and B6. Figure 7.14 shows the new graph, top, followed by the ordered graph.

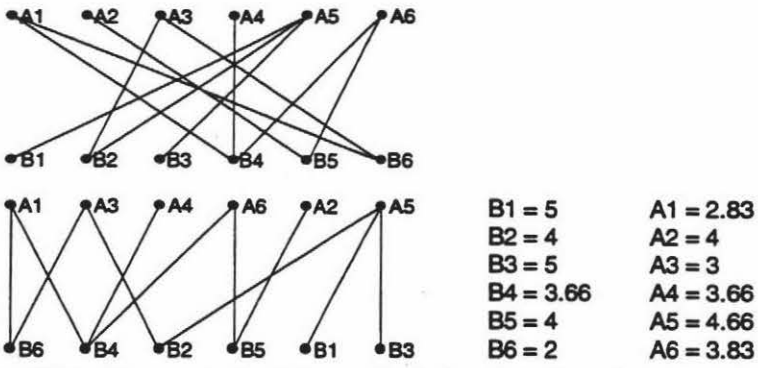


Figure 7.14 - Carpano's algorithm applied when the subgraphs are connected.

The number of edge crossings is down eighty percent from twenty five to five. This increases the readability but does not produce as good an effect as the minimal edge crossings would, with zero crossings. One method that would help multiple subgraph graphs are if the subgraphs were split into separate entities and processed individually. Figure 7.15 illustrate the application of this divide and conquer method.

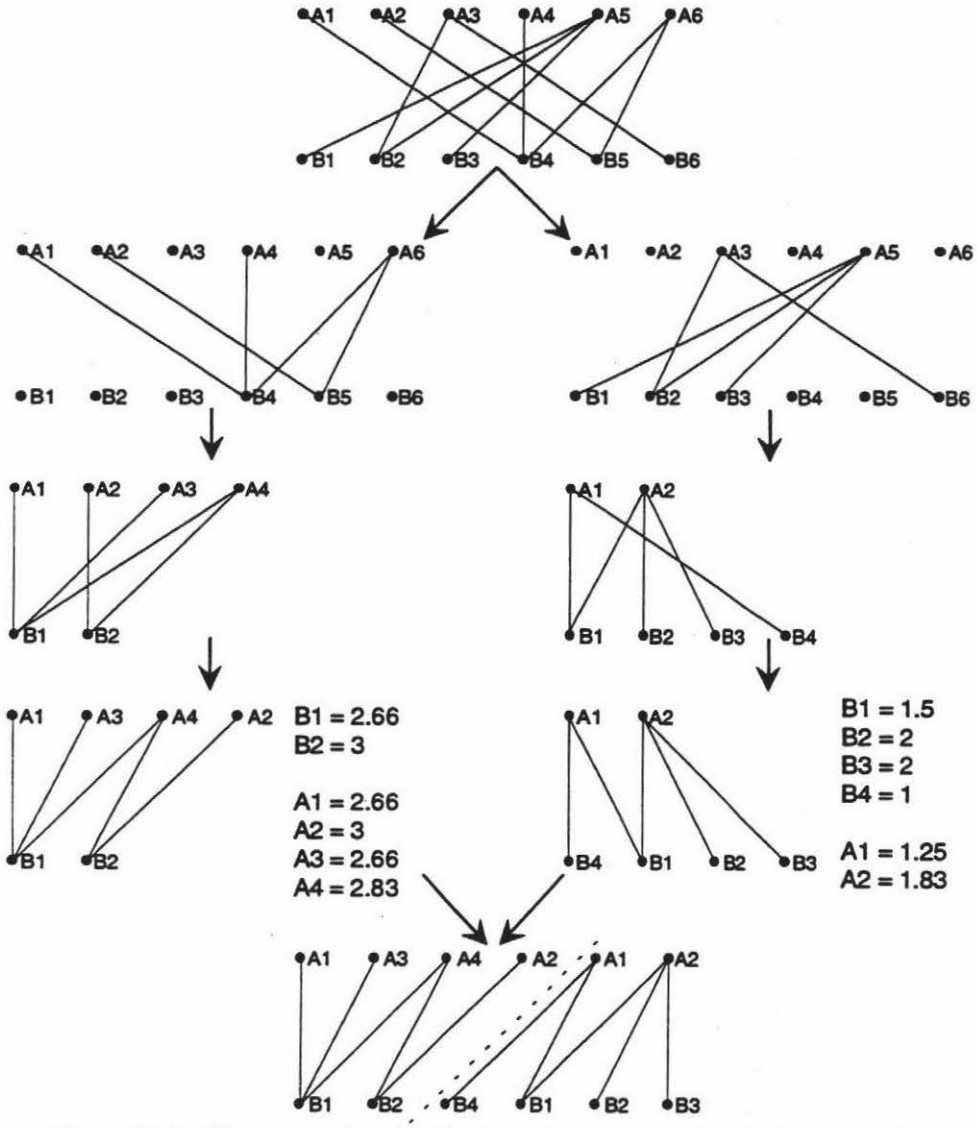


Figure 7.15 - The progression of a divide and conquer graph ordering approach.

The divide and conquer approach was adopted because of the added quality it can provide with very little extra processing. Since all hierarchies will be stored from the top node downwards it is a simple task of passing the structures sequentially to be processed.

7.1 Multi-layer Graphs

This algorithm in its present form is not applicable to the needs of the CASE tool. There may be tree structures present in the C++ code that only consist of two levels but trees of a much larger depth have to be catered for.

To accommodate multi-level graphs a hybrid approach to Carpano's algorithm has been developed. To test the hybrid approach the graph presented in figure 7.16 was used. Different variations and refinements were investigated to see if they are plausible alternatives.

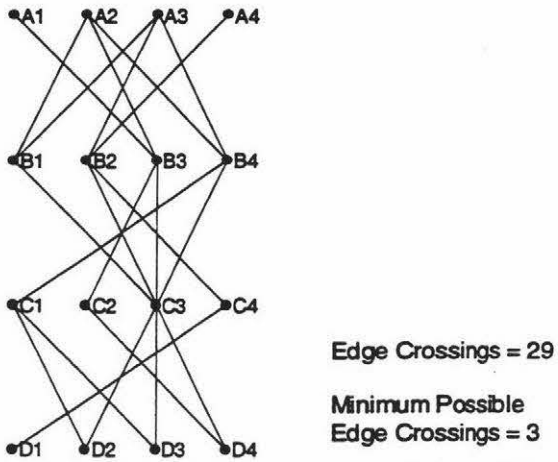


Figure 7.16 - The example graph to experiment with the multi-level algorithm.

Carpano's algorithm processes the second and then the first level. It was considered plausible that the same idea of using a bottom up approach could correctly sequence the nodes of a multi-level graph. To help illustrate how each hybrid algorithm is processing the node data, a diagram like the one in figure 7.17 will set out the stages of calculations.

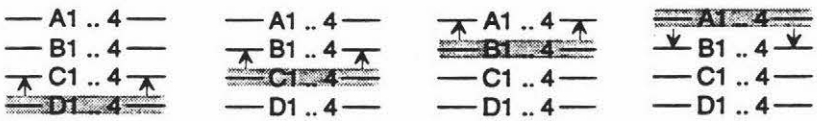


Figure 7.17 - The processing of each level of the graph.

Figure 7.17 is split into four sections with each section representing a level of processing. Each section is processed sequentially from left to right and then top to bottom, which is not required if there is only one level. In the first section "D1 .. 4" is shaded to represent that this line will be processed.

The arrows above the shaded section indicate that the edges being processed go from the 'D' nodes to the 'C' nodes. In the forth section, the nodes "A1 .. 4" are being processed in relation to their edge connections with any 'B' nodes. Figure 7.18 shows the same data as figure 7.17 but in a non-graphical format.

Sequence	Level being processed	Edges connect to level
1	D	C
2	C	B
3	B	A
4	A	B

Figure 7.18 - How each level of the graph is processed.

Figure 7.19 shows the results from the application of the forementioned hybrid algorithm. The conclusion from these results is that the algorithm is not suitable for re-ordering multi-level graphs, even though the number of edge crossings has dropped thirty one percent. Out of the three rows of edges only one, the top, displayed suitable improvements. This row produced a minimum number of crossings. The number of edge crossings in the middle row stayed constant while the last row increased by one.

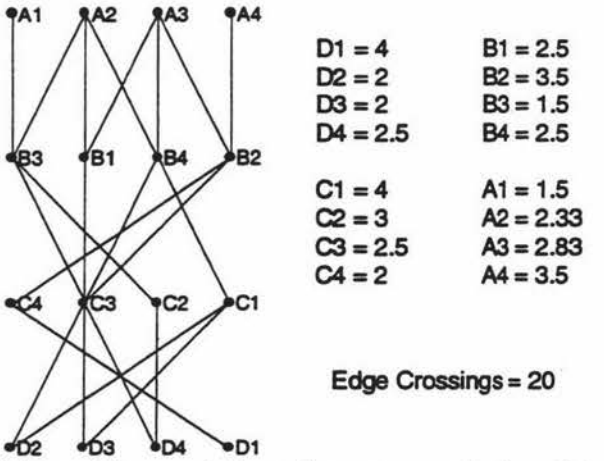


Figure 7.19 - The resulting graph from a bottom up application of the algorithm.

Since the previously mentioned hybrid algorithm actually failed to show any promise of good results, an alternative had to be found. Another alternative, figure 7.20, starts at the 'B' nodes and works downwards to the bottom and then calculates the top 'A' nodes. The results of this are identical to the previous example so they can also be ruled out.

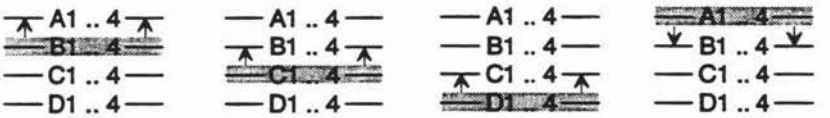


Figure 7.20 - An alternative sequence for applying the algorithm.

To keep more in tune with the original algorithm a new approach was suggested. Since Carpano's algorithm was applied firstly to the bottom row and then to the top row, a similar method is applied to the multi-level graph. Figure 7.21 - shows this new approach.

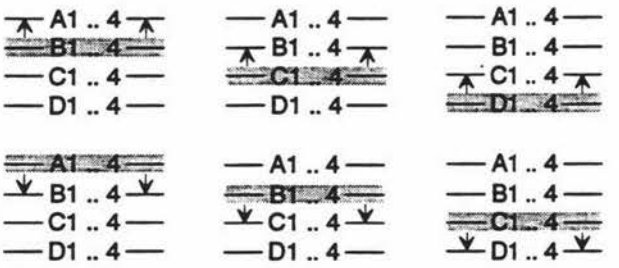
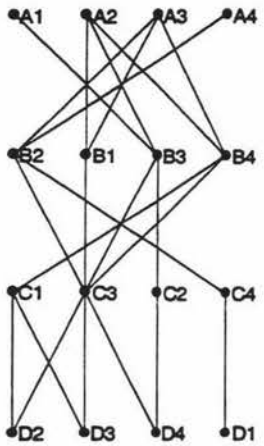


Figure 7.21 - The bottom down then top up approach.

The application of this method (figure 7.22), proves it is also an unsuitable alternative. The number of edge crossings is worse than the previously mentioned alternatives. For the first row of edges the number of crossings increased by one while the second row stays constant. The third row is the only one that displays any success, with the minimum number of crossings present.



B1 = 2.5 D1 = 4 B1 = 2.5
B2 = 3.5 D2 = 2 B2 = 2.25
B3 = 1.5 D3 = 2 B3 = 2.75
B4 = 2.5 D4 = 2.5 B4 = 3.25

C1 = 4 A1 = 1.5 C1 = 2
C2 = 3 A2 = 2.17 C2 = 2.5
C3 = 2.5 A3 = 2.83 C3 = 2.17
C4 = 2 A4 = 3.5 C4 = 4

Edge Crossings = 22

Figure 7.22 - The resulting graph from a bottom then top application of the algorithm.

From figures 7.22 and 7.19 it was easy to ascertain the most beneficial aspect of these alternatives. The last position, consisting of two levels, is always the best example of a planar graph. The new approach would need to consist of more than one top down cycle.

This was called the staircase approach because it recursively descends the graph one step below the previous top down cycle. The approach as shown in figure 7.23 is:

- Top down from the 'B' level nodes.
- Adjust the 'A' level nodes.
- Top down from the 'C' level nodes.
- Top down from the 'D' level nodes.

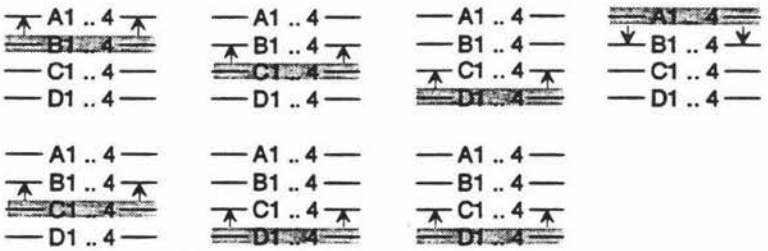
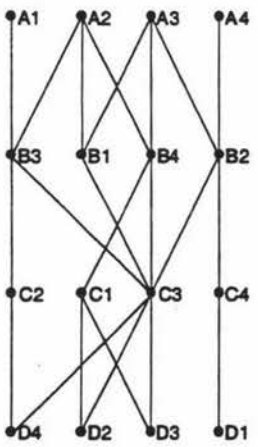


Figure 7.23 - The staircase approach.

The resulting graph, figure 7.24, is a vast improvement upon the original graph and the previously mentioned alternatives. The number of edge crossings is three above the minimum possible number.



B1 = 2.5 C1 = 2.5 D1 = 3.5
B2 = 3.5 C2 = 1.5 D2 = 2.5
B3 = 1.5 C3 = 2.5 D3 = 2.5
B4 = 2.5 C4 = 3.5 D4 = 2

C1 = 4 D1 = 3.5
C2 = 3 D2 = 2.5
C3 = 2.5 D3 = 2.5
C4 = 2 D4 = 2

D1 = 4
D2 = 2
D3 = 2
D4 = 2.5

Edge Crossings = 6

A1 = 1.5
A2 = 2.17
A3 = 2.83
A4 = 3.5

Figure 7.24 - The resulting graph from the staircase approach.

Even though the last step for the 'D' nodes is redundant it will not be removed at this stage. Instead one slight variation will be tested to see whether any improvement is possible. This variation involves resetting the values of the previously recursed top down cycle.

The values will be reset to their new priority positions and the next top down cycle will use these values. Figure 7.25 illustrates the changes that occurred. The node information in the rectangular boxes represents the previous columns nodes after their values have been reset to their new ordering.

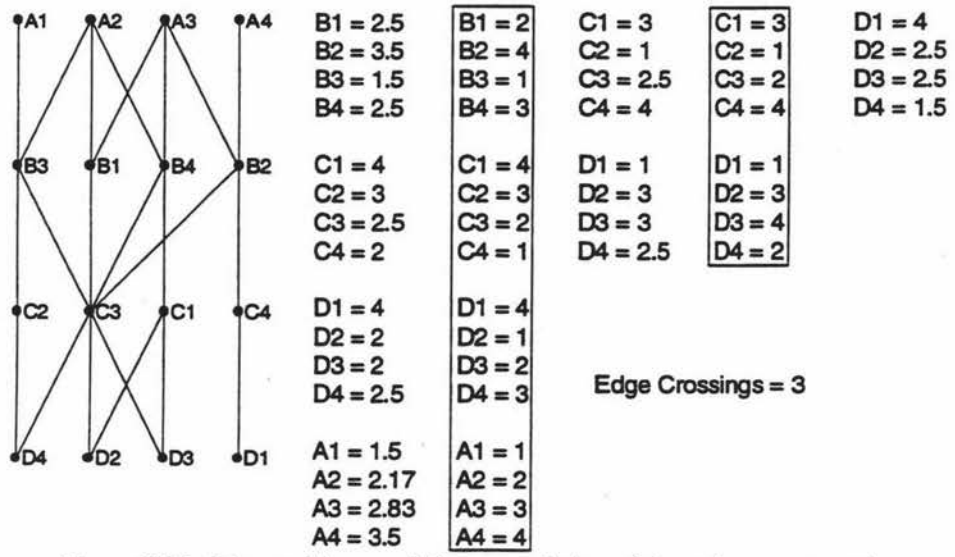


Figure 7.25 - The resulting graph from a variation of the staircase approach.

The application of this variation to the staircase approach shows that this method is capable of facilitating a perfect ordering of a graph at least some of the time. Since there are obvious redundancies in the forementioned approach, a new refined staircase approach was suggested (figure 7.26).

Figure 7.26 - The refined staircase approach.

The processing of the 'C' and 'D' nodes from the first cycle and the 'D' nodes from the second cycle have been removed. Likewise the nodes that need to be reset before proceeding to the next cycle can also be pruned. The subsequent graph and node values are shown in figure 7.27.

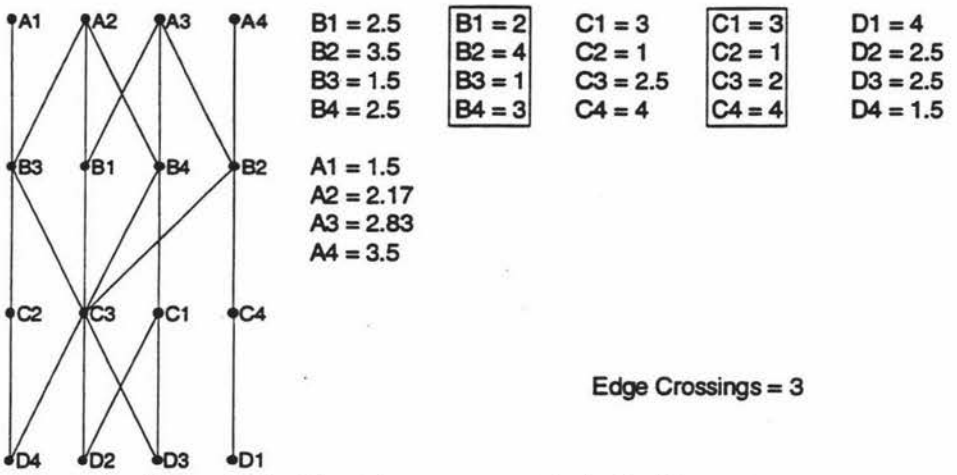


Figure 7.27 - The staircase approach in its final incarnation.

From all indications the staircase approach to the application of Carpano's algorithm is very efficient and produces a quality result. This approach has been tested with many other examples including the graph in figure 7.28.

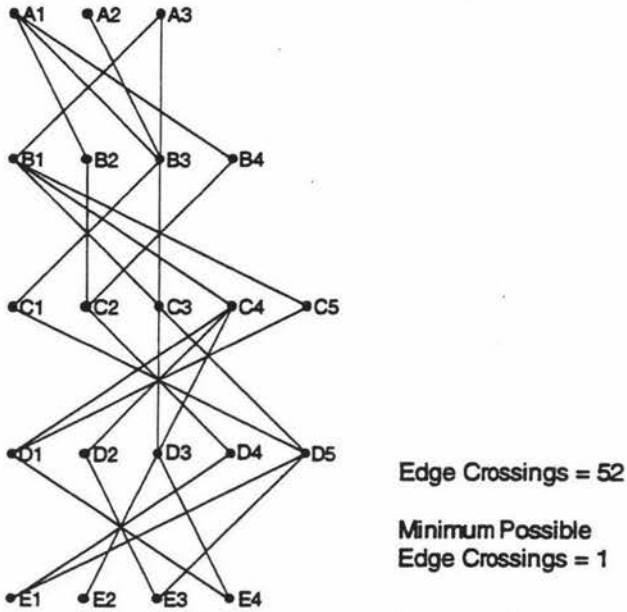


Figure 7.28 - The new example test graph.

The reason for this example's inclusion is because it takes an extremely cris-crossed graph and produces an answer that was unexpected. To develop these examples, a clean graph that contains the minimum possible edge crossings, is transformed into a graph with a large number of crossings. The staircase approach is then applied to the graph and in the best circumstances the original graph is produced. In this case, however, the approach taken managed to get rid of a crossing between the 'C' and 'D' level nodes. The result, figure 7.29, ends up paying the penalty for this by having to include two crossings at the next level, which were not in original model.

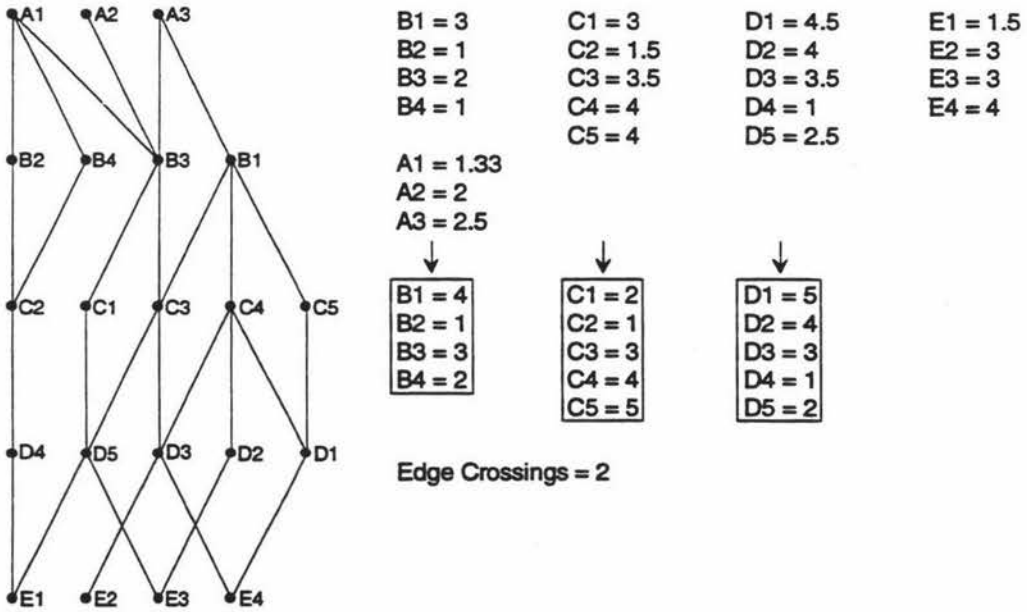


Figure 7.29 - The application of the staircase approach.

The testing of the staircase approach to the application of Carpano's algorithm has, in all cases examined, proved complementary. The number of edge crossings were closer to the minimum possible crossings than that of the unordered graph they originated from. It was this reason that the approach was adopted for hierarchical drawing in the re-engineering section of the CASE tool.

7.2 Connections that descend more than one level.

A problem that has not been addressed by the staircase approach is for connections that descend more than one level without connecting any of the levels in between. In all the previous examples the connections transcended only one level. Figure 7.30 shows an example of a connection that descends more than one level.

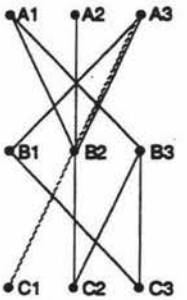


Figure 7.30 - A multilevel connection graph.

The easiest method to find out whether such nodes are a problem is to apply the staircase approach to the graph and look at the result. Figure 7.31 illustrates what happens to the graph.

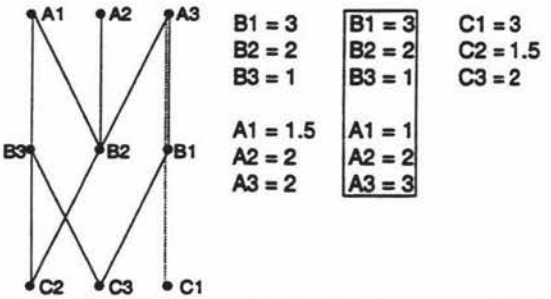


Figure 7.31 - The graph after the standard staircase approach has been applied.

The resulting graph, like the original, has one connection overlayed across another. This sort of answer is unacceptable. It would cause confusion about which connection was connected to what node. Instead of connections from 'A3' to 'B1' and 'A3' to 'C1' it would appear to be connections from 'A3' to 'B1' and 'B1' to 'C1'.

The obvious approach to rectify this mis-representation is to add dummy nodes placed between the levels. The multi-level connections would migrate through these points to the lower levels. The dummy nodes would never appear on the final graph but the connection would still travel through this point. Figure 7.32 shows the staircase approach applied to the dummy node modified graph.

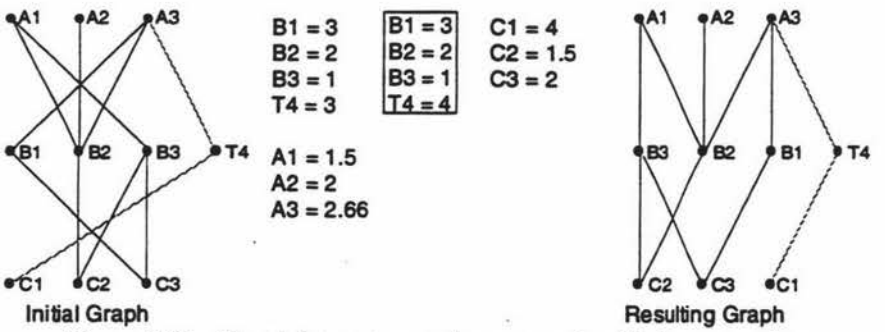


Figure 7.32 - The staircase approach on a graph with dummy nodes.

Since the dummy node approach satisfactorily represents a hierarchical graph it was added to enhance the staircase approach.

7.3 Tidying Up

The final alterations to a hierarchical graph is the padding out certain nodes to increase the aesthetics of a graph. Figure 7.33 shows a before and after shot of a graph where padding greatly increases the readability.

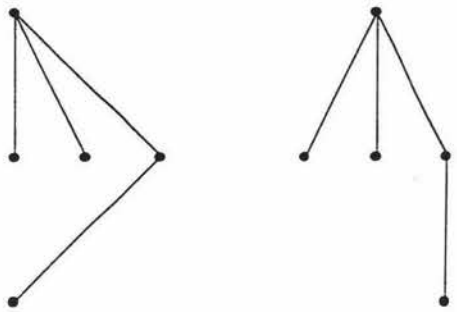


Figure 7.33 - The effects of padding.

To facilitate the tidying up operation the following set of steps were applied. The graph in the before shot of figure 7.34 will be used to demonstrate the outcome of each step or group of steps.

Step one sets up a loop that will iterate through the graph level by level. The iteration is a combination of a top down and a bottom up traversals. To understand why both approaches were used, they have to be looked at singularly. Figure 7.34 illustrates how the top down and bottom up approaches produce differing results.

Step 1: *For each level of nodes (going from second to bottom upwards then the second to top downwards) do*

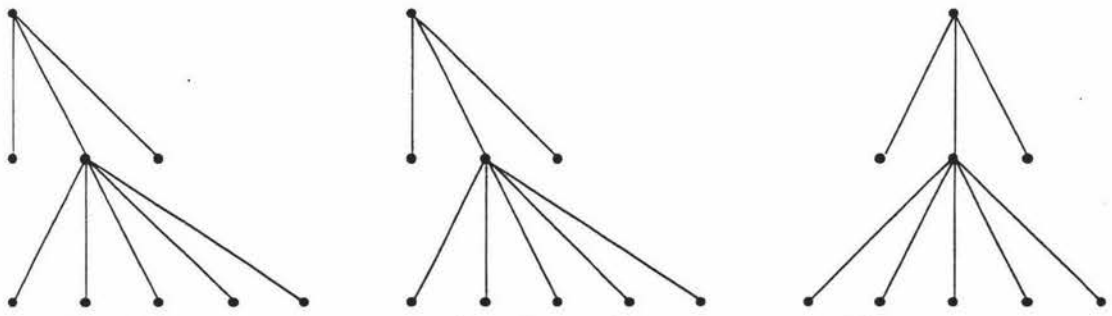


Figure 7.34 - Top down and bottom up padding.

The Christmas tree shaped original graph could either be in its present form or flipped upside down. Therefore no single method could be guaranteed to tidy up the graph correctly. It is for this reason that both the top down and bottom up approaches have been used. There is no significance in their ordering. The reason for the first level being left out for each iteration is based on the linking of connections. In the top down approach, nodes at the current level are moved depending on their relationship with connections to the level above. Since there is no level above the top level it can be ignored. The same logic also applies for the bottom up part of the iteration.

The next step calculates the number of spaces that can be added for padding. The number is calculated by subtracting the width, in nodes, of this level from the maximum width of the graph. The resulting number specifies how many dummy nodes can be used for padding.

Step 2: *padding = the maximum number of nodes at any levels minus the number of nodes in this level.*

Step three sets the direction that the connections will be processed. If the 'for' loop in step one is going from top to bottom, then *connectedTo* point to the level above otherwise it points to the level below. Step four sets variable information so that the 'while' loop in step five will be able to step through each node at this level.

Step 3: *If the direction for step 1 is going down then*
connectedTo = the level above this level
else
connectedTo = the level below this level
Step 4: *NumOfNodesAtThisLevel = the number of nodes at this level*
CurrentNode = 1

Step five loops through the proceeding steps while there is still padding space left and while there are still nodes to process.

Step 5: *While padding is greater than zero and*
Current is less than equal to NumOfNodesAtThisLevel do

Steps six through eight facilitate the actual padding. There are two checks that have to be done before a dummy node can be added for padding. These identify whether there are connections from the appropriate level and whether padding can correct any centering problems.

Step 6: *If node Current has one or more connections to level connectedTo then*
Step 7: *If the centre most connection is to the left of Current then*
Step 8: *insert padding node at this position.*
Step 9: *Increment Current by one*

If the two checks are both true then a padding space is added. Regardless of the outcome, the pointer to the current node being worked upon is incremented. This now points to the next position that a node can appear at.

If step eight had just inserted a node then the new pointer position will once again point to this node. If step eight was never executed then the pointer will either be at the next node or at the end of the node list.

8. Routing Non-Hierarchical Graph Connections.

8.1 Positioning Hierarchies.

When first researching positioning algorithms for non-hierarchical graphs a perfect solution was sought. It was later ascertained from Johnson (1994) and Griffin (1994) that the problem was defined as NP-Hard. To define NP-Hard it is first necessary to specify NP-Complete. An algorithm that is NP-Complete can be solved for small examples. As the example becomes more difficult its complexity increases exponentially. At a certain stage the problem is no longer able to be solved. From this stage optimisation algorithms are used. These can not be guaranteed to produce a perfect answer, depending on the algorithm and the difficulty of the problem, just a good answer.

An example of an NP-Complete problem is the travelling salesman puzzle (Weiss 1992). The problem has the following definition:

- There is number of nodes or cities.
- Each city is connected to at least two other cities but not necessarily every other city.
- The salesperson has to visit every city only once, travelling the shortest possible distance in the overall trip.

In summary, optimisation algorithms are usually used on NP-Complete problems because they can be used on all cases. NP-Hard is defined as an algorithm that is at least as hard as any NP-Complete algorithm (Hudson 1994).

The first possibility that was considered involved the modification of the approach outlined in the previous sub-section. Instead of just applying the staircase approach from top to bottom it would also be applied from left to right. Even at the start there were uncertainties about this approach. Each point had to be moved into either a square or rectangular shape. This could pose problems in particular situations where these shapes are not the best for displaying the current graph. Figure 8.1 illustrates a 3x3 graph and one possible solution that will be used for the following examples.

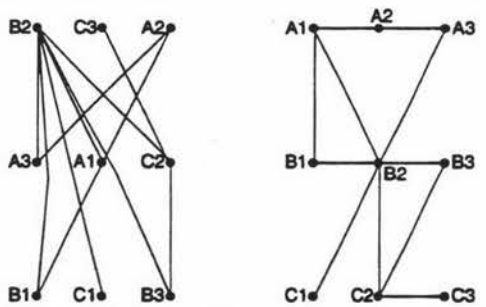


Figure 8.1 - The unordered graph example and one possible solution.

Figure 8.2 illustrates the transformation of the graph to encompass the connections transcending multiple levels.

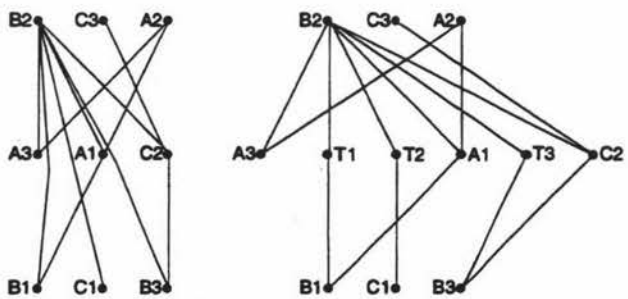


Figure 8.2 - Introduction of temporary points.

Figure 8.3 shows the resulting graph after the staircase algorithm is applied top down. Note that the positions of the temporary variable are still adhered to even though their distance apart has been reduced.

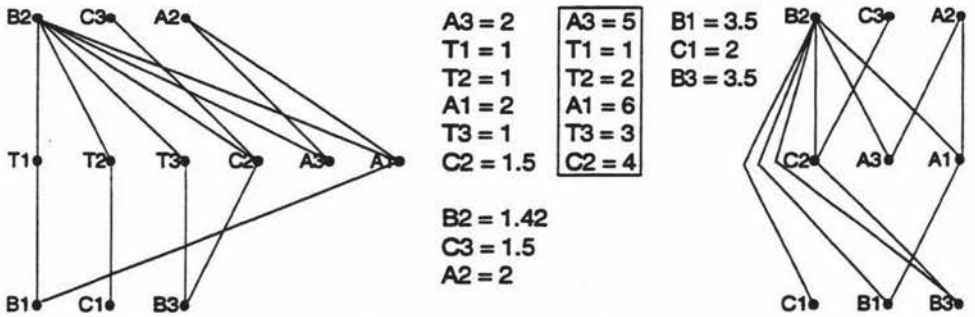


Figure 8.3 - The model after the top down staircase approach is used.

The resulting graph now contains only half the number of crossings than the original graph had. One area that is of concern is the point 'B2'. Theoretically this point should be in the centre of its group. Being left in the corner will mean there is a longer distance to get to the connecting point, increasing the chance of crossings.

Figure 8.4 shows the model before and after transformation. To simplify the examples, the model has been rotated ninety degrees clockwise. Due to the rotation, the next application of the staircase approach will be top down on the new model, which is equivalent to left to right on the previous model.

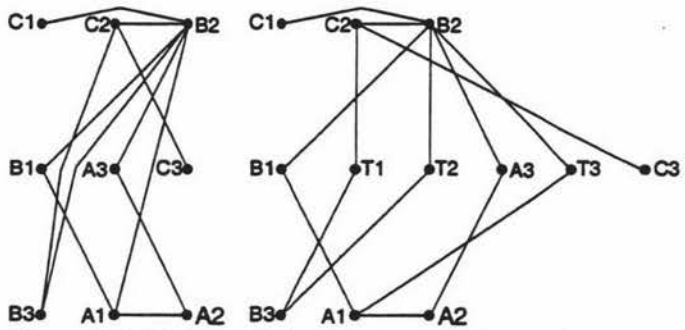


Figure 8.4 - The model after rotation and the transformation.

The result of the final stage of processing is shown in figure 8.5. Even though the number of crossings has been reduced to twenty percent of the original figure, the diagram is still not suitable. The final diagram is too spread out and some connections cover unnecessarily long distances.

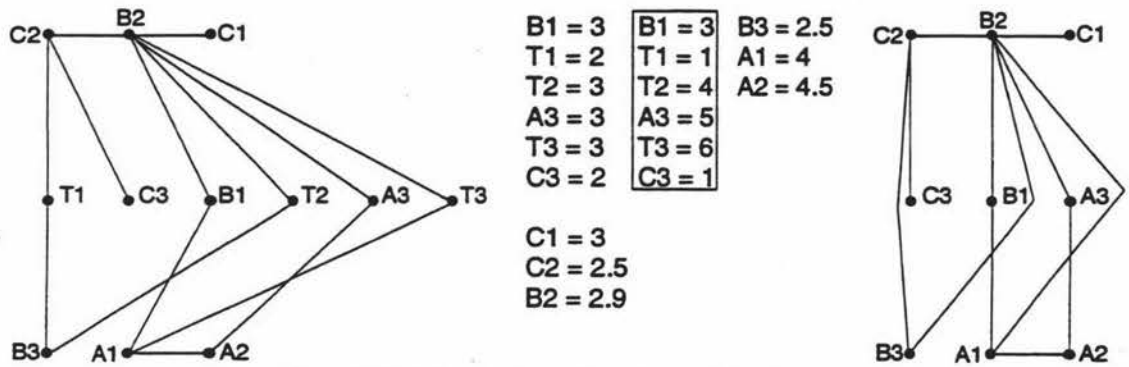


Figure 8.5 - The final transformation of the diagram.

The second and final idea involved the application of a group of heuristic's to the model. These heuristic's along with other processing information will be shown in the following text as steps. A complete set of these will appear in Appendix D. The example graph used in the previous attempt, figure 8.1, will also be used in this approach.

Determining the Start Point.
The first step finds the largest group of points that can be positioned as a start to the overall placement process. Placement had to start somewhere and it was considered that the point with the largest number of connections could cause the biggest problems if not positioned first.

Step 1: *Find the point with the largest number of connections.*

Placing the Start Point in a Grid for Working.
Steps two and three set up the area in which the graph will be assembled. The size is not restricted to that shown below, if more space is needed, it can be allocated later. The grids current size allocation should be satisfactory to assemble most examples. Figure 8.6 shows the grid after steps two and three have been applied.

Step 2: *Prepare a grided square area full of empty cells that has a length and height of*
((square root of the number of points rounded down to an integer) + 4)
For the example being used the working would be
= ((square root of 9 rounded down to an integer) + 4)
= ((3 rounded down to an integer) + 4)
= ((3) + 4)
= 7

Step 3: *Place the cell with the largest number of connections into the middle cell.*

Empty	Empty	Empty	Empty	Empty	Empty	Empty
Empty	Empty	Empty	Empty	Empty	Empty	Empty
Empty	Empty	Empty	Empty	Empty	Empty	Empty
Empty	Empty	Empty	B2	Empty	Empty	Empty
Empty	Empty	Empty	Empty	Empty	Empty	Empty
Empty	Empty	Empty	Empty	Empty	Empty	Empty
Empty	Empty	Empty	Empty	Empty	Empty	Empty

Figure 8.6 - The grided cell area.

To be able to place all points connected to the centre, an area has to be calculated that will accommodate them. The entire area is not used instead just the border region. Exploiting the inner region would, in some cases, cause conflicts in the connections.

Determine Position to Place the Surrounding Points.
Step four specifies the equation needed to calculate the area required for placement. Figure 8.7 illustrates the number of points that would have to be present to encompass connections to the number of points shown. The shaded area shows which squares would actually be used.

Step 4: *distance = ((Number of connections to the centre point / 4) rounded up) +1*
If distance is less than three then it is given the value of three

Examples.
14 points = ((14 / 4) rounded up) + 1 = (3.5 rounded up) +1 = 4 + 1 = 5
3 points = ((3 / 4) rounded up) + 1 = (0.75 rounded up) + 1 = 1 + 1 = 2 ⇒ 3

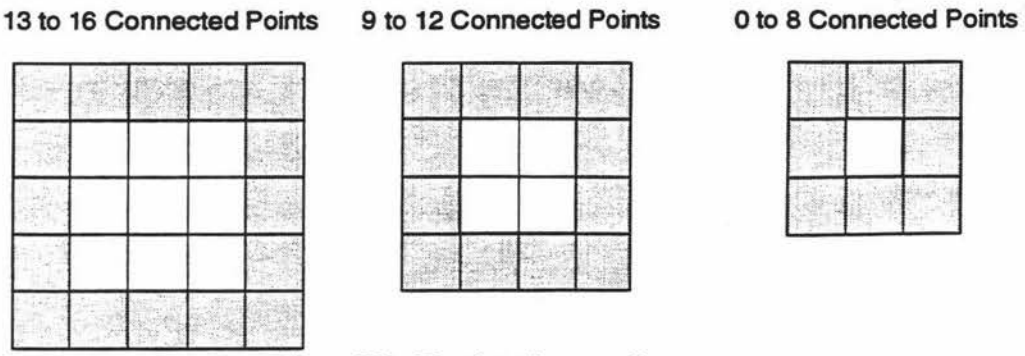


Figure 8.7 - The size of surrounding areas.

Ordering Surrounding Points for Placement.

Step five sorts the points, that are connected directly to 'B2', the centre point. The points, upon being sorted, are placed into an array whose length can be ascertained as the distance around the outside of the square calculated in step four. The exact length is equal to distance * 2 + 2 * (distance - 2).

Step 5: *Sort and position the points connecting to 'B2' in accordance with the seven following criteria, which are illustrated in figures 8.8 and 8.9.*

(1) Find new position values for the connections. These new values are calculated by the averaged positions of any connections to other points at the same level. The lists marked with a (1) in figure 8.8 show the working and results for this criteria.
It should be noted that if a point refers to a previously calculated point then this points new value is used. For example the point A1 would use the new information from point A3 and B1.
(2) Sort each group, points with the same new value, by the number of connections to points at the same level. The resulting list is placed in descending order.
(3) Sort the groups, keeping all points in a group together, according to the number of points in the group.

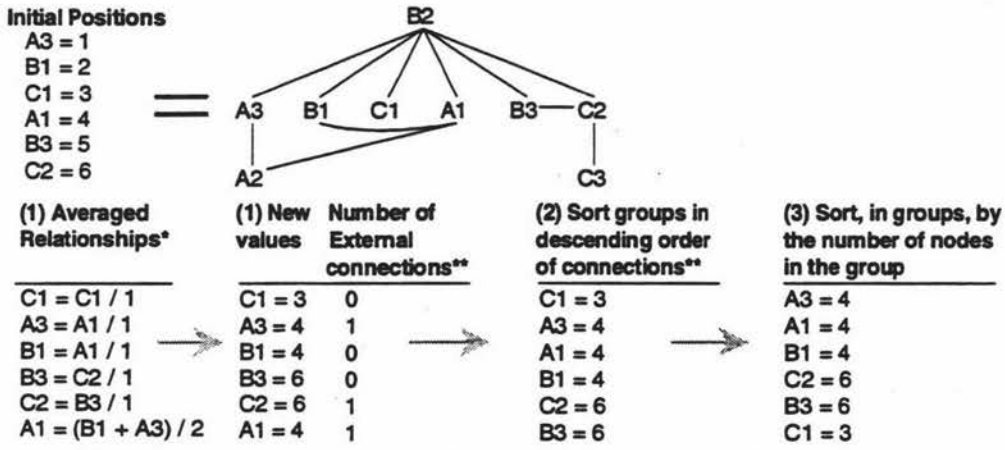
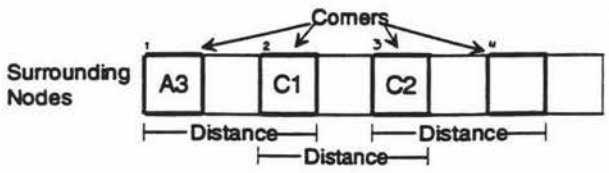
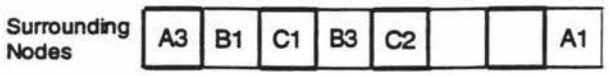


Figure 8.8 - The application of criteria 1 to 4 of step 5.

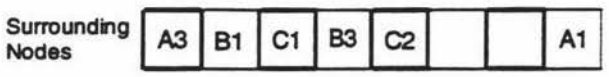
*(4) Create an array of ((distance * 2) + ((distance - 2) * 2)) length.*
(5) Place the start points of up to the first eight groups into firstly the corner positions and then the centre cells between the corners. Use the corners in this order: 1, 3, 2, 4 (figure 8.9).
(6) Place the rest of the points in the group around the already placed point.
(7) Place any remaining points in the left over empty positions, from left to right.



(5) Using the length 'distance' place the first node of each group in the corner positions then the centre squares between corners. Use corners in this order 1,3,2 then 4



(6) Surround the placed nodes with nodes of the same group.



(7) Place the remaining nodes into the free squares going from left to right

Figure 8.9 - The application of criteria 5 to 7 of step 5.

Place the Surrounding Points.

Steps six and seven position the points connecting to 'B2' and mark the unused cells as free. The unused cells are marked as free so that they will have a higher precedence than the empty cells. The higher precedence makes sure they will be used before the empty cells, keeping the points clustered around the centre.

Step 6: *Place all the points onto grid, starting at position (figure 8.10)*

x = middle - ((distance -1) / 2) rounded down.

y = x.

Working for the given example is:

x = 4 - (((3 - 1) / 2) rounded down) = 4 - (1 rounded down) = 3

y = 3

This will place the cells around the outside of a square with sides of distance length. All point positions, used and not used, are placed into the pointPos list.

Step 7: *Mark all cells not used FREE.*

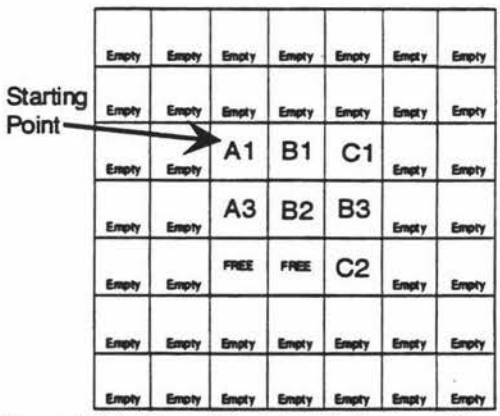


Figure 8.10 - The grid with added connections.

Prepare for Next Pass.

Step eight places all the points connected to 'B2', the centre point, into a list for general processing. Steps nine and ten delete the centre point and its connections from the graph. This stops the centre point from being used again or any of its connections being used to find relationships during sorting. The resulting graph is shown in figure 8.11.

Step 8: *Place all the points connected to the centre point into the nextPoints list.*

Step 9: *Delete all connections between the centre point, 'B2', and any other point.*

Step 10: *Delete the centre point, 'B2'*

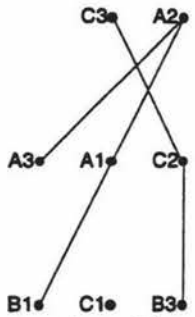


Figure 8.11 - The centre point and its connections removed.

Remove Connections to the Same Level.
Step eleven removes all connections between points in the nextPoints list. This refers to the connections between 'B1' and 'A1' and between 'B3' and 'C2'. These are no longer needed because the points connecting them have already been placed. Figure 8.12 shows the resulting graph.

Step 11: *For all points positions in the nextPoints list do*
if the current point connects to any other cell in the nextPoints list then
delete the connection.

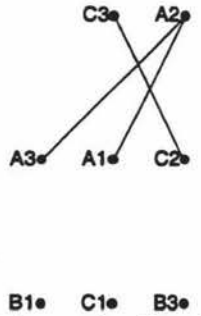


Figure 8.12 - The graph with points removed.

Free Grid Cells not Needed.
Step twelve sets all new placement positions that will not be used as free. Figure 8.13 illustrates which positions would be marked free for each given point.

Step 12: *For all cells in nextPoints do*
if this cell is free or the point has no connections then
Mark the cell(s) directly out from this position as free, figure 8.13.

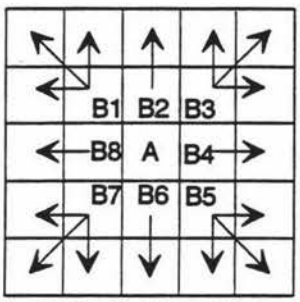


Figure 8.13 - Cells directly out from other cells.

Like previously mentioned, when cells are marked free they can be used as storage areas by other points. Figure 8.14 illustrates the changes in the grid.

Empty	Empty	Empty	Empty	Empty	Empty	Empty
Empty	Empty	Empty	FREE	FREE	FREE	Empty
Empty	Empty	A1	B1	C1	FREE	Empty
Empty	Empty	A3	B2	B3	FREE	Empty
Empty	FREE	FREE	FREE	C2	Empty	Empty
Empty	FREE	FREE	FREE	Empty	Empty	Empty
Empty	Empty	Empty	Empty	Empty	Empty	Empty

Figure 8.14 - The grid after the free cells are marked.

Sort Outer Layer of Points.
Step thirteen sorts all points that still have connections in nextPoints and places them into the list connectionPoints. The points at the beginning of the list have the largest number of connections while those at the end have the least number of connections. The contents of connectionPoints will be (A1, A3, B1 and C2).

Step 13: *Sort all points, that have connections, in nextPoints into descending order according to the number of connections they have. Store the new list in connectionPoints.*

Placing an Outer Layer of Points.
Steps fourteen to twenty one iterate through all the points given in connectionPoints and place any points connected to them. Connection information is taken from the current version of the graph not the original. After each point is placed, according to the criteria in figure 8.13, it is added to the new list ,newConnectionPoints, and then marked as used.

Marking the points as used prevents them being reused by another point which may have a similar connection. If this type of marking was not used it would be possible to place more than one occurrence of the same point. Finally, the pointer to which point is being worked on is incremented and if there are still points unprocessed a jump is made back to step fifteen.

Step 14: *currentPoint = 1.*

Step 15: *Put point currentPoint of connectionPoints into thePoint.*

Step 16: *Sort all points connected to thePoint which are not marked used. The sort brings points that are connected, either directly or indirectly, close to each other. This is the same as the application of criteria one in step four.*

Step 17: *Place the points connected to thePoint, according to the information provided in figure 8.15. If one of these positions is blocked then that cell is skipped.*

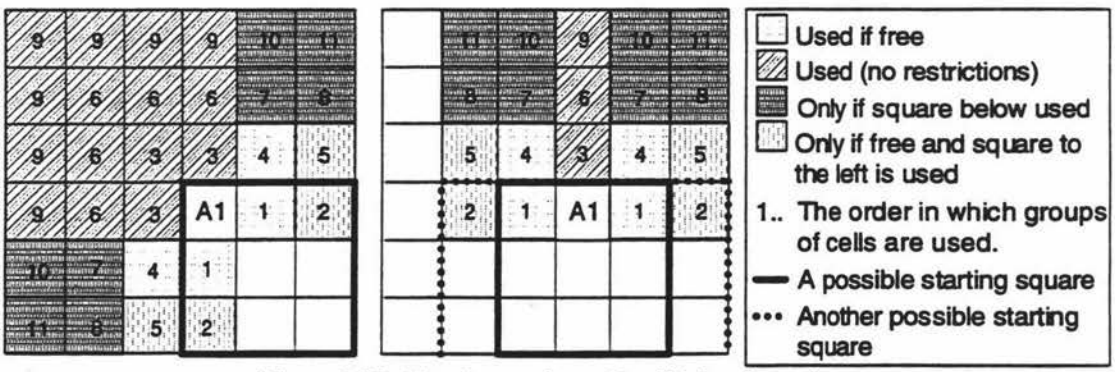


Figure 8.15 - The Areas where A1 will place its points.

Step 18: *Place all points added in step 17, that are not marked as used, into the list newConnectionPoints.*
Step 19: *Mark all the points added in step 17 as used.*
Step 20: *Increment the currentPoint.*
Step 21: *If there are still more unprocessed points in connectionPoints then go to step 15.*

The result of the iteration through steps fifteen to twenty one is shown in figure 8.16.

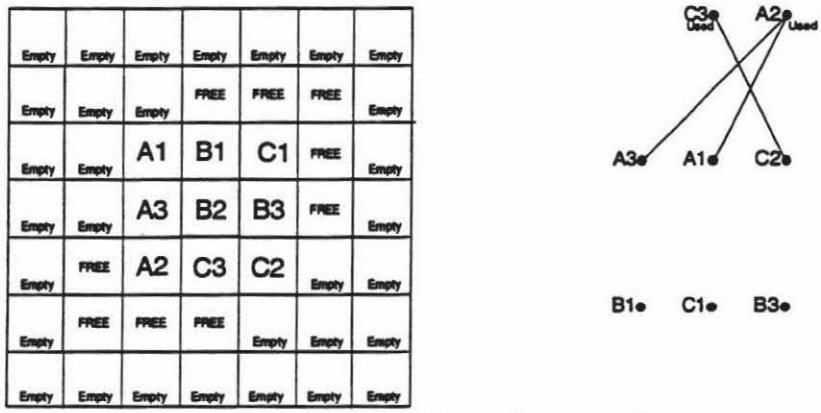


Figure 8.16 - The current state of the example.

Remove all Placed Points and Connections.
Steps twenty two through to twenty four tidy up after the placement process. Since the points and their connections represented in connectionPoints are no longer needed they are removed from the graph, figure 8.17. Finally the nodes that have just been placed are put into the connectionPoints list in case more processing is still needed.

Step 22: *Delete all connections from the points in connectionPoints.*
Step 23: *Delete all points in connectionPoints.*
Step 24: *Move all the points from newConnectionPoints to connectionPoints.*

Figure 8.17 - The graph after step twenty three.

Branch Back if more Point Need Placing.
At step twenty five connectionPoints contains only points 'A2' and 'C3', which currently have no connections. Therefore the jump in step twenty five is not invoked. This initial example terminates here because all the points are in a tightly knit formation.

Step 25: *If any of the points in connectionPoints have connections then go to step 14.*

At this stage the grid looks complete. However there is one important factor to be considered. In figure 8.16 the grid treats all points as equal sizes. This is unrealistic since each point represents a hierarchical graph of yet unspecified proportions. Figure 8.18 shows the proportions that characterise each point.

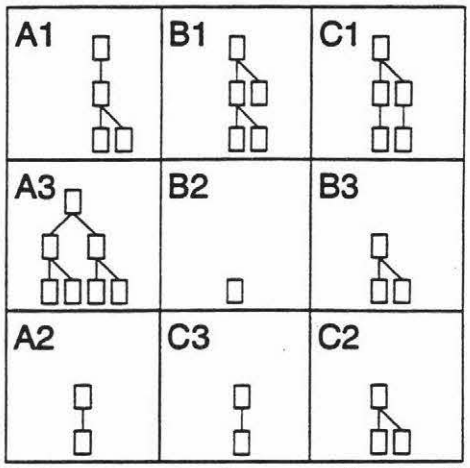


Figure 8.18 - The current point placements showing the point sizes.

The point positions given in the graph in figure 8.16 should still be considered as placement information, even though they are relative positions. Steps twenty six to thirty one outline how the positions are calculated to a more detailed level.

Place Hierarchies Directly out from the Centre.

The next two steps place the centre point along with the points directly above, below, to the right and to the left of the centre point. Figure 8.20 shows the placement with the shaded lines indicating the 'gap' mentioned in step twenty seven. Note that all point positions, both horizontal and vertical, are kept to a strict grid which can be seen in figure 8.21.

Step 26: *Place the centre point.*

Step 27: *Place the point directly above, below and to the left and right of the centre point excluding corner points. The offset from the centre point is:*
X= (half the (maximum width of W1 and W2)) + gap.
Y= (half the (maximum height of H1 and H2)) + gap
For the square around the centre point.
W1 = The top row of points except the corners.
W2 = The bottom row of points except the corners.
H1 = The left column of points except the corners.
H2 = The right column of points except the corners.
See figure 8.19 for additional information.
gap = 1 (temporarily), indeterminable at this stage.

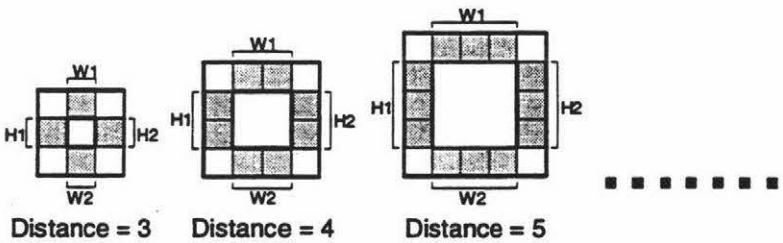


Figure 8.19 - A pictorial explanation for the values for W1,W2,H1 and H2.

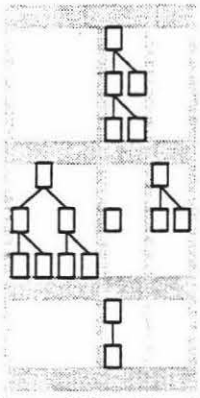


Figure 8.20 - Graph placement.

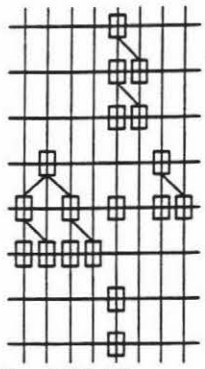


Figure 8.21 - Grid Alignment.

Place Corner Hierarchies.
Step twenty eight fits the corner points in snuggly against the current graph. Alignment is still kept with the grid but the points can cross the given 'gap' boundaries shown in figure 8.20. Figure 9.22 illustrates the graph at this stage.

Step 28: *Place the corners according to the border positions set out in step twenty seven and then try to move them closer to the centre. The movement of the points position is one horizontal and one vertical movement. This is stopped when both fail to make any progress.*

The bottom right most point , though it is a corner point, is positioned towards the centre. Since the middle bottom point was only one wide, the bottom right hand corner point was able to be pushed over. The movement towards the centre is restricted by the constraint that you cannot move past the centre most point or intrude upon another points coverage area. Figure 8.23 illustrates the normally invisible coverage areas.

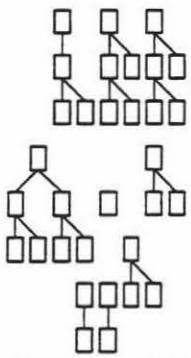


Figure 8.22 - Graph Placement.

Figure 8.23 - Coverage areas.

Moving the middle right-hand side point towards the centre point could also compact the graph more than in its present form. It does not violate any coverage areas and there is space available. What stops it being moved is that it is part of the initial positioning structure defined in steps twenty six and seven.

If this is compromised, problems can occur in connecting points. One such example from figure 8.23 is between the centre and the top right hand point. Moving the forementioned points in would make a connection path more difficult.

Place the outer Hierarchies.
The final set of steps iterates through the remaining layers of points and places them in a similar fashion to that indicated in steps twenty seven and eight. The major difference is that this time the points above, below, left and right of the currently placed points are jiggled the distance of one point to see whether they will fit in better.

Jiggling moves a point either up and down, if the point is on the sides, or left and right, if the point is on either the top or bottom. By trying one position in either direction a point can sometimes be made to fit in more securely.

Step 29: *Place the points that are not adjacent to the corners of the already placed points. Start placement outside of the outer boundary of the existing graph, figure 8.24. This is the same method as specified in step 28.*

Side point should be moved up and down one point position from centre to see whether they fit in any closer. Likewise the top and bottom points are moved left and right one point position from centre. If they can not then their previous position should be maintained.

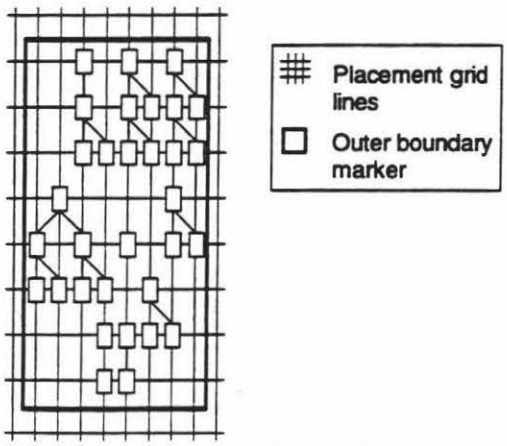


Figure 8.24 - The outer boundary marker shown with the placement grid visible.

Step 30: *Place the corner points in the same method as described in step twenty eight.*
Step 31: *While there is still another layer of points go to step 29.*

8.2 Connection Generation and Routing.

Since all the points have been placed it is time to add the connections between these points. To accomplish this the relationships between each point have to be identified. Figure 8.25 illustrates the connection relationships of each point. The connection lines are only shown here to indicate where connections will be placed.

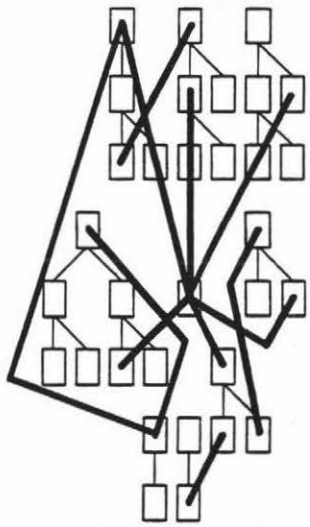


Figure 8.25 - The current graph with connection relationships showing.

The approach that is outlined below was influenced by ideas of Tamassia, Battista and Batini (1988). Their idea of tidy orthogonalisation uses a grid over the graph between which it places connections. The space between one grid line can only contain one connection line. Figure 8.26 shows an example from the paper by Tamassia *et al.*

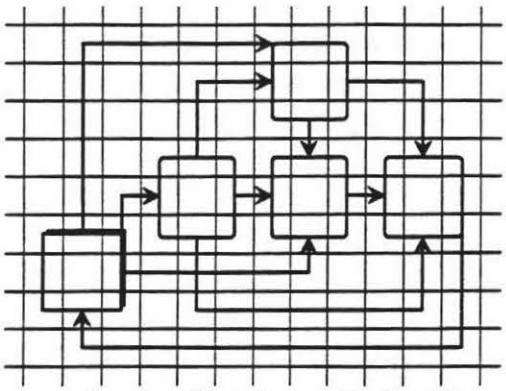


Figure 8.26 - An example using tidy orthogonalisation, Tamassia *et al* (1988).

The differences in the approach taken when compared with Tamassia *et al* are:

- The grid is courser.
- More than one connection can occupy a grid line.

The new grid lines for the current graph are shown in figure 8.27. A horizontal and vertical grid line intersect every node inside each point. There are also grid lines surrounding each node, both horizontally and vertically.

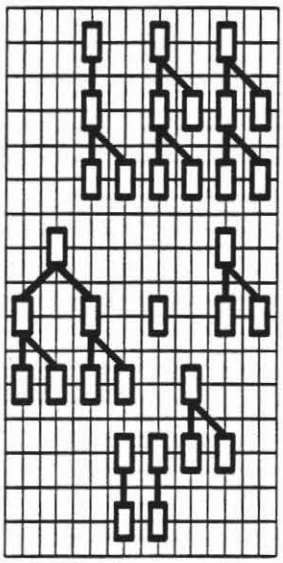


Figure 8.27 - The new grid pattern.

The two nodes that will be connected are identified and a rectangular region is placed around the parameter of the points that embody these nodes, figure 8.28. This encompassing region includes one grid line either side of the forementioned points. Figure 8.29 illustrate the region isolated from the overall graph.

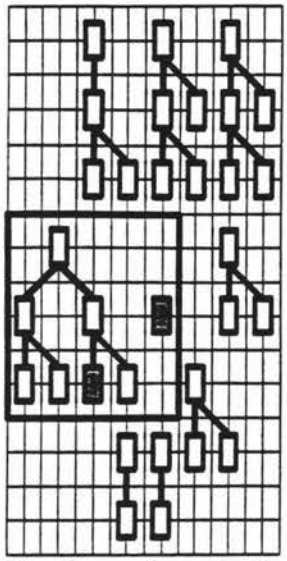


Figure 8.28 - Connection node identification.

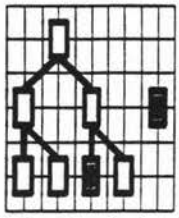


Figure 8.29 - The isolated region.

One arbitrary point is chosen as the start point. For the purpose of this thesis these connections will only originate from the sides. The following algorithm is then executed on the isolated region.

The Initial Setup of the Isolated Region.
In steps one and two, the isolated area is readied for routing. The removal of lines in step one is used to prevent any connections from taking short cuts through the centre of a point. This improves the aesthetics of the diagram and will prevent unnecessary crossings. Figure 8.30 shows the current example being worked on.

Step 1: *Remove the individual grid line segments that touch a connection.*
Step 2: *Identify the start points (first grid crossing either side of the node).*

Step two establishes the starting points. The starting points are where connections will be attempted from. Figure 8.31 illustrates the identification of the starting points for the current example.

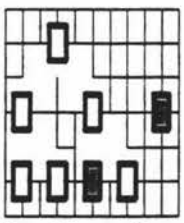


Figure 8.30 - Touching grid segments removed.

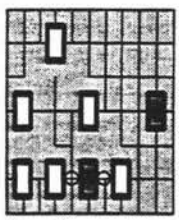


Figure 8.31 - Start points identified.

Catering for Special Circumstances.
The removal of segments in step one caters for all connections except when connecting to a node in a point with the following characteristics:

- Any of the nodes parents have at least two other nodes.
- The node has at least one other child node, with the same parent, on either side.

Under the removal parameters set out in step one this node would be isolated from all other nodes except its sibling nodes. Figure 8.32 illustrates a circumstance where this would happen.

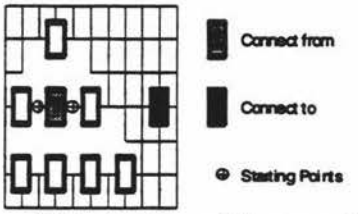


Figure 8.32 - An impossible connection.

To get around these circumstances, a path has to be created to these inner nodes. Steps four and six add these pathways. Steps three and five only initiate the additional pathways if the blockage is at that particular point. This will allow the connection to get in or out without allowing unnecessary and unwanted short cuts through the opposing point. Figure 8.33 shows the additional pathways.

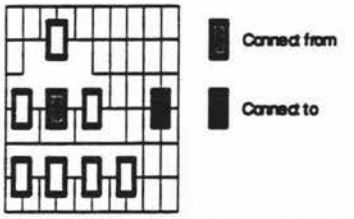


Figure 8.33 - An impossible connection.

There is no instance in the current example where the application of steps four and six are necessary.

Step 3: *If the total number of paths from the start points for the start node is greater than four, including the path back to the start node, then go to step 5.*

Step 4: *Add a vertical grid line down from each starting point. The line should go down until they reach the next horizontal grid line, if it was present.*
Add the missing horizontal grid line proceeding both sides of the 'from' node until it either reaches the side of the isolated region or the actual grid line for this horizontal position.

Step 5: *If the total number of paths from the finish points for the finish node is greater than four, including the path back to the finish node, then go to step 7.*

Step 6: *Add a vertical grid line down from each starting point. The line should go down until they reach the next horizontal grid line, if it was present.*
Add the missing horizontal grid line proceeding both sides of the 'to' node until it either reaches the side of the isolated region or the actual grid line for this horizontal position.

Setup the Initial Parameters.
The next steps initialise the number of bends and the distance travelled for both start positions.

Step 7: *Get a pointer for each of the first two start positions.*
Step 8: *Set both pointers to have a distance of 1 and 'bends' of 0.*

Setup the Prioritised Order of the Nodes.
Step nine places the start positions into a priority queue, right then left. The queue sorts its contents on the number of bends and if asked to return a value, it returns the first pointer found with the lowest number of bends. If there are more than one pointer with the same number of bends then the first found will be returned.

The sequence in which the nodes are placed in the queue will mean that any path on the right that has the same number of bends as the one on the left will arrive first. There is no evidence that would suggest this is more desirable, it is simply a matter of forced choice, one must come first.

Step 9: *Push the right and then the left pointer into a priority queue which sorts by lowest 'bends' number.*
Step 10: *Get (remove) the pointer with the lowest 'bends' from the queue.*

Retrieve Positional Information.
The values that will be used for the next intersection visited are set.

Step 11: *Set numberBends to bends for pointer.*
Step 12: *Set totalDistance to distance for pointer plus 1.*

Work Through Every Alternative Path.
Iterate through all possible paths, not including the path taken to get to this point, to facilitate the following indented actions for each path.

Step 13: *For each path for pointer (except the one travelled to get to this point).*

Calculate the Number of Bends to the New Position.
Information on the next grid intersection is then gathered. If the current path involved a corner to get to this point then increase the value containing the number of bends visited so far.

Step 14: *Get a pointer to the next intersection.*
Step 15: *If this path introduces a bend then increment numberBends by 1.*

Test for the Exit Condition.
If the destination for the connection has been reached then return the path that was taken to get to this point. The algorithm is then terminated since the best path possible has been found.

Step 16: *If the next intersection is the destination then return the path and STOP.*

Avoid Inefficient Paths.
If the intersection that is about to be visited has already been visited then its current values are checked. If they are lower than either numberBends or totalDistance then this path is no longer followed up. A jump is made to step thirteen to allow other paths to be considered.

Step 17: *If the next intersection has a lower bends or lower distance then go to step 13.*

Prioritise the Next Position.
The intersection whose pointer was obtained in step eight has its values for bends and distance updated. The values of numberBends and distance are used in the update. The pointer is then placed on the priority queue. Step nineteen is the looping point for the path iterations set out in step thirteen.

Step 18: *Set the values for pointer to the next intersection to numberBends and distance.*
Step 19: *Place the pointer into the priority queue.*

Iterate Through more Points.
Finally step twenty jumps back to step four to continue working on the contents of the priority queue. This is an unconditional jump because it is always possible in every case for a path to be found and the algorithm to exit at step sixteen. The addition of grid lines through and around each point is the reason a path is always possible.

Step 20: *Go to step 10.*

Figure 8.34 illustrates the algorithm outlined above. This is not the example that is currently being worked through because it is too complicated to show. Instead a more simplistic example is used to clarify the problem.

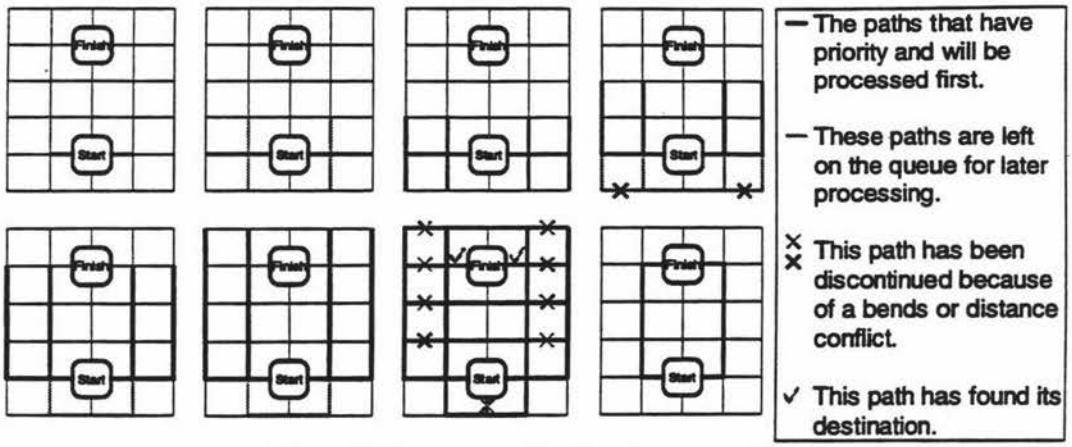


Figure 8.34 - An example of routing connections.

It should be noted that out of the final two possible paths the one on the right would be chosen. This is because at the start the right side is handled first which means it will reach the finish first. As the algorithm currently stands, after the first connection reaches the end it is terminated and no more paths are calculated.

When the algorithm is applied to the example area from figure 8.29 the result is the connection shown in figure 8.35.

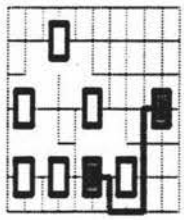


Figure 8.35 - The path chosen by the algorithm.

The same procedure is then applied for all other connections. Figure 8.36 illustrates the isolated connection regions for the other connections.

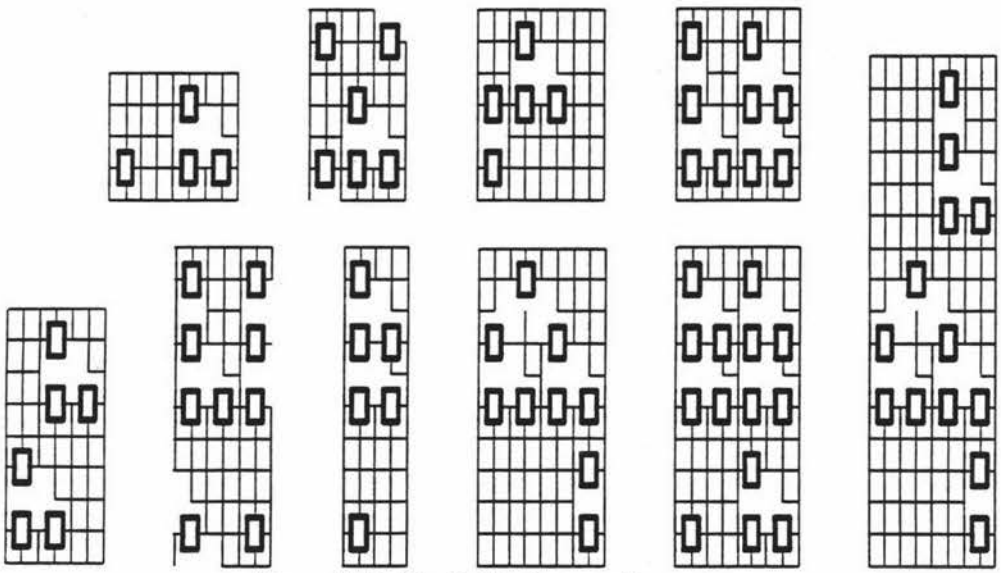


Figure 8.36 - The isolated connection regions.

Upon identifying these regions the algorithm is applied. The results are shown in figure 8.37.

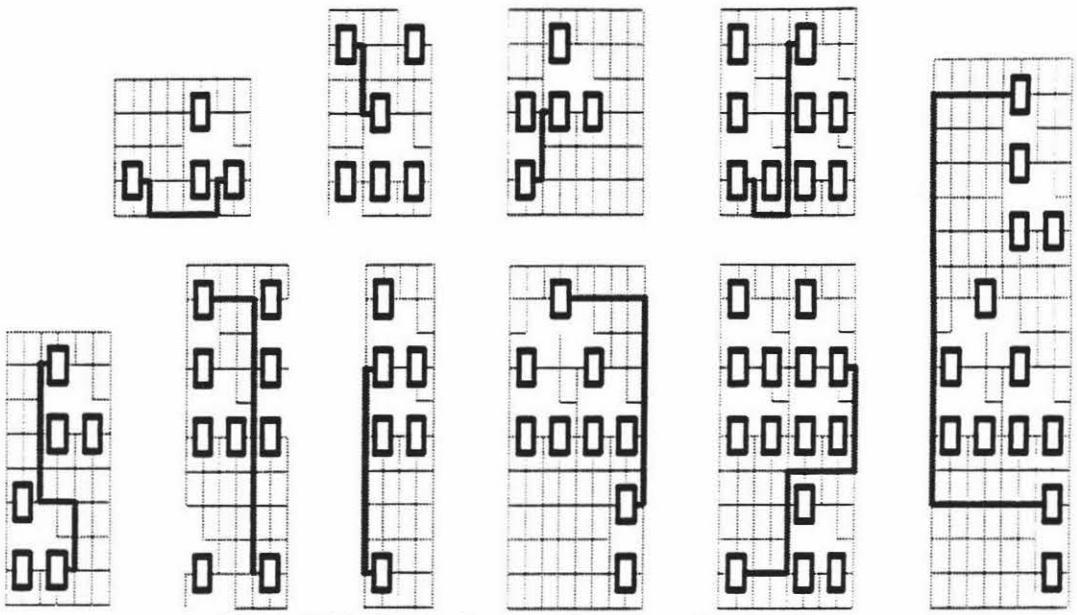


Figure 8.37 - The application of the connection algorithm.

Individualisation of Connections.

The connections have been added but the routing problem is not complete. There is still the problem of more than one connection occupying the same grid line. This is not evident in figure 8.37 but if the graph is looked at as a whole, figure 8.38, then it is obvious.

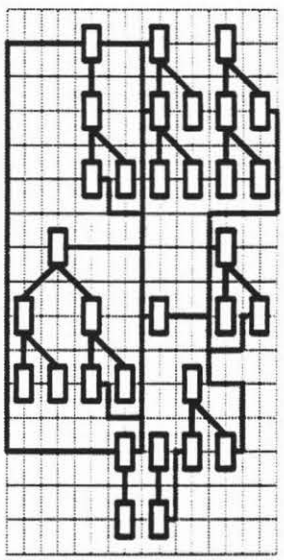


Figure 8.38 - The entire graph with connections added.

To encompass multiple connections on a grid line, the maximum number of connections at any point has to be calculated. This is done for all horizontal and vertical grid lines. This information is then used to calculate the width of the grid line. This is also the value of 'gap' mentioned in step twenty seven of the previous sub-section.

Figure 8.39 shows the count values for both the vertical and horizontal grid lines. These values are used to expand the grid at the given grid line. The end result is extra space to add connections. Figure 8.40 shows the expanded graph.

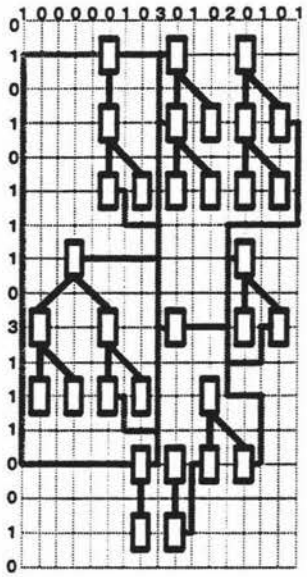


Figure 8.39 - The maximum connection values.

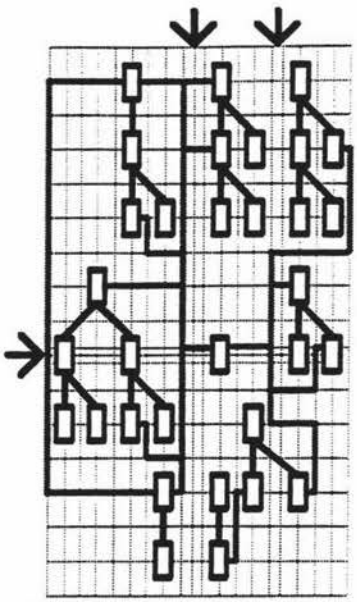


Figure 8.40 - The expanded graph

The use of these extra grid lines is the next problem to solve. The connections can not just be placed on any line. If they were then more crossings would occur, even in the best possible case. A placement algorithm needed to be defined to minimise these crossing, increasing the aesthetics of the graph.

Solution: *The first connection that turns left from a group of connections is placed in the left most position. The second connection to turn left is then placed in the second position from the left, etc. The same logic is applied to the right side with placements starting from the right. If a conflict area occurs when trying to draw from both ends of a connection then insert an extra grid line at one of the conflicts and try again. Figure 8.41 illustrates a conflict.*

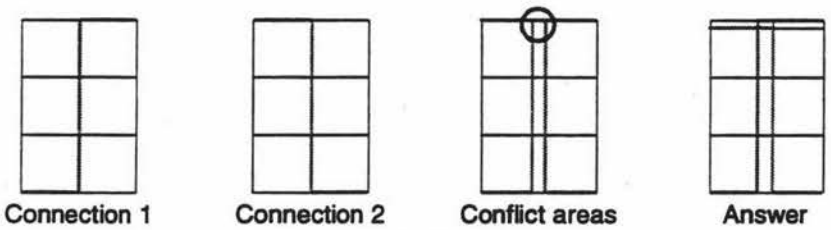


Figure 8.41 - Problem fix.

If an intersection point that is required for placement is already used, figure 8.42, follow the same rules as set out in the previous diagram.

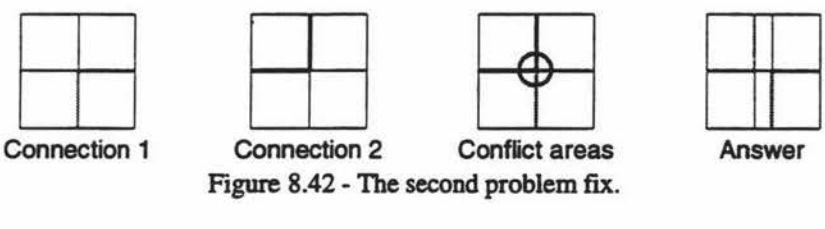


Figure 8.42 - The second problem fix.

Figure 8.43 below shows the result of the solution being applied to the expanded graph. The problem outlined in figure 8.41 actually occurred when replacing the connection lines. As a result an extra grid line was added making four vertical lines through the centre of the graph.

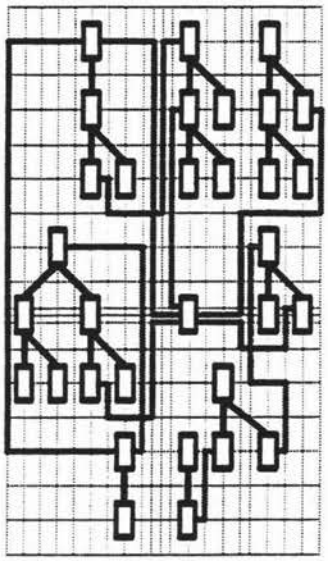


Figure 8.43 - The graph when the connections can take advantage of the extra grid lines.

At this stage the graph is just about ready to be plotted. The only information now required is the height and width of every node in the graph. This information is used to calculate the distance between the grid lines.

In the graph shown above the second horizontal grid line from the top intersects three nodes. The maximum height of these nodes plus padding is the distance between the first and third horizontal grid line. The process is done for the other horizontal grid lines as well as the vertical lines, which use the maximum width.

Since this part of the CASE tool has not actually been completed at this stage an example produced by the tool can not be shown. Instead the example from figure 8.43 will be continued with random values used for the height and width of the individual nodes. Figure 8.44 shows the final graph.

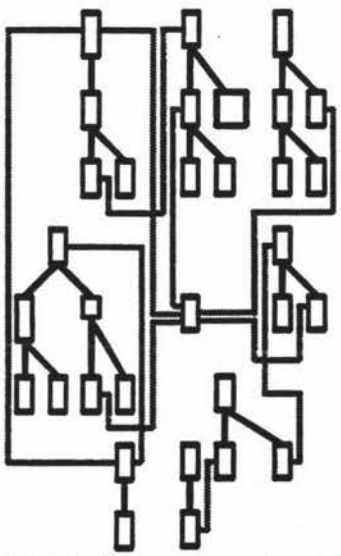


Figure 8.44 - The final presentation of the graph.

Even though the diagram above may be seen as less than perfect, the complexity involved should be remembered. There are undoubtedly other heuristic's that could enhance the aesthetics of the diagram. However what was sought here was a relatively simple answer.

9. Diagramming Deficiencies.

There are two deficiencies with the current specification both of which are in the non-hierarchical graph sub-section. The deficiencies are for the placement of connecting points around the centre and for placing connecting points further out.

When points are placed around the centre position they are held at a distance calculated by the size of the points above, below and to the side of the centre. If one of the points to the side was very tall or one of the points on the top or bottom was very wide then the algorithm would not be as effective.

Depending on the size of the centre point this could cause a large empty space in the centre of the diagram. This would lower the quality of the diagram because the information available would be very spread out.

The second deficiency involves the placement of points further out. In figure 8.15 there are a large number of positions allocated for placement of connection points, which can be expanded upon if needed. As these build up several layers deep they hinder connections and the positioning of future points.

The connections hindered are for the point having its connecting points added. Connections may have to weave around existing points to get to its destination. As the layers build up, the positioning of the points connected to the points in the layers becomes increasingly difficult.

9.1 Future Enhancements.

Since the prototype in chapters six to eight have not been fully implemented it is not always possible to find all areas of future enhancement. When completed, the operations of both the parsing and diagramming sub-sections can be examined in a more easily judgable situation.

Undoubtedly most evaluations will show that extra rules can be added to enhance the non-hierarchical graph sub-section. There has to be a tradeoff between functionality and the size of the code. Current estimates of the implemented code size are approximately fifty percent of the CASE tools code final size. This is very large percentage, so any additions will have to be justified in terms of what functionality they provide.

Language Specification.
In the future it is envisioned that the parser will be able to accept a language definition file. This would then specify how it would accept, manipulate, display and save a language. This would allow the tool to be language independent as well as methodology independent. This enhancement would provide a a customisable tool which will be adaptable to any user specific requirements.

Understanding Relationships.
There are inherent difficulties in converting code, of any sort, into a visual diagrammatic representation. When a person writes or reads code they gain a different level of understanding than is possible for a program. These differences may be small but they can make a considerable difference. For example the Coad and Yourdon relationship of whole-part relationship and association. The former relationship refers to one classes reliance upon another class to make up the whole entity. The latter refers to circumstances where one class may possibly have another class in its makeup. Both relationships are attributes of a class. An association would have to be implemented as a pointer to a class type. The problem with this is that a whole-part relationship can be shown in the exact same way. Only a persons overall understanding of the class, and maybe the system, can solve such a problem.

The same applies to the mapping's of a relationship, whether class X can have zero to many instance connections of class Y. If the attribute that represents this relationship is a pointer then it is uncertain whether it is a zero or one relationship. Even harder than this are one to many relationship implemented through an array template or an array class. No program would be able to understand the importance of these classes.

There are two ways to deal with this problem. The first is to ignore the parts of relationships where the problems happen. This is not a very good answer. If a notation is chosen to represent the data then it is not very practical to ignore parts of it. The other alternative is to rely on configuration files and the user for information that the tool can not provide. When a notation is loaded for the first time the user corrects the wrong information by supplying the missing information. The information is saved as code and another specification file.

This file specifies any information that the user has given that is not available from the code files. The next time the code is loaded, it is used in conjunction with the special information file. The combination of files information could provide the best overall answer.

9.1.3 Saving Template.

Another practical addition to this part of the CASE tool are saving templates. These would encapsulate the structure that C++, or another language, will be saved as. This will allow a user to change the structure of their code, or someone else's, to meet an organisations guidelines.

This feature will complement the existing save function which tries to keep the same structure as the original code. The user will have the ability to either save in the same format or to save according to a save template.

10. Conclusion

The implementation of an initial prototype of a methodology independent CASE tool has successfully shown that this is a viable project for the Object Oriented Development Research group which will benefit software companies dealing with software construction. The ideas and achievements set out in the thesis and in the prototype show that a full system development is both a realistic and viable project. Implementation of such a system could be produced at a relatively low cost.

Moot successfully shows that this technology is viable in a small system development. The CASE tool is only of moderate size but it already shows the capacity to present a versatile and highly adaptive diagramming environment. An environment that overcomes pitfalls present in most of today CASE tools.

The main strength of the tool is the flexible notation description language. Users can define a large number of new methodologies specifications or adapt the existing ones. Any limitations placed on the development of a methodology are from the user, not the system.

The benefits from this are prevalent not only in commercial organisations but also in academic circles as well. New methodologies that were once restricted to pen and paper can now be tested and refined on a working system. Such flexibility can greatly increase the number of new methodologies and the speed at which they are developed bringing continuing benefits to the software engineering community.

The re-engineering features of the tool show the potential to move Moot out of the realm of a diagramming tool into code generation and reuse. The object oriented approach has helped focus the software engineering community on reuse of code. Moot provides maximum coverage by augmenting reuse through facilities to handle new and existing code.

By not complying to the exact grammatical specification of C++ the CASE tool is able to tolerate minor mistakes in coding. Such parsing techniques allow the user to use Moot as a trouble shooting and problem identification tool in comparison to the Borland coding environment where partially complete code can not be handled.

Every effort is made to keep the reworked code in a similar format to what was initially read in. Code is kept in the same files and specifications are kept as closely as possible to the original structure. If the code files are examined outside of the tool at a later date they will be more inline with the users style of programming.

However the user is not just restricted to this style. Moot will also be able to re-range the coding structure to particular styles. This will benefit the user re-engineering code that is not their own or code that lacks the structure they prefer.

The diagrammatic representation of the data elicited from C++ shows promise. Even though this section of the tool is incomplete the examples worked through illustrate how a difficult and complicated problem can be overcome. Placing graphs with a minimal number of crossings is an NP-Hard problem. All that can be achieved is placement of nodes so that their aesthetics are increased.

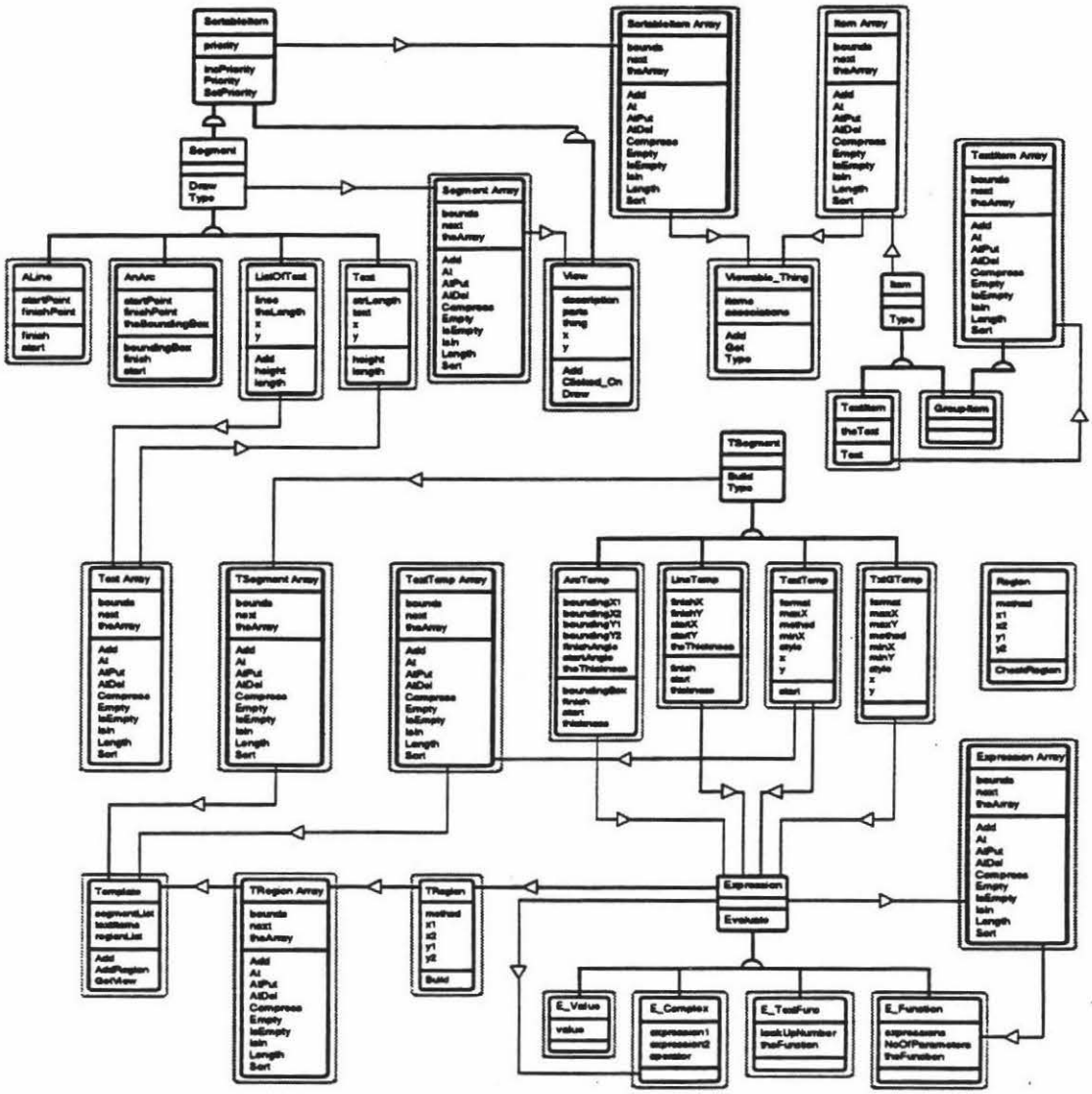
The best possible placement is generated by using heuristic's which are to the users advantage. Connection lengths and crossing of connections are minimised where ever possible. Placement of nodes and hierarchies is facilitated in the most aesthetically pleasing layout so that readability and understandability is maximised.

It is always possible to create a structure that will not be very readable after placement. The algorithms mention in this thesis aim to provide the best possible aesthetics when processing and time limitations are taken into account. All examples that have so far been applied to these algorithms have been easier to read and understand after their transformation.

The main question that this thesis has to answer is should development continue? The answer to this is yes. The progress in the CASE tool so far shows that this is very successful. The diagrams in this thesis that have been drawn by the tool as well as the worked examples from the re-engineering sections all show positive results. The culmination of existing and future work should provide the basis for a powerful, flexible, configurable, customizable and easy to use tool. These generic features of the new CASE tool will allow it to benefit all sectors and companies dealing with software construction.

Appendix A

The Initial Object Model



Appendix B

Template Definition Grammar

```
<Template>              ::= Notation <String> <Template_Collection> End
<Template_Collection>   ::= <Group_Templates> <Object_Templates> <Connection_Templates>

<Group_Templates>       ::= Template Group <Group> End | ε
<Group>                 ::= Shape <Equations> <Segments> End <Group> | Shape
                            <Equations> <Segments> End

<Equations>             ::= Equations <Each_Equation> End | ε
<Each_Equation>         ::= = <Expression> <Each_Equation> | ε

<Expression>            ::= <Expression> + <Term> | <Expression> - <Term> | <Term>
<Term>                  ::= <Term> * <Factor> | <Term> / <Factor> | <Factor>
<Factor>                ::= <Float> | ( <Expression> ) | <TextFunc> | <Function> |
                            <EQ_Ref>

<Float>                 ::= <Digit> <Float> | . <Integer> | <Digit> | .
<Integer>               ::= <Digit> <Integer> | <Digit>
<Digit>                 ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<TextFunc>              ::= Property( <Integer> , <FuncToDo> )
<FuncToDo>              ::= Height | Length

<Function>              ::= Max( <ListOfExpr> ) | Min( <ListOfExpr> )
<ListOfExpr>            ::= <Expression> | <Expression> , <ListOfExpr>

<EQ_Ref>                ::= EQ( <Integer> )

<Segments>              ::= <Arc_Template> <Segments> | <Line_Template> <Segments> |
                            <Text_Template> <Segments> | <List_Template> <Segments> |
                            <Group_Reference> <Segments> | ε

<Group_Reference>       ::= Group( <Integer> ) at <Expression> , <Expression>

<Arc_Template>          ::= Arc in box <Expression> , <Expression> , <Expression> ,
                            <Expression > Starting <Expression> Extent <Expression>
                            Thickness <Expression> Colour Line <Colour> Fill <Colour>
                            Pattern <Pattern>

<Line_Template>         ::= Line <Expression> , <Expression> , <Expression> , <Expression>
                            <Line_Variations>
<Line_Variations>       ::= <MoreExpr> Thickness <Expression> Colour Line <Colour> Fill
                            <Colour> Pattern <Pattern> Arrows <Arrows> | Thickness
                            <Expression> Colour <Colour> Pattern <Pattern> Arrows
                            <Arrows>
<MoreExpr>              ::= , <Expression> , <Expression> | , <Expression> , <Expression>
                            <MoreExpr>
<Arrows>                ::= Beginning | End | Both | None

<Text_Template>         ::= Text at <Expression> , <Expression> Width <Expression> MaxX
                            <Expression> MinX <Expression> Style " <StyleType> " Format
                            ' <FormatType> ' ID <Integer> Font <String>

<StyleType>             ::= N | <StyleAll>
<StyleAll>              ::= B <StyleUI> | U <StyleBI> | I <StyleBU>
<StyleUI>               ::= U <StyleI> | I <StyleU> | ε
```

```
<StyleBI>            ::= B <StyleI> | I <StyleB> | ε
<StyleBU>            ::= B <StyleU> | U <StyleB> | ε
<StyleB>             ::= B | ε
<StyleU>             ::= U | ε
<StyleI>             ::= I | ε

<FormatType>         ::= L | C | R

<String>             ::= " <String_Parts> "
<String_Parts>       ::= <AlphaNum> <String_Parts> | ε
<AlphaNum>           ::= <Letters> <AlphaNum> | <Integer> <AlphaNum> |
                         <Special_Characters> <AlphaNum> | ε
<Letters>            ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q |
                         r | s | t | u | v | w | x | y | z | A | B | C | D | E | F | G |
                         H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V |
                         W | X | Y | Z
<Special_Characters> ::= ` | ~ | ! | @ | # | $ | % | ^ | & | * | ( | ) | { | } | [ | ] | ; |
                         : | < | > | , | . | / | ? | - | _ | + | = | \ | "| '

<List_Template>      ::= List at <Expression> , <Expression> to <Expression> ,
                         <Expression> MaxX <Expression> MinX <Expression> MaxY
                         <Expression> MinY <Expression> Style " <StyleType> " Format '
                         <FormatType> ' ID <Integer> Font <String>

<Object_Templates>   ::= Template Object <String> <Object_Collection> End
                         <Object_Templates> | ε
<Object_Collection>  ::= <Equations> <Segments> <Regions> <Redefine_IDs>

<Regions>            ::= Regions <Each_Region> End | ε
<Each_Region>        ::= = <Expression> , <Expression> , <Expression> , <Expression>
                         Action <Integer> With <TextOrID> | ε

<TextOrID>           ::= <String> <TextOrID> | <String> | ID( <Integer> ) <TextOrID> |
                         ID( <Integer> )

<Redefine_IDs>       ::= Redefine ID <Redefine> End | ε
<Redefine>           ::= <Integer> Named <String> <Redefine> | ε

<Connection_Template> ::= Template Connection <String> <Connection> End | ε
<Connection>         ::= Connect from ( <ListOfInteger> ) to ( <ListOfInteger> ) using
                         <ConnectLine> End <Connection> | ε
<ListOfInteger>      ::= <Integer> , <ListOfInteger> | <Integer>

<ConnectLine>        ::= Start <SFPoint> Finish <SFPoint> Lines <Lines> Thickness
                         <Expression> Colour <Colour> Pattern <Pattern> Arrows
                         <Arrows> <C_Segments> <Regions> <Redefine_IDs>
<SFPoint>            ::= Normal | <PorM> <Expression>
<PorM>               ::= Plus | Minus
<Lines>              ::= Single | Poly | HVPoly

<C_Segments>         ::= <C_Arc> <C_Segments> | <C_Line> <C_Segments> | <C_Text>
                         <C_Segments> | <C_List> <C_Segments> | <C_Group>
                         <C_Segments> | ε
<C_Arc>              ::= <Arc_Template> At <Point>
<C_Line>             ::= <Line_Template> At <Point>
<C_Text>             ::= <Text_Template> At <Point>
```

<C_List> ::= <List_Template> At <Point>
<C_Group> ::= <Group_Reference> At <Point>
<Point> ::= <EitherEnd> <PorM> <NoOrPercent> Offset <Expression>
<EitherEnd> ::= Start | Finish
<NoOrPercent> ::= <Expression> | <Integer> Percent <PercentOption>
<PercentOption> ::= <PorM> <Expression> | ε

Appendix C

Tokens

There are eighty seven different tokens used in the parser to facilitate the recognition of any template file. These tokens are:

ACTION	AMPERS = '&'	AND	ANY	ARC
ARROWS	AT = '@'	BEGIN	BEGINNING	BOTH
BOX	CLP = '{'	COLON = ':'	COLOUR	COMMA = ','
CONNECT	CONNECTION	CRP = '}'	DIV = '/'	END
EQ	EQUALS = '='	EQUATIONS	ERRORRR	EXTENT
FILEEND	FILL	FINISH	FORMAT	FROM
GREATER = '>'	GROUP	HASH = '#'	HEIGHT	HVPOLY
ID	IN	LENGTH	LESS = '<'	LINE
LIST	LP = '('	MAX	MAXX	MAXY
METHOD	MIN	MINUS	MINX	MINY
MULT = '*'	NAME	NAMED	NONE	NORMAL
NOTATION	NUMBER	OBJECT	OFFSET	PATTERN
PERCENT = '%'	PIPE = '|'	PLUS = '+'	POLY	PROPERTY
REDEFINE	REGION	RETURN	RP = ')'	SEMI = ';'
SLP = '['	SRP = ']'	TEMPLATE	TEXT	THICKNESS
TILDER = '~'	TO	SEGMENTS	SINGLE	SLASH = '\'
START	STRING	STYLE	USING	VALUE
WIDTH	WITH			

The tokens NUMBER, STRING and VALUE refer to a floating point number, a list of characters of characters surrounded by quotation marks and a single character enclosed in single quotes. All non-tokenised information has to be contained in one of the three previous forms.

Also recognised, but not mentioned above are comments. These start with "//", end with a carriage return and can contain any characters or tokens. This is not listed among the tokens since it never tokenised. If the parser finds a comment while looking for a token it skips over and looks for the next token.

Appendix D

Steps for Padding Nodes in a Hierarchical Graph

Step 1: *For each level of nodes (going from second to bottom upwards then the second to top downwards) do*

Step 2: *padding = the maximum number of nodes at any levels minus the number of nodes in this level*

Step 3: *If the direction for step 1 is going down then*
connectedTo = the level above this level
else
connectedTo = the level below this level

Step 4: *NumOfNodesAtThisLevel = the number of nodes at this level*
CurrentNode = 1

Step 5: *While padding is greater than zero and*
Current is less than equal to NumOfNodesAtThisLevel do

Step 6: *If node Current has one or more connections to level connectedTo then*
Step 7: *If the centre most connection is to the left of Current then*
Step 8: *insert padding node at this position.*
Step 9: *Increment Current by one*

Steps for Positioning Hierarchies in a Non-Hierarchical Graph

Step 1: *Find the point with the largest number of connections.*

Step 2: *Prepare a grided square area full of empty cells that has a length and height of*
((square root of the number of points rounded down to an integer) + 4)
For the example being used the working would be
= ((square root of 9 rounded down to an integer) + 4)
= ((3 rounded down to an integer) + 4)
= ((3) + 4)
= 7

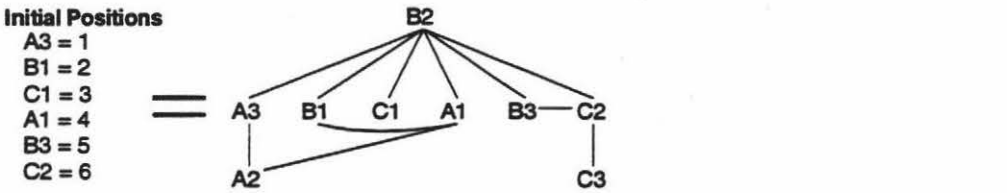
Step 3: *Place the cell with the largest number of connections into the middle cell.*

Step 4: *distance = ((Number of connections to the centre point / 4) rounded up) +1*
If distance is less than three then it is given the value of three

Examples.
14 points = ((14 / 4) rounded up) + 1 = (3.5 rounded up) +1 = 4 + 1 = 5
3 points = ((3 / 4) rounded up) + 1 = (0.75 rounded up) + 1 = 1 + 1 = 2 ⇒ 3

Step 5: *Sort and position the points connecting to 'B2' in accordance with the seven following criteria, which are illustrated in figures D.1 and D.2.*

(1) Find new position values for the connections. These new values are calculated by the averaged positions of any connections to other points at the same level. The lists marked with a (1) in figure D.1 show the working and results for this criteria.
It should be noted that if a point refers to an already calculated point then this points new value is used. For example the point A1 would use the new information from point A3 and B1.
(2) Sort each group, points with the same new value, by the number of connections to points at the same level. The resulting list is placed in descending order.
(3) Sort the groups, keeping all points in a group together, according to the number of points in the group.

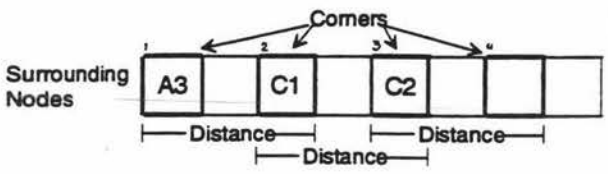


(1) Averaged Relationships*	(1) New values	Number of External connections**	(2) Sort groups in descending order of connections**	(3) Sort, in groups, by the number of nodes in the group
C1 = C1 / 1	C1 = 3	0	C1 = 3	A3 = 4
A3 = A1 / 1	A3 = 4	1	A3 = 4	A1 = 4
B1 = A1 / 1	B1 = 4	0	A1 = 4	B1 = 4
B3 = C2 / 1	B3 = 6	0	B1 = 4	C2 = 6
C2 = B3 / 1	C2 = 6	1	C2 = 6	B3 = 6
A1 = (B1 + A3) / 2	A1 = 4	1	B3 = 6	C1 = 3

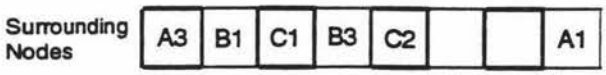
* Nodes with the smallest number of connections to either C1, A3, B1, A1, B3 or C2 re calculated first, then the next smallest, etc.
** These are only connections to nodes not at the same level, eg connections to nodes other than A1, A3,B1,B3,C2 and C1

Figure D.1 - The application of criteria 1 to 4 of step 5.

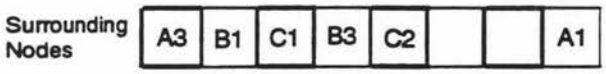
*(4) Create an array of ((distance * 2) + ((distance - 2) * 2)) length.*
(5) Place the start points of up to the first eight groups into firstly the corner positions and then the centre cells between the corners. Use the corners in this order: 1, 3, 2, 4.
(6) Place the rest of the points in the group around the already placed point.
(7) Place any remaining points in the left over empty positions, from left to right.



(5) Using the length 'distance' place the first node of each group in the corner positions then the centre squares between corners. Use corners in this order 1,3,2 then 4



(6) Surround the placed nodes with nodes of the same group..



(7) Place the remaining nodes into the free squares going from left to right

Figure D.2 - The application of criteria 5 to 7 of step 5.

Step 6: *Place all the points onto grid starting at position*
x = middle - ((distance -1) / 2) rounded down.
y = x.
Working for the given example is:
x = 4 - (((3 - 1) / 2) rounded down) = 4 - (1 rounded down) = 3
y = 3
This will place the cells around the outside of a square with sides of distance length. All point positions, used and not used, are placed into the pointPos list.

Step 7: *Mark all cells not used FREE.*

Step 8: *Place all the points connected to the centre point into the nextPoints list.*

Step 9: *.Delete all connections between the centre point, 'B2', and any other point.*

Step 10: *Delete the centre Point, 'B2'*

Step 11: *For all points positions in the nextPoints list do*
if the current point connects to any other cell in the nextPoints list then
delete the connection

Step 12: *For all cells in nextPoints do*
if this cell is free or the point has no connections then
Mark the cell(s) directly out from this position as free, figure D.3.

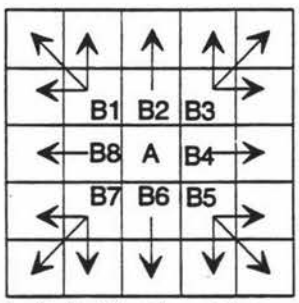


Figure D.3 - Cells directly out from other cells.

Step 13: *Sort all points, that have connections, in nextPoints into descending order according to the number of connections they have. Store the new list in connectionPoints.*

Step 14: *currentPoint = 1*

Step 15: *Put point currentPoint of connectionPoints into thePoint*

Step 16: *Sort all points connected to thePoint which are not marked used. The sort brings points that are connected, either directly or indirectly, close to each other. This is the same as the application of criteria one in step four.*

Step 17: *Place the points connected to thePoint, according to the information provided in figure D.4. If one of these positions is blocked then that cell is skipped.*

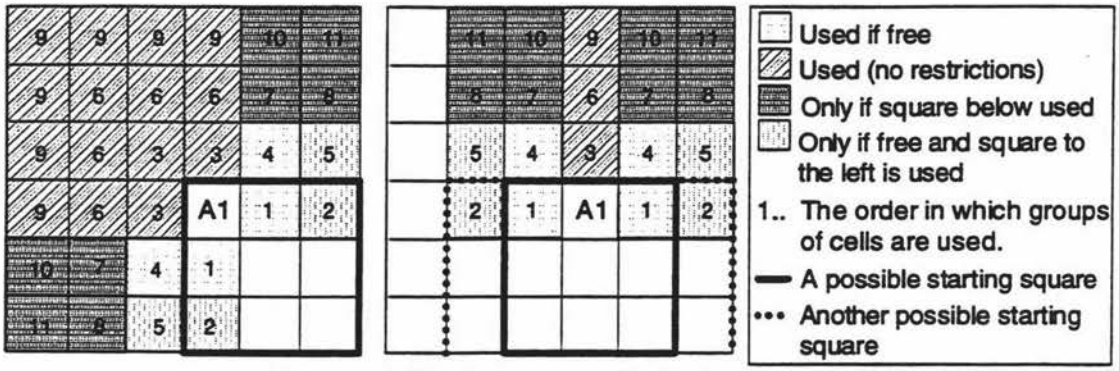


Figure D.4 - The Areas where A1 will place its points.

Step 18: *Place all points added in step 17, that are not marked as used, into the list newConnectionPoints*

Step 19: *Mark all the points added in step 17 as used.*

Step 20: *Increment the currentPoint.*

Step 21: *If there are still more unprocessed points in connectionPoints then go to step 15.*

Step 22: *Delete all connections from the points in connectionPoints.*

Step 23: *Delete all points in connectionPoints.*

Step 24: *Move all the points from newConnectionPoints to connectionPoints.*

Step 25: *If any of the points in connectionPoints have connections then go to step 14.*

Step 26: *Place the centre point.*

Step 27: *Place the point directly above, below and to the left and right of the centre point excluding corner points. The offset from the centre point is:*

X= (half the (maximum width of W1 and W2)) + gap.

Y= (half the (maximum height of H1 and H2)) + gap

For the square around the centre point.

W1 = The top row of points except the corners.

W2 = The bottom row of points except the corners.

H1 = The left column of points except the corners.

H2 = The right column of points except the corners.

See figure D.5 for additional information.

gap = 1 (temporarily), indeterminable at this stage.

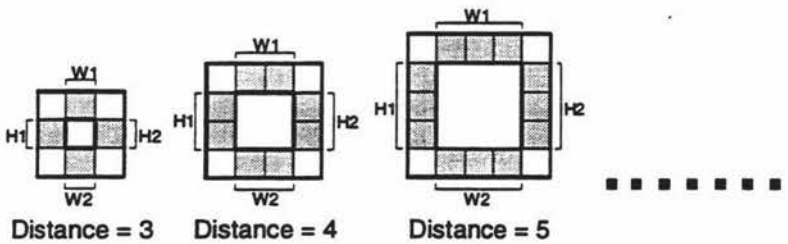


Figure D.5 - A pictorial explanation of the values for W1,W2,H1 and H2.

Step 28: *Place the corners according to the border positions set out in step twenty seven and then try to move them closer to the centre. The movement of the points position is one horizontal then one vertical movement. This is stopped when both fail to make any progress.*

Step 29: *Place the points that are not adjacent to the corners of the already placed points. Start placement outside of the outer boundary of the existing graph, figure D.6. This is the same method as specified in step 28.*

Side point should be moved up and down one point position from centre to see whether they fit in any closer. Likewise the top and bottom points are moved left and right one point position from centre. If they do not then there previous position should be maintained.

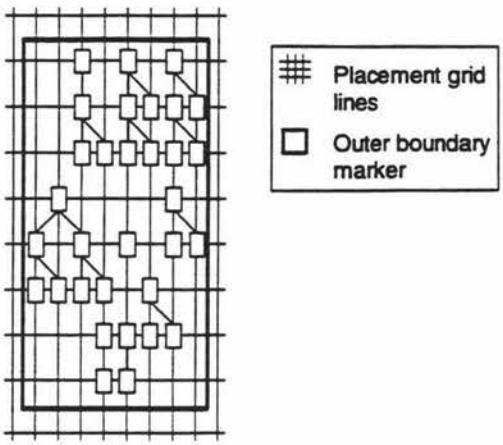


Figure D.6 - The outer boundary marker shown with the placement grid visible.

Step 30: *Place the corner points in the same method as described in step twenty eight.*
Step 31: *While there is still another layer of points go to step 29.*

Connection Generation and Routing Steps for a Non-Hierarchical Graph

Step 1: *Remove the individual grid line segments that touch a connection.*
Step 2: *Identify the start points (first grid crossing either side of the node).*

Step 3: *If the total number of paths from the start points for the start node is greater than four, including the path back to the start node, then go to step 5.*

Step 4: *Add a vertical grid line down from each starting point. The line should go down until they would reach the next horizontal grid line, if it was present.*
Add the missing horizontal grid line proceeding both sides of the 'from' node until it either reaches the side of the isolated region or the actual grid line for this horizontal position.

Step 5: *If the total number of paths from the finish points for the finish node is greater than four, including the path back to the finish node, then go to step 7.*

Step 6: *Add a vertical grid line down from each starting point. The line should go down until they would reach the next horizontal grid line, if it was present.*
Add the missing horizontal grid line proceeding both sides of the 'to' node until it either reaches the side of the isolated region or the actual grid line for this horizontal position.

Step 7: *Get a pointer for each of the first two start positions.*
Step 8: *Set both pointers to have a distance of 1 and bends of 0.*

Step 9: *Push the right then the left pointer onto a priority queue which sorts by lowest bends number.*
Step 10: *Get (remove) the pointer with the lowest bends from the queue.*

Step 11: *Set numberBends to bends for pointer.*
Step 12: *Set totalDistance to distance for pointer plus 1.*

Step 13: *For each possible path for pointer, that is not marked used.*

Step 14: *Get a pointer to the next intersection.*
Step 15: *If this path introduces a bend the increment numberBends by 1.*

Step 16: *If the next intersection is the destination then return the path and STOP.*

Step 17: *If the next intersection has a lower bends or lower distance then go to step 13.*

Step 18: *Set the values for pointer to the next intersection to numberBends and distance.*
Step 19: *Place the pointer into the priority queue.*

Step 20: *Go to step 10.*

Solution for Preventing Routing Conflicts in Non-Hierarchical Graphs

Solution: *The first connection that turns left from a group of connections is placed in the left most position. The second connection to turn left is then placed in the second position from the left, etc. The same logic is applied to the right side with placements starting from the right. If a conflict area occurs when trying to draw from both ends of a connection then insert an extra grid line at one of the conflicts and try again. Figure D.7 illustrates a conflict.*

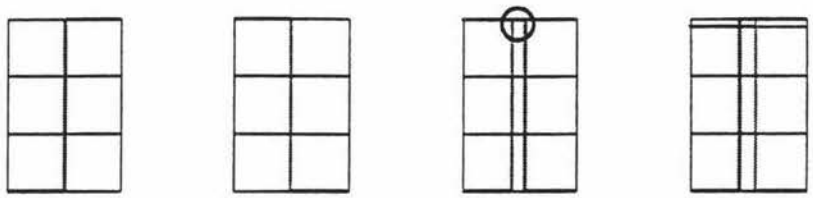


Figure D.7 - How to fix the problem of two shared horizontal, or vertical, grid lines.

If an intersection point that is required for placement is already used, figure D.8, follow the same rules as set out in the previous diagram.

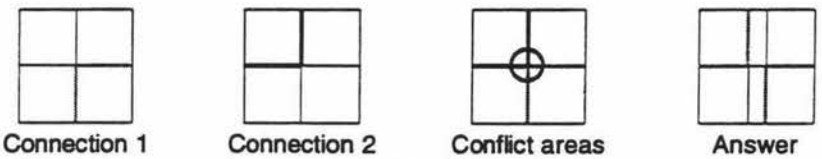


Figure D.8 - How to fix the problem of shared corners between two connections.

Appendix E

C++ Class Specification Grammar

<class_specifier> ::= <class_head> { <member_list> }
<class_head> ::= <class_key> <optional_identifier> <optional_base_spec> |
<class_key> <identifier> <optional_base_spec>
<class_key> ::= class

<optional_identifier> ::= <identifier> | ε
<identifier> ::= <character> <charNum>
<charNum> ::= <character> <charNum> | <number> <charNum> | _ <charNum> | ε
<character> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q |
r | s | t | u | v | w | x | y | z | A | B | C | D | E | F | G |
H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V |
W | X | Y | Z
<number> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0

<optional_base_spec> ::= <base_spec> | ε
<base_spec> ::= : <base_list>
<base_list> ::= <base_specifier> | <base_list> , <base_specifier>
<base_specifier> ::= <complete_class_name> |
virtual <opt_access_specifier> <complete_class_name> |
<access_specifier> <optional_virtual> <complete_class_name>

<complete_class_name> ::= <qualified_class_name>
:: <qualified_class_name>
<qualified_class_name> ::= <identifier> | <identifier> :: <qualified_class_name>
<optional_virtual> ::= virtual | ε

<opt_access_specifier> ::= <access_specifier> | ε
<access_specifier> ::= private | protected | public

(NB: member_list is the actual workings of the method and attribute interpretation sub-sections so it is not defined in the above specification.)

Appendix F

Lexical Analysis

Lexical analysis is the process in which a stream of characters is read in from left to right from a source file. These characters are grouped into tokens which are sequences of characters that have a collective meaning. (Aho, Sethi and Ullman 1988).

This section describes the front end retrieval system for the template and C++ parsing elements of the CASE tool. The parser uses the tokens supplied by the lexical analyser to generate information for the rest of the CASE tool. Figure F.1 illustrates the relationships involved.

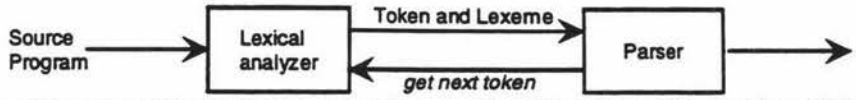


Figure F.1 - Interaction of lexical analyser with parser (modified from Aho, Sethi and Ullman 1988)

The lexical analyser produces two different types of information. The forementioned tokens and lexemes. Lexemes are the sequence of characters that a token represents. Take for example the line "If x = 15 then". If this is converted into lexemes and token then the result will be that shown in figure F.2.

The Text being parsed	Token	Lexeme
if	IF	if
x	IDENTIFIER	x
=	=	=
15	NUMBER	15
then	THEN	then

Figure F.2 - The differences between tokens and lexemes.

Using both lexemes and tokens will allow the parser to provide the raw data to use but also easily identify what has been received. It would be possible to use only the lexemes to identify what information has been received but this would take longer to generate the same results.

Tokens are stored as an enumerated type which is a method of assigning integer values to specific names. If, for example, the "days" enumerated type is made up of "sun, mon, tues, weds, thur, fri, sat" then each name will be assigned a value: sun=0, mon=1, tues=2, weds=3, etc.

When assignments or comparisons are made then the name can be used instead of the value, ie. "day = mon;" or "if (day == sun)". This is not only easier to handle but significantly reduces the time taken for comparisons.

Lexemes on the other hand are stored in the form of strings. If these were passed around the various methods for comparison, then each time the string will have to be stepped along and each character compared. This would be grossly inefficient when compared to enumerated types.

Thus why have lexemes? Not every piece of information has to be viewed in an abstract format. When building up expression trees, described in the next subsection, the actual value of the numerical information has to be used. If tokens alone were the only source of information then the data could be identified as numerical but its true value would never be known.

The lexical analyser is built into the Parser class. The overall class hierarchy is shown in figure F.3.

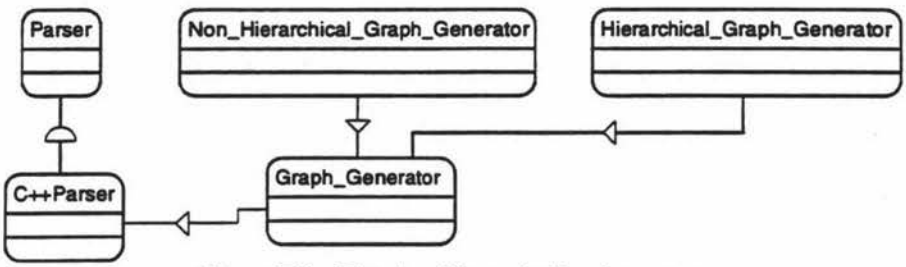


Figure F.3 - The class hierarchy for the parser.

F.1 Character Retrieval

This is the lowest form of data acquisition. The character retrieval method gets the next character from the character stream. Provision is also made for a look ahead of one character. This allows the token retrieval method to identify between various tokens, such as the division sign, "/", and the start of a comment, "//".

As well as retrieval of characters this method also keeps track of the current position in the file being read. It shows the line number and the position offset into that line. For example, the current character may be on line five offset eight.

F.2 Token Retrieval

There are two methods for acquiring the token information and error checking. The first involves asking for the next token and doing a comparison to see if this was the token expected. If not then an error message is printed. This method works but is relatively inefficient. There is a requirement to have a statement for comparing the token and one for producing the error method if unsuccessful.

The other method involves passing a list of tokens that the parser is willing to except to the token retrieval method. If the token just parsed is not amongst this group then the token retrieval method will call the error method and halt the parsing. This method returns the actual token received.

This way if there is more than one token that can be expected further comparison can be made in the method that called for token retrieval. All later comparisons no longer have to look for any errors since that problem has been circumvented.

Numerical data when being stored in a lexeme is converted and stored in a numerical format. This facilitates ease of use since the data would undoubtedly have to be converted at some stage. All numerical information is stored as a floating point or real number.

Treating the data this way simplifies storage problems. There are no side affects because a number, such as "1", acts the same way in its floating point interpretation, "1.0".

F.2.1 Token Recognition

F.2.1.1 Template Definition Language

To increase the speed of processing the raw text into tokens only a partial comparison of the parsed text was facilitated. Even though the template definition file is not meant to be hand written this trait was thought to add a certain robustness to the parsing.

An example of this feature can be shown for a language that only has two tokens starting with the letter "f", "fit" and "fat". The token retrieval method would identify an incoming word started with the letter "f".

The next stage would only involve comparing the second letter of the word with either "i" or "a" to differentiate between the tokens. It would be possible for the lexical analyser to then identify words like "fii", "fil" and "fiY" as being the word "fit".

F.2.1.2 C++ Language

C++ presented a different problem than the template definition language. It is more specific in the recognition of keywords where even the case sensitivity is important. To facilitate the recognition of such a strictly formalised language nothing short of a full comparison would suffice.

Not all C++ keywords are tokenised. Instead only a small proportion that can be explicitly used to define a class, its attributes and its methods were made available. The outcome was a lexical analyser that only identified information of relevance.

F.2.2 Comments

When processing information the token retrieval methods first comparison checks to see if the next data segment is a comment. Comments can be in one of the following formats:

- A string starting with the characters "//" and delimited with a carriage return. This type of comment is acceptable to both the template definition language and C++. Comments can be placed at any position in the data file since a definition can span as many lines as necessary.
- A string starting with a "/*" and delimited by a "*/". This is a 'C' style comment and is only acceptable by C++. This type of comment can be positioned in the same manner as the previous comment.

The parser of either the template language or C++ has no need for testing if a comment is present. The token retrieval method extracts the comment before any token identification has been initiated. In the template language parser this information is just ignored since it serves no purpose.

The C++ parser on the other hand needs this information when it is in the context of a class or method declaration. To facilitate the extract of important comments the lexical analyser buffers the last comment. When assembling class or method information the parser can utilise the comment information if necessary.

F.2.3 Token Look Ahead

To facilitate the identification of which command path to follow, token look ahead is used. This is the buffering of one token, and the related data, ahead of the one currently being used. To accomplish this the token, lexeme, line and offset information is buffered one cycle ahead of that processed.

This facility is not always needed but there are exceptions that require this service. An example of the use for look ahead can be shown in the assembling of the component parts of an expression. The expression retrieval only wants the token that it specifically needs.

If the next token was retrieved and found that it need not constitute a part of an expression then there would be no use for it. There would also be no means in which the parser could place the information back. The result would be a missing token from the file.

F.3 Expression Retrieval

As mentioned earlier and now repeated for clarification an expression consists of four parts (figure F.4). These are:

- Single numerical values
- Two expressions combined together with one of "*\-+".
- A text function to find the height or width of a text segment.
- An expression function that performs the specified function on one or more expressions.

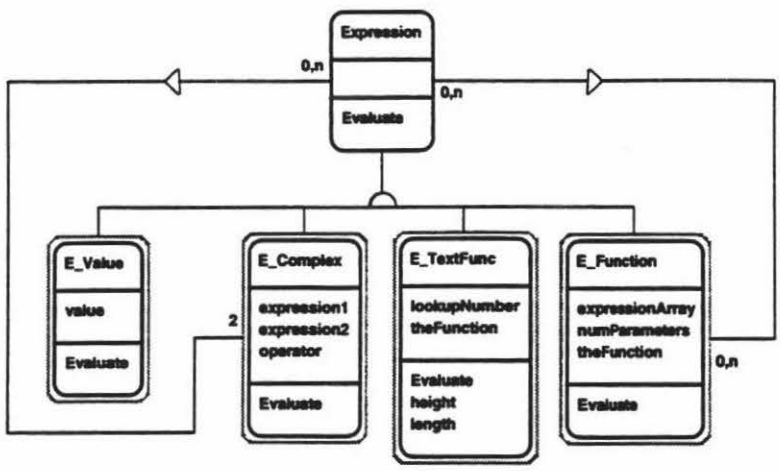


Figure F.4 MOOT - Different types of expressions

The expression retrieval method assembles all the given expressions into an expression tree structure. An expression is parsed in, one token at a time, using the token retrieval method. To built the expression tree two stacks are used to accumulate the individual parts of the expression.

The first stack is for value, functions that will later provide a value or a pointer to a partially build expression tree. The second stack accommodates all the mathematical operators. The logic behind the expression tree structure is as follows:

```
while the next token is a valid part of an expression then
    Get token
    If token is a number, equation reference, expression function or text function then
        Make the appropriate expression class and push onto the value stack.
    else
        If token is a right parenthesis then
            While the top item on the operator stack is not a left parenthesis
                Build a complex expression using the value and operator stacks
        else
            If token is a "+" or "-" and ( ( the top item on the value stack is a "*" or "/" ) or
              ( the top item on the value stack is the same as token and not equal to a ")" ) then
                Build a complex expression using the value and operator stacks
            Place token onto the operator stack.
While the operator stack is not empty
    Build a complex expression using the value and operator stacks
```

An example of an expression being transformed into an expression tree is shown in figure F.5. For this example the expression "1.3 * EQ(1) - 4 * (max(1,4) / 3 + 1)" is used.

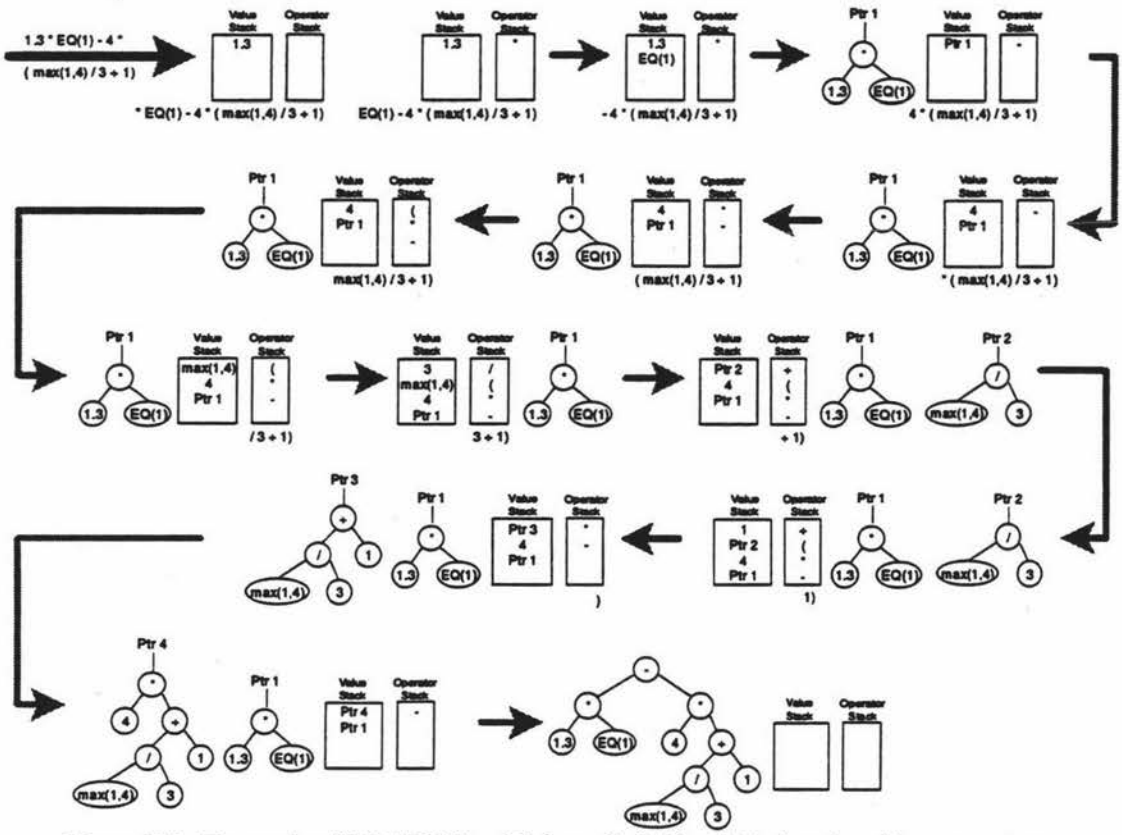


Figure F.5 - Expression "1.3 * EQ(1) - 4 * (max(1,4) / 3 + 1) being placed in a parse tree.

F.4 Error Handling

At this stage in the prototype development an error will cause the CASE tool to exit. In the future, facilities to handle errors, without exiting, will be added. This will place a dialog box on the screen and halt all parsing and lexical analysis activities.

The current error facilities encompass more than just exiting the program. Information is presented that will help to pinpoint the error within either the template definition or C++ file. Four different types of information are given. These are:

- What token was found.
- What tokens were expected.
- The line number the error occurred on.
- The offset, measured in characters, from the left hand side of the page.

All this information was given because it was expected that not all of the information may be correct at any one time. This does not mean that the program is producing the wrong information. Instead the individual characteristics that have caused the error may be misleading.

For example if the following line:

Text at 10,10 Width EQ(1) MaxX 0 MinX 0 Style 'N' Format 'L' ID 666

Is accidentally presented as:

List at 10,10 Width EQ(1) MaxX 0 MinX 0 Style 'N' Format 'L' ID 666

Then the character offset (which will be at the 'Width' option), the token found (WIDTH) and the tokens expected (TO) will not accurately specify the error. Instead the line number will be the only indication of where the error has occurred.

To accomplish error handling using positional information, accurate data has been kept for every character and token acquisitioned. Since one token look ahead is possible, twice as much information needs to be kept about the token position.

When a new token is started the lexical analyser finds out the line and column of its first characters. This is calculated by the number of carriage returns and characters scanned. When another method requests a token from the lexical analyser the token present is checked for correctness.

If the information is incorrect then an error message is passed to the error method. This then displays the positional information along with the error message passed. The error message created by the lexical analyser shows which tokens were expected and which were found.

References

AHO A.V., SETHI R. and ULLMAN J.D., (1986) *Compilers: Principles, Techniques and Tools.* Addison - Wesley Publishing Company: Reading, Massachusetts.

ALDERSON A., (1991) *Meta-CASE Technology.* Software Development Environments and CASE Technology, European Symposium, Springer-Verlag: Berlin.

ANDERSON R., KAHAN S. and SCHLAG M., (1990) *An O(n log n) Algorithm for 1-D Tile Compaction.* Graph - Theoretic Concepts in Computer Science, 15th International Workshop WG'89, Edited by Nagl M., Springer-Verlag: Germany.

ARNOLD R.S. (1994) *Software Reengineering: A Quick History.* Communications of the ACM, May 1994, Vol. 37, Number 5, pp 13 - 14.

BACHMAN C., (1988) *A CASE for Reverse Engineering.* Datamation, July 1988, Vol. 34, No. 13, pp 49 - 56.

BEYNON-DAVIES P., (2nd Ed), (1993) *Information Systems Development: An Introduction to Information Systems Engineering.* The MacMillan Press Ltd: London.

BIGGERSTAFF T.J., BHARAT G., MITBANDER G. and WEBSTER D.E., (1994) *Program Understanding and the Concept Assignment Problem.* Communications of the ACM, May 1994, Vol. 37, Number 5, pp 72 - 82.

BOLOIX G., SORENSON P.G. and TREMBLAY J.P., (1992) *Transformations using a meta-system approach to software development.* Software Engineering Journal, November 1992, pp 425 - 437.

BOLOIX G., SORENSON P.G. and TREMBLAY J.P., (1993) *Software Metrics Using a Metasystem Approach to Software Specification.* Journal of Systems Software 1993, pp 273 - 294.

BOOCH G., (1991) *Object Oriented Design with Applications.* Benjamin / Cummings Publishing Company: Redwood City, California.

BOOCH G., (1994) *Object Oriented Analysis and Design with Applications.* 2nd Ed. Benjamin / Cummings Publishing Company: Redwood City, California.

BRANDENBURG F.J., (1990) *On the Complexity of Optimal Drawings of Graphs.* Graph - Theoretic Concepts in Computer Science, 15th International Workshop WG'89, Edited by Nagl M., Springer-Verlag: Germany.

BROUGH M., (1992) *Methods for CASE: a Generic Framework.* Advanced Information Systems Engineering: Third International Conference CAiSE'92, Edited by Loucopoulos P., Springer-Verlag: Berlin.

BRUMBAUGH D.E., (1994) *Object-Oriented Development: Building CASE Tools with C++.* John Wiley & Sons: New York.

BUDD T., (1991) An Introduction to Object Oriented Programming. Addison - Wesley Publishing Company: Reading, Massachusetts.

BUDGEN D. and FRIEL G., (1992) *Augmenting the Design Process: Transformations from Abstract Design Representations.* Advanced Information Systems Engineering: Third International Conference CAiSE'92, Edited by Loucopoulos P., Springer-Verlag: Berlin.

CARPANO M., (1980) *Automatic Display of Hierarchized Graphs for Computer - Aided Decision Analysis.* IEEE Transactions on Systems, Man and Cybernetics, November 1980, Vol 10, No. 11, pp 705 - 715.

CATTELL R.G.G., (1991) *Object Data Management: Object - Oriented and Extended Relational Database Systems.* Addison - Wesley Publishing Company: Reading Massachusetts.

CHARTRAND G. and OELLERMANN O.R., (1993) *Applied and Algorithmic Graph Theory.* McGraw - Hill: New York.

CHIKOFSKY E.J. and CROSS II J.H., (1990) *Reverse Engineering and Design Recovery: A Taxonomy.* IEEE Software, January 1990, pp 13 - 17.

CHOI S.C. and SCACCHI W., (1990) *Extracting and Restructuring the Design of Large Systems.* IEEE Software, January 1990, pp 66 - 71.

CLARK J. and HOLTON D.A., (1991) *A first look at Graph Theory.* World Scientific: Singapore.

COAD P. and NICOLA J., (1993) *Object Oriented Programming.* Prentice - Hall : Englewood Cliffs, New Jersey.

COAD P. and YOURDON E., (1991a) *Object - Oriented Analysis.* 2nd Ed. Prentice - Hall International: Englewood Cliffs, New Jersey.

COAD P. and YOURDON E., (1991b) *Object - Oriented Design.* Prentice-Hall International: Englewood Cliffs, New Jersey.

COLEMAN D., (1994) *Object-Oriented Development : The Fusion Method.* Prentice - Hall: Englewood Cliffs, New Jersey.

COX B.J., (1986) *Object Oriented Programming: An Evolutionary Approach.* Addison - Wesley Pbulishing Company: Reading Massachusetts.

CURTIS B., (1992) *The CASE for Process.* The Impact if Computer Supported Technologies on Information Systems Development, Edited by Kendall K.E., DeGross J.I. and Lyytinen K. North-Holland: Amsterdam.

DATE C.J., (1990) *An Introduction to Database Systems, Volume I.* 5th Ed. Addison - Wesley Publishing Company: Reading, Massachusetts.

DEDOUREK J.M., SORENSON P.G. and TREMBLAY J.P., (1989) *Meta Systems for Information Processing System Specification Environments.* INFOR, August 1989, Vol 27, No 3, pp 311 - 337.

DEMETROVICS J., KNUTH E. and RADÓ P. (1982) *Specification Meta Systems.* Computer, May 1982, pp 29 - 35.

DI BATTISTA G. and TAMASSIA R., (1988) *Upward Drawings of Acyclic Digraphs.* Graph - Theoretic Concepts in Computer Science, International Workshop WG'87, Edited by Göttler H. and Schneider H.J., Springer-Verlag: Germany.

EVEN S., (1979) *Graph Algorithms.* Computer Science Press: Maryland.

FINKELSTEIN C., (1990) *An Introduction to Information Engineering: From Strategic Planning to Information Systems.* Addison-Wesley Publishing Company: Sydney.

FIRESMITH D.G., (1993) *Object Oriented Requirements Analysis and Logical Design: A Software Engineering Approach.* John Wiley & Sons: New York.

FIRST CLASS, The Object Management Group Newsletter. March / April 1994, Volume 4, Issue 2.

GADWAL D., FINDEISEN P.S., SORENSON P.G., TREMBLAY J.P. and MILLAR L.B., (1994) *Generating Customizable Software Specification Environments Using Metaview.* Technical Report, Department of Computational Science: Saskatoon, Saskatchewan, Canada.

GIBBONS A., (1985) *Algorithmic Graph Theory.* Cambridge University Press: Cambridge.

GIBSON M.L., (1988) *A Guide to Selecting CASE Tools.* Datamation, July 1988, Vol. 34, No. 13, pp 65 - 66.

GONDRAN M., MINOUX M. and VAJDA S., (1979) *Graphs and Algorithms.* John Wiley & Sons: Chichester.

GOOR G.V.D., HONG S. and BRINKKEMPER S., (1992) *A Comparison of Six Object-Oriented Analysis and Design Methods.* School Report, University of Twente: Netherlands.

GRIFFIN J. (1994) Personal communication. Lecturer, Department of Mathematics, Massey University: Palmerston North.

HAINE P., (1992) *Second Generation CASE: Can it be Justified.* CASE: Current Practice, Future Prospects, Edited by Spurr K. and Layzell P., John Wiley & Sons: Chichester.

HAM J. (1994) Template Generator for a Methdology Independent Object-Oriented CASE Tool. Massey University Honors Report.

HILLE R.F., (1988) *Data Abstraction and Program Development using Pascal.* Prentice Hall: New York.

HUDSON J. (1994) Personal communication. Senior Lecturer, Department of Computer Science, Massey University: Palmerston North.

JACOBSON I., CHRISTERSON M., JONSSON P. and ÖVERGAARD G., (1994) *Object Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley Publishing Company: Wokingham, England.

JARZABEK S. and TAN C.L., (1994) *Modelling multiple views of common features in software reengineering for reuse.* Advanced Information Systems Engineering, CAiSE'94. Springer - Verlag: Berlin.

JAYAKUMAR R., THULASIRAMAN K. and SWAMY M.N.S., (1989) *$O(n^2)$ Algorithms for Graph Planarization.* Graph - Theoretic Concepts in Computer Science, 15th International Workshop WG'88, Edited by van Leeuwen J., Springer-Verlag: Germany.

JOHNSON M. (1994) Personal communication. Phd Student, Department of Mathematics, Massey University: Palmerston North.

KOPACHE M.E. and GLINERT E.P., (1988) *C^2: A Mixed Textual / Graphical Environment for C.* IEEE Proceedings Workshop on Visual Languages. IEEE Press.

LANG B., (1991) *CASE Support for the Software Process: Advances and Problems.* 3rd European Software Engineering Conference, ESEC'91, Springer-Verlag: Berlin.

LENGAUER T., (1990) *Combinatorial Algorithms for Integrated Circuit Layout.* John Wiley & Sons: Chichester.

LEWIS T., (1990) *Code Generators.* IEEE Software, May 1990, pp 67 - 70.

LORENZ M., (1993) *Object Oriented Development: A Practical Guide.* Prentice - Hall : Englewood Cliffs, New Jersey.

LORENZ M. and KIDD J., (1994). *Object Oriented Software Metrics.* Prentice - Hall : Englewood Cliffs New Jersey.

MARKOSIAN L., NEWCOMB P., BRAND R., BURSON S. and KITZMILLER T., (1994) *Using an Enabling Technology to Reengineer Legacy Systems.* Communications of the ACM, May 1994, Vol. 37, Number 5, pp 58 - 70.

MARTIN J. and ODELL J.J., (1992) *Object - Oriented Analysis and Design.* Prentice - Hall : Englewood Cliffs, New Jersey.

MARTIN J., (1993) *Principles of Object - Oriented Analysis and Design.* Prentice - Hall : Englewood Cliffs, New Jersey.

MARTTIIN P., (1994) *Towards Flexible Process Support with a CASE Shell.* Advanced Information Systems Engineering, CAiSE'94. Springer - Verlag: Berlin.

MEHANDJISKA-STAVREVA D., PAGE D. and CLARK P., (1994) *Development of an Intelligent Object-Oriented CASE Tool.* Object Oriented Information Systems Conference, OOIS'94.

MEYER B., (1994) *Reusable Software: The Base Object-Oriented Component Libraries.* Prentice - Hall : Englewood Cliffs, New Jersey.

MILLER L., (1990) *Programming by Components.* CommUNIXations, May 1990, pp 14 - 19

NILSSON E.G., (1990) *CASE Tools and Software Factories.* Advanced Information Systems Engineering: Third International Conference CAiSE'90, Edited by Steinholtz B., Solvberg A. and Bergman L., Springer-Verlag: Berlin.

NISHIZEKI T. and CHIBA N., (1988) *Planar Graphs: Theory and Algorithms.* North-Holland: Amsterdam.

PAGE D., CLARK P. and MEHANDJISKA-STAVREVA D., (1994) *An Abstract Definition of Graphical Notations for Object Orientated Information Systems.* Object Oriented Information Systems Conference, OOIS'94.

PAPAHRISTOS S. and GRAY W.A. (1991) *Federated CASE Environment.* Advanced Information Systems Engineering: Third International Conference CAiSE'91, Edited by Anderson R., Bubenko jr J.A. and Solvberg A., Springer-Verlag: Berlin

PAULISCH F.N. (1993) *The Design of an Extendible Graph Editor.* Springer-Verlag Berlin.

POCOCK J.N., (1991), *VSF and its Relationship to Open Systems and Standard Repositories.* Software Development Environments and CASE Technology, European Symposium, Springer-Verlag: Berlin.

POWERS M.J., ADAMS D.R. and MILLS H.D., (1984) *Computer Information Systems Development: Analysis and Design.* South-Western Publishing Company: Cincinnati.

PREECE J., BENYON D., DAVIES G., KELLER L. and ROGERS Y. (eds) (1993) *A Guide to Useability: Human Factors in Computing.* Addison-Wesley: Wokingham, England.

PRESSMAN R.S., (3rd Ed) (1992) *Software Engineering: A Practitioner's Approach.* McGraw-Hill Inc: New York.

QUILICI A., (1994) *A Memory-Based Approach to Recognising Programming Plans.* Communications of the ACM, May 1994, Vol. 37, Number 5, pp 84 - 93.

REICH G. and WIDMAYER P., (1990) *Beyond Stiener's problem: A VLSI oriented generalization.* Graph - Theoretic Concepts in Computer Science, 15th International Workshop WG'89, Edited by Nagl M., Springer-Verlag: Germany.

REINGOLD E.M. and TILFORD J.S., (1981) *Tidier Drawings of Trees.* IEEE Transactions on Software Engineering, March 1981, Vol. 7, No. 2.

RICH C. and WILLS L.M., (1990), *Recognizing Program's Design: A Graph-Parsing Approach.* IEEE Software, January 1990, pp 82 - 89.

ROSSI M., GUSTAFSSON M., SMOLANDER K., JOHANSSON L. and LYYTINEN K., (1992) *MetaModelling Editor as a Front End Tool for a CASE Shell.* Advanced Information Systems Engineering: Third International Conference CAiSE'92, Edited by Loucopoulos P., Springer-Verlag: Berlin.

ROTHWELL D.M., (1993) *Databases: An Introduction.* McGraw - Hill Book Company: London.

RUMBAUGH J., BLAHA M., PREMERLANI W., EDDY F. and LORESON W., (1991) *Object Oriented Modeling and Design.* Prentice Hall: Englewood Cliffs, New Jersey.

SENN J.A., (1990), *Information Systems Management.* Wadsworth Publishing Company: Belmont California.

SMOLANDER K., LYYTINEN K., TAHVANAINEN V., MARTTIIN P., (1991), *MetaEdit - A Flexible Graphical Environment for Methodology Modelling.* Advanced Information Systems Engineering: Third International Conference CAiSE'91, Edited by Anderson R., Bubenko jr J.A. and Solvberg A., Springer-Verlag: Berlin.

SOMMERVILLE I., (1992) *Software Engineering.* Addison - Wesley Publishing Company: Reading Massachusetts.

SORENSON P.G., (1988) *First Generation Case Tools: All Form But Little Substance?.* Technical Report, Department of Computational Science, Saskatoon, Saskatchewan, Canada.

SORENSON P.G., TREMBLAY J.P and McALLISTER A., (1988) *The Metaview System for Many Specification Environments.* IEEE Software, March 1988, Vol 5. No 2, pp 30 - 38.

STROUSTRUP B., (1991) The C++ Programming Language. (2nd Ed), Addison - Wesley Publishing Company: Reading Massachusetts.

SUGIYAMA K., TAGAWA S. and TODA M., (1981) *Methods for Visual Understanding of Hierarchical System Structures.* IEEE Transactions on Systems, Man and Cybernetics, February 1981, Vol 11, Number 1, pp 109 - 125

SUMNER M., (1992) *The Impact of Computer Assisted Software Engineering on Systems Development.* The Impact if Computer Supported Technologies on Information Systems Development, Edited by Kendall K.E., DeGross J.I. and Lyytinen K. North-Holland: Amsterdam.

SWAMY M.N.S. and THULASIRAMAN K., (1981) *Graphs, Networks and Algorithms.* John Wiley & Sons: New York.

TAMASSIA R., DI BATTISTA G. and BATINI C. (1988) *Automatic Graph Drawing and Readability of Diagrams.* IEEE Transactions on Systems, Man and Cybernetics, January/February 1988, Vol 18, Number 1, pp 61 - 79.

TERRY P.D., (1986) *Programming Language Translation: A Practical Approach.* Addison - Wesley Publishing Company: Wokingham, England.

TROTTER W.T., (Ed), (1993) *Planar Graphs.* American Mathematical Society.

TRULLEMANS C., (Ed), (1986) *Algorithms for VLSI.* Academic Press: London.

VAN VLIET H., (1993) *Software Engineering : Principles and Practice.* John Wiley & Sons : New York.

VAUCHER J.G. (1980) *Pretty - Printing of Trees. Software - Practice and Experience.* Vol 10, pp 553 - 561.

VERHOEF T.F., TER HOFSTEDE A.H.M., WIJERS G.M., (1991) *Structuring modelling knowledge for CASE shells.* Advanced Information Systems Engineering: Third International Conference CAiSE'91, Edited by Anderson R., Bubenko jr J.A. and Solvberg A., Springer-Verlag: Berlin.

WALDIN E. and RICKETTS H., (1994) *Tailored CASE: A case study in meta-CASE.* Objects in Europe, Autumn 1994, Vol 1, No 4, pp 29 - 30.

WARFIELD J.N., (1977) *Crossing Theory and Hierarchy Mapping.* IEEE Transactions on Systems, Man, and Cybernetics, July 1977, Vol smc-7, No 7, pp 505 - 523.

WATERS R.C. and CHIKOFSKY E., (1994) *Reverse Engineering: Progress Along Many Dimensions.* Communications of the ACM, May 1994, Vol. 37, Number 5, pp 23 - 24.

WEISS M.A., (1992) *Data Structures and Algorithm Analysis.* Benjamin / Cummings Publishing Company: Redwood City, California.

WETHERELL C. and SHANNON A., (1979) *Tidy Drawings of Trees.* IEEE Transactions on Software Engineering, September 1979, Vol. 5, No. 5.

WIRFS-BROCK R., WILKERSON B. and WIENER L., (1990) *Designing Object - Oriented Software.* Prentice Hall: Englewood Cliffs, New Jersey.

YOURDON E., (1993) Decline and Fall of the American Programmer. Yourdon Press, Prentice - Hall : Englewood Cliffs, New Jersey.