OBJECT-ORIENTED SOFTWARE DEVELOPMENT IN STRUCTURAL ENGINEERING

by

KEVIN MICHAEL ELBURY

B.A.Sc. (Civil), The University of British Columbia, 1990

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

MASTER OF APPLIED SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

Department of Civil Engineering

We accept this thesis as conforming
to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA
APRIL, 1992

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

(Signature) _____

Department of CIVIL ENGINEERING

The University of British Columbia
Vancouver, Canada

Date APRIL 30, 1992

Abstract

The recent emergence of the object-oriented paradigm has created a very powerful methodology to aid software developers in the creation of complex applications. This technology is quite common in fields such as computer science and computer engineering but still remains relatively unexplored in more traditional disciplines such as Civil Engineering. The paradigm enforces several basic necessities required by complex, modern software applications. These include management of complexity, data modelling, information hiding, software reusability, and software evolution.

The purpose of this thesis is to give an overview of the object-oriented paradigm. This discussion includes a review of the necessary requirements of an object-oriented language. This is followed by the presentation of a software diagramming notation which can aid in the data modelling and design of a software system before coding is started. Also presented is a discussion on the pragmatics of object-oriented development.

A universal structural analysis preprocessor called "Cross Link" is developed by the author to demonstrate the application of the object paradigm. Cross Link is intended to provide a unified, easy to use, graphical preprocessing environment that can be used as a front end for any type of finite element analysis programme or CADD package. This is achieved through the implementation of a powerful macro programming language which allows users to manipulate the finite element database in many different ways.

Table of Contents

Table of Figures

Acknowledgements

I would like to take this opportunity to thank the many people who have helped me in completing this thesis under the Professional Partnership Programme. I extend my gratitude to Mr. Bill Kendrick, P.Eng., Chief Engineer at Canron Inc., and Mr. Dave Halliday, P.Eng., Special Projects Manager at Coast Steel Fabricators Ltd. I also thank Mr. David Lo, P.Eng., for his contribution to this thesis, Mr. Manfred Frank, P.Eng., and Mr. Phil Sullivan, P.Eng., for their practical perspectives on the steel fabrication business. Finally, I thank Dr. Siegfried F. Stiemer and Dr. Helmut P. Prion, my graduate advisors, for the valuable support and guidance they provided for me.

Kevin M. Elbury

1 INTRODUCTION

Software development often involves the process of mapping or decomposing a complex problem space into modules that are ordered and simplified. The success of this mapping is often determined by the methodology used to analyze and design the "building blocks" that will solve the problem. Recent advances in development tools that support the object-oriented paradigm are now giving the software developers the ability to create complex software applications that excel over traditional techniques in modelling a problem space. As engineers, we naturally deal with complexity through the classification and decomposition of concepts into hierarchial or tree-like formations. The object-oriented development process follows in a manner very similar to this so it may be argued that the paradigm is well suited as a development tool for complex engineering applications.

Early observations also indicate that object-oriented software is far more economical to produce than software developed using traditional procedural approaches such as Structured Design. This is mostly due to the large amount of code that can be reused both inside and across applications. Object-oriented software also tends to be much more adaptable and easier to maintain as programme requirements change because of features such as encapsulation, inheritance and information hiding. The purpose of this thesis is, firstly, to provide an overview of object-oriented principles in a language independent manner and secondly, to explain the object-oriented development of structural engineering software application.

The second chapter of this thesis introduces the software life cycle, a process oriented procedure which most software follows as it is developed. Two different development paradigms are presented: Structured Development and Object-Oriented Development. Different types of knowledge representation are presented and compared with the two paradigms and it is concluded that the structured paradigm follows an algorithmic approach which places the data being modelled secondary to the procedure. Conversely, the object-oriented paradigm follows a more data-centered approach whereby entities are recognized as objects that have both state and behavior.

The third chapter outlines the four principles an object-oriented language must support in order to be classified as "object-oriented". These include encapsulation, classification, flexible sharing and interpretation. Also presented is an brief overview of the Booch Design Notation which

is a diagramming technique developed to aid designers in the creation of software design specifications. Finally, some practical considerations in object-oriented development are discussed, including classification and designing for reusability.

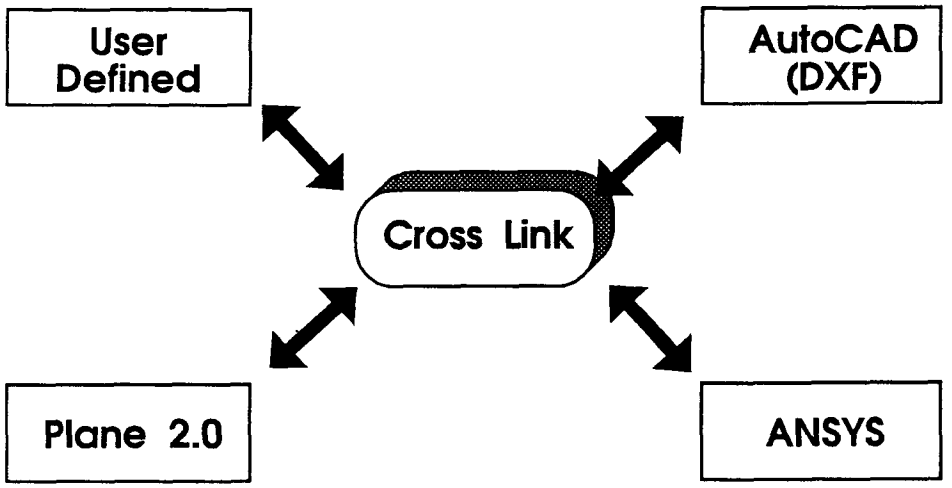


Figure 1.
Cross Link as a Universal Structural Analysis Preprocessor

The fourth and final chapter looks at the design and implementation of Cross Link, a universal structural analysis preprocessor. Developed using object-oriented techniques, the application allows structural analysts to generate finite element models for *any* type of analysis programme, design utility or CADD (Computer Aided Drafting and Design) application using an easy to use, integrated environment (Figure 1). The preprocessor supports two editing modes. The first, an interactive drawing mode, lets the analyst *draw* a structure on the display and perform all usual editing operations such as moving, resizing and setting boundary conditions on nodes and elements. The second editing mode, driven by a powerful macro programming language, allows users to write macros to manipulate the structure database in many different ways. For example, macros can be developed to enhance existing editing features, perform automated mesh generation or design parametric structures (ie. a truss with variable span and depth etc.). Most importantly though, the macro facility provides a means for a two way link (import and export) to *any* type of analysis programme, providing the analysis programme's file format is known. As an example, two macros (for import and export) are developed that bridge Cross Link with the ANSYS finite element programme.

2 THE SOFTWARE LIFE CYCLE

The *process model* used in software engineering formalizes the phases of the software life cycle to make each more visible. The most general process model is that of the *Waterfall Life Cycle,* proposed by Royce (1970) (Figure 2). Based on this model, a multitude of specialized paradigms[1] have evolved that attempt to provide better models of some of the various aspects of the software development process. However, the waterfall life cycle model remains the most widely accepted process model used in software development. The five phases of the cycle include: Analysis, Design, Coding, Testing and Integration. [18]

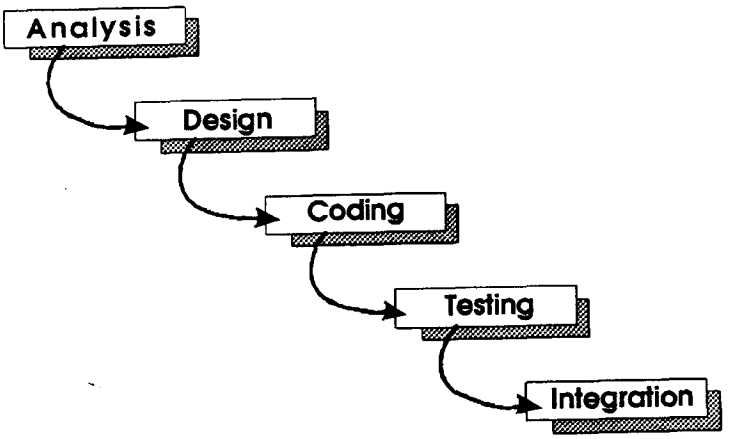


Figure 2.
The Five Phases of the Waterfall Life Cycle.

Phase 1: Analysis

The analysis phase establishes the software system services and constraints through consultation with the users and developers of the system. Generally, the goal is to "build a vocabulary of the problem domain" [2]. This involves identification or invention of tangible objects, roles, events, interactions and procedures. The result of this process is a "Software

1 Other paradigms include Exploratory Programming, Prototyping and Formal Transformations [18]

2 Booch, Grady. *Object-Oriented Design.* pg. 141 [3].

Requirements Document" that must be readable by both users and developers. Popular analysis techniques include Structured Analysis (SA), Object-Oriented Analysis (OOA), Domain Analysis and Entity-Relationship Modelling [3,6,18].

Phase 2: Design

Using the "Requirements Document", design proceeds with the specification of software components and associated functionalities. The results of design are usually presented in a way that allows for easy transformation into a computer code. Several design techniques exist; correct selection depends on the type of analysis used. The two most common are Structured Design and Object-Oriented Design.

Phase 3-5: Coding, Testing and Integration

Using the design documents, the problem is coded into the computer using a programming language suitable to the type of analysis and design performed. Testing of each module in the system is then performed according to the specifications in the Requirements Document. This phase is followed by integration whereby all modules and programme units are tested as a whole, again according to Requirements Document specifications.

2.1 Structured and Object-Oriented Development

This section introduces the two popular software development paradigms, one established and one emerging. These are the Structured Development and Object-Oriented development paradigms. Of the different software development paradigms that have evolved over the past 30 years, Structured Development (SD) has come forth as an efficient software development technique. Dale and Orshalick describe SD as (Figure 3):

> *"A design methodology that works from an abstract functional description of a problem (top) to a detailed solution (bottom); a heirarchial approach to problem solving that divides a problem into functional sub-problems represented by modules...The design consisting of a hierarchy of separate modules with lower level modules containing greater detail than higher level modules."*[3]

However, within the past 15 years, progression of the Object-Oriented (OO) development paradigm has progressed to the point that it can now be called a formal software development

3 Dale, Nell. and Orshalick, David. *Introduction to PASCAL and Structured Design*, p. A50 [7].

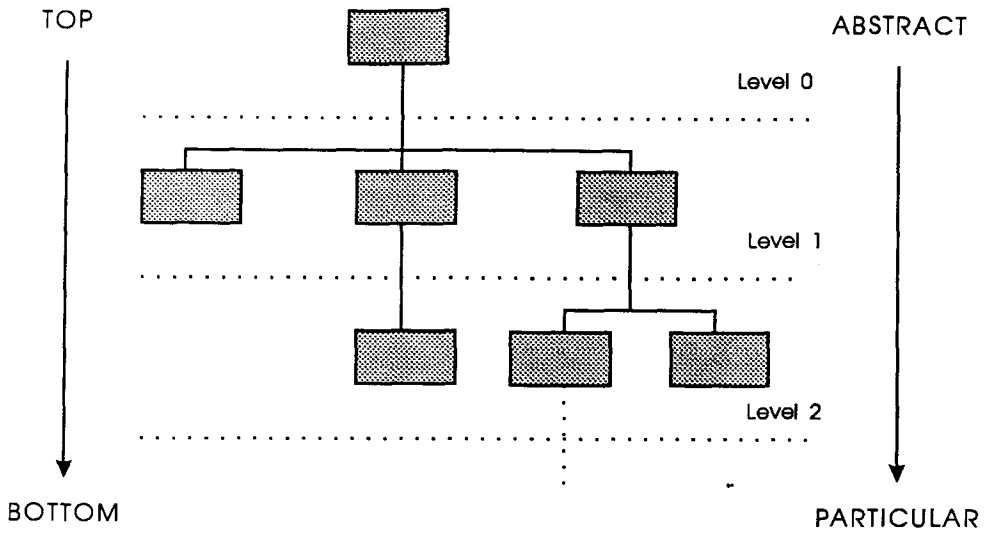


Figure 3.
Structured or Top-Down Software Development.

technique. Being a formal technique, every phase in the development life cycle fully supports the paradigm. The OO model views a system as an assembly of *objects*, each capable of enveloping its own state and behavior. Objects use *messages* to communicate with other objects. *Classes* provide a template that describes the behavior of an object and the *inheritance* mechanism allows a class to be specialized from existing classes (Figure 4). Fundamentally, the OO paradigm addresses the issues of code reusability, information hiding and software evolution.

Structured and Object-Oriented development have evolved essentially from the differing processes humans use to classify knowledge. Loy breaks this classification into two categories: *structural representation* and *functional representation* [11]. The structural paradigm organizes knowledge into entities that have both state and services with the services having secondary importance over the state (the Object Oriented paradigm). The functional paradigm, on the other hand, represents knowledge as an algorithm or procedure. Conversely, functional knowledge places the procedure primary and the data secondary (the Structured paradigm). Recognition of these differing representations of knowledge has lead to hybrid analysis and design techniques that bring together the advantages of the structural and functional approaches [8].

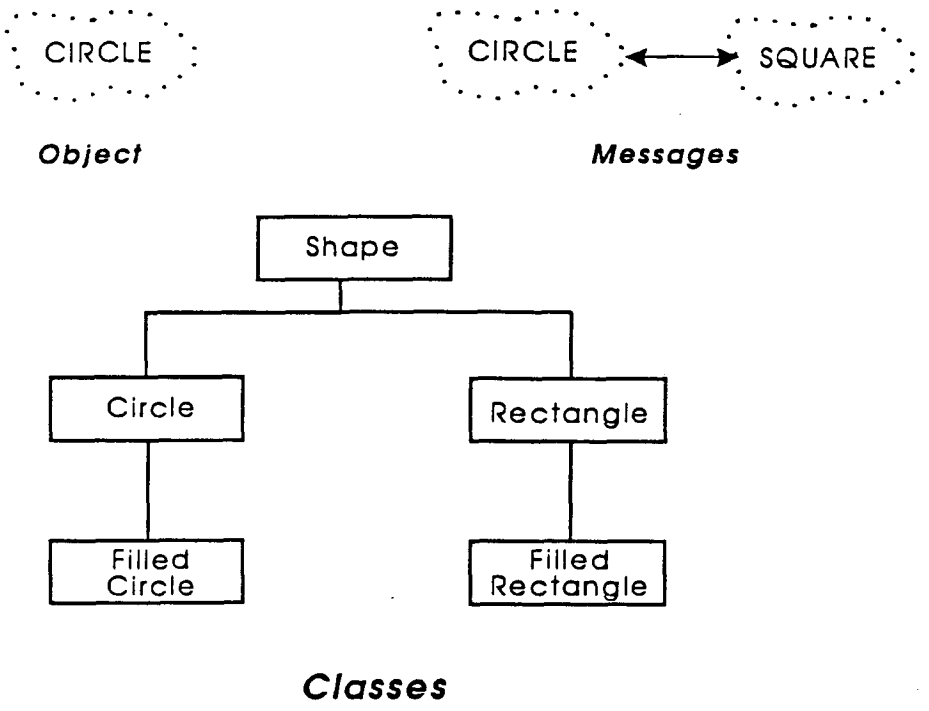


Figure 4.
The Three Concepts of Object-Oriented Programming.

It is interesting to note the process in which software development occurs using either of the two paradigms. The Structured Development approach is to "divide and conquer". High level or abstract properties of the system are identified and broken down into lower level components. Programme coding then begins with implementation of the low level components, eventually ending in coding at a high level. In this way, analysis and design usually occur top-down and implementation, bottom-up. Thus, any changes made in analysis and design are much more difficult to implement at the coding stage.

In object-oriented development, the goal is to identify the major objects or entities in the system. From these, one abstracts as many properties as possible common to all objects. These form the abstract classes which are the foundation of the system. Programme coding then begins with implementation of the abstract classes. The more specialized classes are then implemented using the inheritance mechanism which allows the behavior encapsulated in the parent class(es) to be shared with derived classes. Thus in the object-oriented paradigm, development occurs in the same

order in which the classes are abstracted. This makes the process of analysis, design and coding much more synchronized than that of Structured Development. In this way, it becomes much easier to return to any phase in the process life cycle and make modifications as more is learnt about the system.

3 BACKGROUND ON OBJECT-ORIENTED (OO) SYSTEMS

3.1 Framework of an OO System

The framework of an object-oriented (OO) system is built around a multidimensional view of the object paradigm. These dimensions are viewed as the fundamental principles of object orientation that provide the basic building blocks for dealing with complexity in software systems. The principles allow most of the object-oriented developments to date to be classified in a coherent and unified manner, thus allowing for objective evaluation of the many different techniques available. The features found in any given OO language can be classified into the following principles of object orientation:

- Encapsulation
- Classification
- Flexible Sharing
- Interpretation [2]

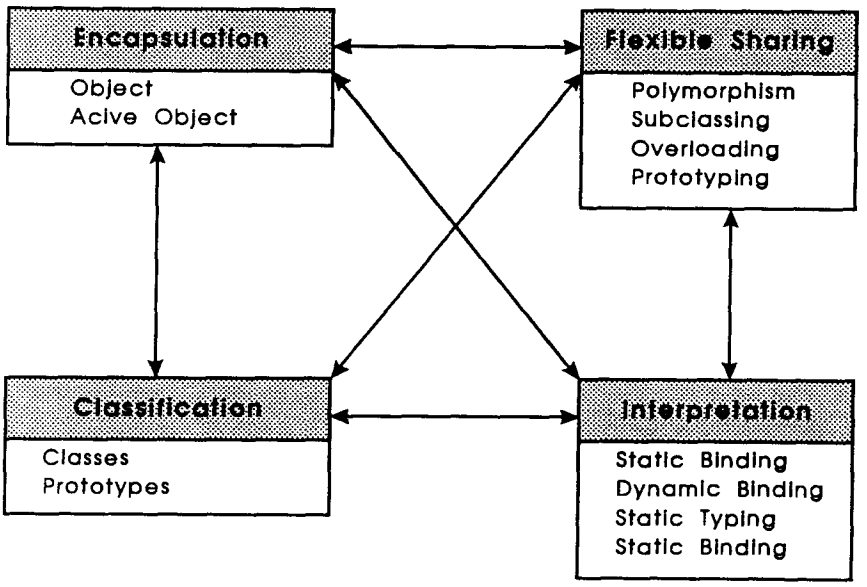


Figure 5.
The multidimensional model of an object-oriented system. Within each dimension there exists several different solution techniques.

Shown in Figure 5 is the generalized framework (principles) of an OO system. Each principle lists a series of different techniques that have been used in various languages to apply the principle. Thus, by looking at a particular technique, the framework enables a connection to be made between it and its associated principle. Following is a brief discussion of the principles of OO systems along with some of the more common associated techniques[4].

Encapsulation

Encapsulation (also referred to as information hiding) provides a method of grouping together the various properties[5] associated with an identifiable entity in the system into one logical unit (an object). Access to the object is provided via its *interface*, which defines the protocol users of the object use to communicate (ie. the object's external view). On the other hand, the *implementation* of an object is the set of properties that only the object itself knows about (ie. the internal view). Snyder describes encapsulation as:

> "*...a technique for minimizing interdependencies among separately written modules by defining strict external interfaces. The external interface of a module serves as the contract between module and clients. If clients depend only on the external interface, the module can be re-implemented without affecting any clients, as long as the implementation supports the same external interface.*" [17]

Object-Oriented languages achieve encapsulation via two techniques: passive objects and active objects.

Passive and Active Objects

In class based languages, passive objects signify the importance of structural organization of objects. In general, passive objects represent state; this state can only be changed when instructed by other objects.

In contrast to class based languages, actor languages are mainly concerned not with the structural organization of objects, but with the "communication structure of interacting

---

4 It is important to note that most object-oriented languages support any or all of the following principles. This leads to a popular question demanded of many so-called object-oriented languages: Is it really object-oriented?

5 The term *property* collectively represents the both the state (ie. *Member variables*) and operations (ie. *Member functions*) associated with an object.

processes"[6]. This is achieved via active objects. Active objects differ from passive objects in that they encompass their own thread of control. While passive objects can only undergo changes in state when explicitly acted upon, active objects can show behavior without being operated on by another object. Active objects are of direct importance in concurrent or multi-tasking computer systems and will likely become more significant as more computers gain multi-tasking capability.

Classification

Classification is a higher level approach to encapsulation. Instead of grouping together properties of an entity into an object, classification works to group together objects with common properties. Given that various combinations of classification can be formed in a given system, one must determine exactly how to classify them. Should objects be classified according to particular attributes (ie. Color, shape) or according to the operations performed by objects (ie. Read, write)? Some common methods of classification are discussed below.

Classes

A class is a template from which objects can be created. It is based on the grouping together of state descriptors and methods for the object. The inheritance mechanism allows one class to acquire, modify or extend the behavior of another class. The result of inheritance is a class hierarchy which is useful for capturing redundant behavior in a particular group of classes (Figure 6).

Class relationships are usually expressed using any one of three basic principles: generalization, aggregation or association[7]. *Generalization* denotes the "kind of" relationship amongst classes. For example, a highrise is a *kind of* building. Secondly, *aggregation* usually denotes the "part of" relationship amongst classes. Here, a storey is not considered a *kind of* building but a *part of* a highrise. Thirdly, *association* expresses some type of connotative connection amongst otherwise unrelated classes. As an example, an earthquake may govern the design of a highrise building but in no way can it be considered dependent class of building.

---

6 Blair, Gordon et. Al., *Object Oriented Languages, Systems and Applications*, p.62 [2].

7 Booch, Grady. *Object Oriented Design.* p.96 [3]

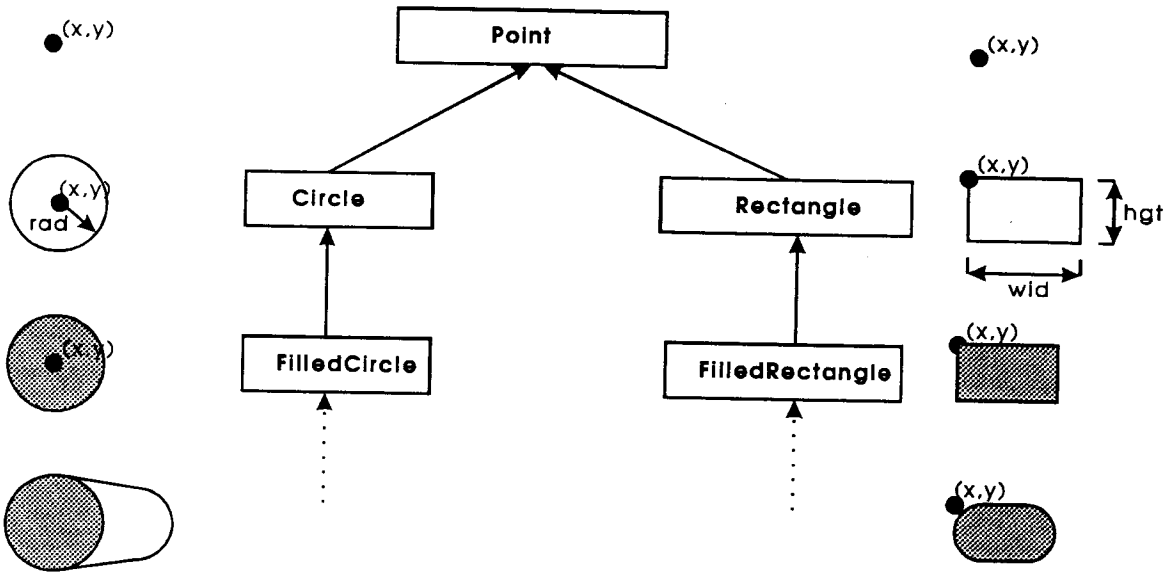


Figure 6.
Classification using classes and inheritance. Inheritance allows classes to evolve from simple to advanced.

Some consider class based systems to be the most restricted form of classification because construction of a given class hierarchy is sometimes difficult due to the many different (and correct) combinations that may be possible [2]. Despite this, class based systems (ie. C++, Smalltalk, CLOS) are currently most popular in object-oriented computing. Moreover, this style of object orientation solves some of the more pragmatic problems of software development, mainly in the areas of code reusability and program evolution.

Prototypes

Alongside classes, prototypes provide a second form of classification. A prototype represents the "default behavior for a concept, and new objects can re-use part of the knowledge stored in the prototype by saying how the new object differs from the prototype" [14]. Objects in a prototypical system use the mechanism of *delegation* to send messages to prototypes that represent general knowledge (in general, any object can serve as a prototype). Similar to class based systems, prototypes also support behavior sharing but in a fundamentally different manner.

Objects that share knowledge with a prototype are constructed using an *extension* object which contains a list of prototypes which can be shared and a *personal* definition of the object that gives it a unique identity. When an extension object receives a message, it first attempts to respond to the message using its personal part. If this is unsuccessful, it then forwards (delegates) the message to its prototypes.

Flexible Sharing

OO systems generally distinguish between two types of techniques, the first being those which support classification and the second being those which support flexible behavior sharing. The techniques discussed in this section enhance behavior sharing and evolution in an OO system.

Polymorphism

Polymorphism is one of the most characteristic features of an object-oriented system. It is defined as "the ability of behavior to have an interpretation over more than one class"[8]. Polymorphism represents a shift towards the modelling of common behaviorisms amongst a group of objects. For example, in the class hierarchy shown in Figure 7, every class exhibits a *Draw()* and *Erase()* function. Users of any one of these classes can generically send a *Draw()* or *Erase()* message to the object and be guaranteed that the it will behave appropriately. If a *Draw()* message were sent to an instance of class *FilledCircle*, the Draw() function defined for filled circle may fill the circle and then call the *Draw()* function for the *Circle* class to draw the circle's outline.

8 Blair, Gordon. et al. *Object Oriented Languages, Systems and Applications*, pp. 35,116. [2]

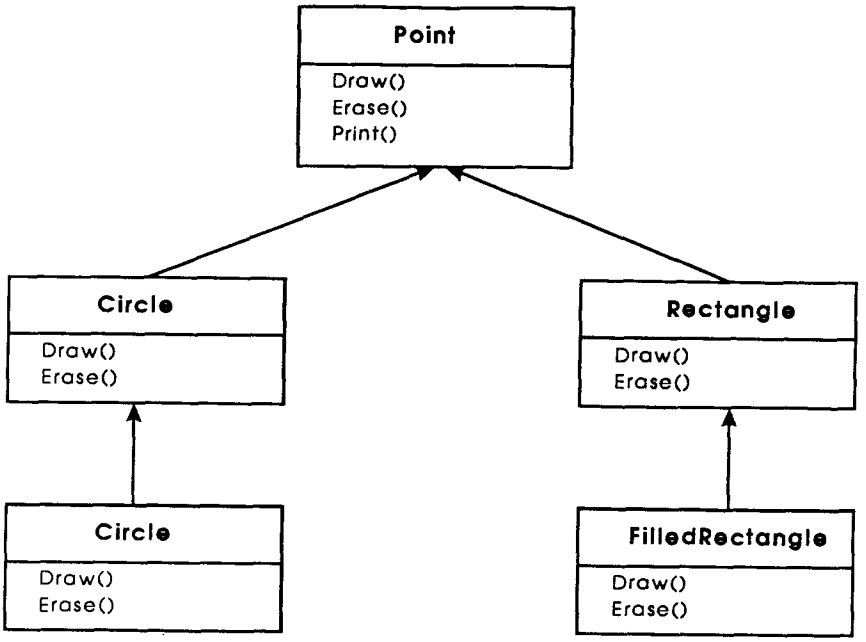


Figure 7.
A class hierarchy for the representation of some geometric shapes. Sending the *Draw()* message to an instance of any of these classes will elicit a uniform response.

Subclassing (Inheritance)

Subclassing or inheritance incorporates the behavior of one class into another. The new class is called the *subclass* of the parent or *superclass*. Through the process of subclassing, a specialization of a class is created that allows existing code to be easily re-used, modified and extended.

A subclass may be specialized in several ways: extension, re-definition, or restriction [2]. Using extension a new method is added to the subclass. By using re-definition as a specialization technique, the subclass maintains the same interface as the parent class but some part of the implementation may be re-coded. Restrictive specialization allows the subclass to inherit only a subset of the methods of the parent class.

Overloading

Overloading permits methods in a class hierarchy to use the same name but to overload the meaning. Overloading is actually another form of polymorphism. In the example shown in Figure 7, the *Draw()* message is an example of an overloaded method (meaning it has a different implementation depending on where it is in the hierarchy).

Interpretation

The principle of interpretation determines exactly how the techniques of flexible sharing are be implemented, specifically at the point where a program is compiled or interpreted into machine language. The issues of interpretation generally involve the concepts of *type checking* and *binding*. When a compiler performs type checking, it determines whether operations are supported by a particular object or type and whether type inconsistencies will result (ie. can a string be added to an integer?). When a compiler performs binding, it tries to locate the correct implementation of a method (which may lie within another superclass). The issue is when to perform typing and binding: compile time (a static process) or run-time (a dynamic process).

Binding: Static or Dynamic?

As a direct consequence of inheritance, classes no longer contain all information about the class in a central location because of the line of superclasses that may exist before it. Thus, binding ensures that the correct method is attached to an object. Binding may be achieved either statically or dynamically.

In static or early binding, methods are bound to objects at compile time. This type of binding relies on the compiler to build a table of class-method relationships and embed calls to methods directly into the code. Static binding has the advantage of no runtime overhead to find the correct method to attach to the object. An additional advantage is that failed bindings (ie. a method that may not exist) are caught at compile time and not at runtime.

Conversely, dynamic or late binding allows methods to be attached to objects at run-time. This offers the advantage of increased flexibility but the disadvantage of possible failed bindings (ie. a method does not exist) at runtime. Dynamic binding is a requirement if an object-oriented language is to support polymorphism.

Typing: Static or Dynamic?

In addition to binding, typing determines whether specified operations are supported by an object and whether type inconsistencies will occur as a result. Like binding, this may be done at either compile time or runtime.

Static typing offers the advantage of entrapment all type errors before program execution begins. This is because all variables and expressions are explicitly bound to a type at compile time. However, this does impose restrictions on the language. Conversely, dynamic typing ensures correctness of type at runtime. This offers more flexibility for the language but also creates extra runtime overhead and burden for the programmer because error trapping facilities must be provided when type inconsistencies do occur (this would never happen in a staticly typed system).

3.2 The Booch OO Design Notation

Several object-oriented design notations exist today. Most common are the EVB, HOOD (Hierarchial Object-Oriented Design) and OOSD (Object-Oriented Structured Design) notations [2,8]. However, Booch's OOD notation is the most common in North America (HOOD is most common in Europe). The purpose of any design notation is to decompose a complex system into smaller subsystems which are more easily approachable. Additionally, a notation should address the issues of data abstraction, information hiding and responsiveness to change (in the future) [2,3]. This section looks in particular at using the Booch Notation in software design. This design method is attractive because not only is it language independent but can be easily adapted into a CASE (Computer Aided Software Engineering) tool (the notation is to clumsy to use by hand)[9].

The Booch notation is based on the Object Model shown in Figure 8 [3]. The Object Model views a software system from several different perspectives. These perspectives include the logical view, the physical view, static semantics and dynamic semantics. The logical view looks at how classes are decomposed and interact with each other while the physical view looks more closely at hardware issues such as where certain classes are defined and implemented and which processes will be

9 The Rose CASE Tool by Rational Systems fully supports the Booch design notation.

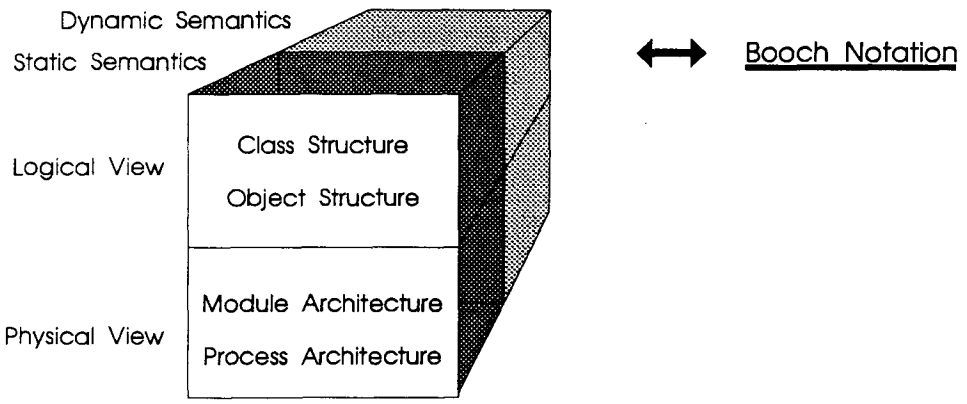


Figure 8.
The Object Design Model forms the basis of the Booch Design Notation.

performed by what processor (in a concurrent system). In the Booch Notation, class diagrams and object diagrams are used to design and present the physical view of the system. Module diagrams and process diagrams are used to design and present the physical view of the system.

The four diagrams listed so far generally describe static processes that are time independent (ie. a snapshot of the system at on instance in time). These are representations of the static semantics of the system. The dynamic semantics of the system describe the time ordering of events (such as message passing). These semantics are designed and presented using state transition diagrams and the timing diagrams.

The purpose of this section is to provide a brief overview of some of the diagrams used in the notation - in particular, class diagrams and object diagrams. The complete specification of the Booch Design Notation can be found in reference [3].

The Process

The general approach using the Booch Design is a five step method. Blair et. al describe this development using the following steps [10]:

(1) identify the objects and their attributes

(2) identify the operations required by each object

10 Blair, Gordon et Al., *Object-Oriented Languages, Systems and Applications*. pp.206-207 [2].

(3) establish the visibility of each object relative to other objects

(4) establish an interface to each object

(5) implement the objects

The first step, the identification of objects and their attributes, is probably the most difficult stage in the process because of the multitude of different and correct groupings that may exist in a problem space. Booch suggests identification of key attributes based on the *nouns* used to describe the problem space.

Identification of the operations suffered by and required of each object establishes the static and dynamic semantics of the object. At this stage it is important to look at the operations required of an object because it enforces decoupling of objects from one another. The third step, to establish object visibility, determines the static dependencies required amongst objects. In other words, one determines what objects see and are seen by a given object.

The establishment of the interface to an object determines the protocol for which objects can communicate amongst themselves. The interface forms the boundary between the internal and external view (implementation) of an object. The final step involves implementation of the object whereby a representation of each object or class of objects is chosen.

Class Diagrams

Class diagrams are used to present the class structure (hierarchial information), its specification, and its relationship with other classes. Class diagrams are built using several different icons and templates. These include:

- Class Category Icon
- Class Icon
- Class Relationship Icon
- Class Template
- Operation Template

Class Category Icon

Because class diagrams can sometimes get very large, the class category can be used to organize them into meaningful blocks. The class category is used to represent only the highest

levels of abstraction in the class diagram and thus gives only an overview of the general architecture of the system (Figure 9). Generally, every component in a class category diagram is a high level reference to a class diagram or another class category.

Figure 9.
The Class Category Icon.

Class Icon

The class icon, shown in Figure 10, represents a class in a class diagram. The class name is enclosed within the dashed line. The dashed line is used to show that clients of the class operate on instances of the class (IE. objects) and not the class itself.

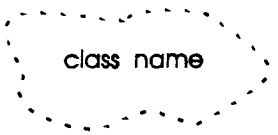


Figure 10.
The Class Icon.

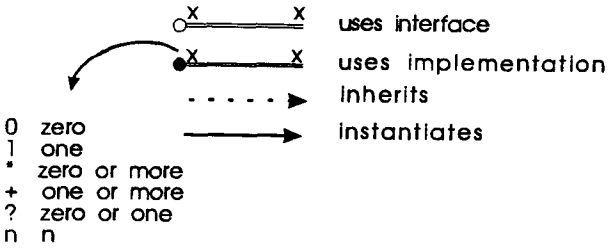


Figure 11.
Class Relationship Icons and Cardinality.

Class Relationship Icon

The class relationship icon shown in Figure 11, displays the various relationships that may exist amongst classes. These include the using, inheritance and instantiation relationships. Using relationships employ a double line with a circle placed by the class that uses the other class. If the circle is filled, the implementation of the class is being used; if not filled, its interface is being used. Circles can be placed on both ends of the icon in instances where there is a two way relationship.

Two numbers can be placed on the icon to indicate cardinality, the number of objects affected by the relationship. For example, a container class would have one instance, but many objects in the container so the cardinality here would be *1* to *n*. If the cardinality is 1 to 1, is is not placed on the icon. Inheritance relationships are shown using a single solid line with the arrow always pointing to the parent class (superclass).

Class Templates and Operation Template

The class template provides the detailed documentation of each class in the system (Figure 12). The information presented is generally an amalgamation of all the other class diagrams. Class templates provide descriptive narrations of the class, inheritance information, class interface and implementation, and descriptions of operations the class perform. Obviously, the class template can become very detailed so it may only be used later on in the analysis/design stage. Additionally, it is not necessary to use all the fields within the template; only the ones capture the important design decisions. The operation template is a spin-off of the class template. The operation template is used to specifically describe the operations (member functions) performed by the class (Figure 13).

Name:	*identifier*
Documentation:	*text*
Visibility:	*exported/private/imported*
Cardinality:	*0/1/n*
Hierarchy:	
Superclasses:	*class names*
Metaclass:	*class name*
Generic Parameters:	*list of parameters*
Interface/Implementation:	
Uses:	*list class names*
Fields:	*list field*
Operations:	*list operations*
Finite State:	*state transition diagram*
Concurrency:	*sequential/blocking/active*
Space Complexity:	*text*
Persistence:	*persistent/transitory*

Figure 12.
The Class Template.

Name:	*identifier*
Documentation:	*text*
Category:	*text*
Qualification:	*text*
Formal Parameters:	*list of declarations*
Result:	*class name*
Preconditions:	*object diagram*
Action:	*object diagram*
Postconditions:	*object diagram*
Exceptions:	*list of exceptions*
Concurrency:	*sequential / guarded / concurrent / multiple*
Time complexity:	*text*
Space complexity:	*text*

Figure 13.
The Operation Template.

Object Diagrams

Object diagrams are used to explain objects and their relationships in the logical design of the system. While class diagrams describe the static semantics, object diagrams describe the dynamic semantics of a design. However, the two are closely related since an object is an instance of a class

and both adhere to the same sets of operations. The difference is "class diagrams document the key abstraction in the system, and object diagrams highlight the important mechanisms to manipulate these abstractions"[3]. An object diagram is completed using the following icons and templates:

- Object Icon
- Object Relationship Icon
- Object Template
- Message Template

Object Icon and Object Relationship Icon

The object icon in Figure 14 is similar to the class icon except a solid line is used to draw the boundary of the object.

Figure 14.
The Object Icon.

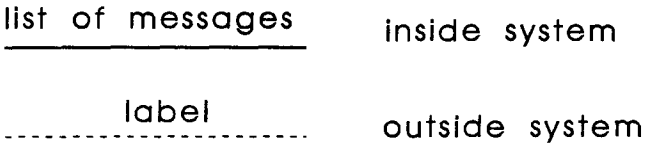


Figure 15.
The Object Relationship Icons.

The object relationship icon shows the flow of messages between objects using lines (Figure 15). A solid line represents a relationship amongst two objects inside the system and a grey line provides a method to document the relationship between the object and any objects outside the software application. Figure 16 shows several detailed icons can be placed at the ends of the object relationship icons to express visibility between objects.

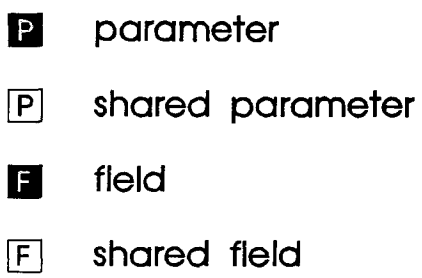


Figure 16.
The Object Visibility Symbols.

Object and Message Templates

As with the class template, the object template provides more detailed information about the object, specifically on the class of the object and its persistence qualities (Figure 17).

Name:	*identifier*
Documentation:	*text*
Class:	*class name*
Persistence:	*persistent / static / dynamic*

Figure 17.
The Object Template.

Operation:	*operation name*
Documentation:	*text*
Frequency:	*aperiodic / periodic*
Synchronization:	*simple / synchronous / balking / timeout / asynchronous*

Figure 18.
The Object Message Template.

Object message templates document the individual messages an object may send to other objects (Figure 18). This template is generally only used to document time critical operations.

3.3 Pragmatics of Object-Oriented Development

Like any other development paradigm, object orientation has its own set of problems, mostly in the area of classification. As already discussed, a large part of the OO analysis and design process lies in the area of classification - the discovery and invention of key abstractions and their interactions. Most inexperienced designers encounter problems in mapping a complex problem space into something ordered. In class based systems, the class hierarchy is the result of this mapping. However, in most instances, a wide variety of solutions may exist in the creation of a class hierarchy because of many different classification criteria.

Generalization and Specialization

One of the main objectives in building a class hierarchy is to maximize code reusability. This is sometimes done by embedding generic behavior into a base class for use by all derived classes. An example presented by Duntemann [9] shows some of the shortfalls of this philosophy. The example involves implementation of an abstract class called *Field* which would enable editing of different types of values (a string field, a data field, or an integer field). The *Field* class takes care of operations that every field requires. This would include painting a title for the field and indicating the field is active but exclude the actual editing of a value. Derived from *Field* would be the classes that perform the actual editing of the value. Operations in the derived class would include moving the cursor within the field, ensuring that only numbers are only typed into an integer field or deleting characters from the field. The derived classed may include a *StringField,* an *IntegerField* or a *DateField* (Figure 19). It is evident, however, that there is a great deal of code duplication (three times), particularly for processes such as moving the cursor in the field and formatting characters read from the keyboard.

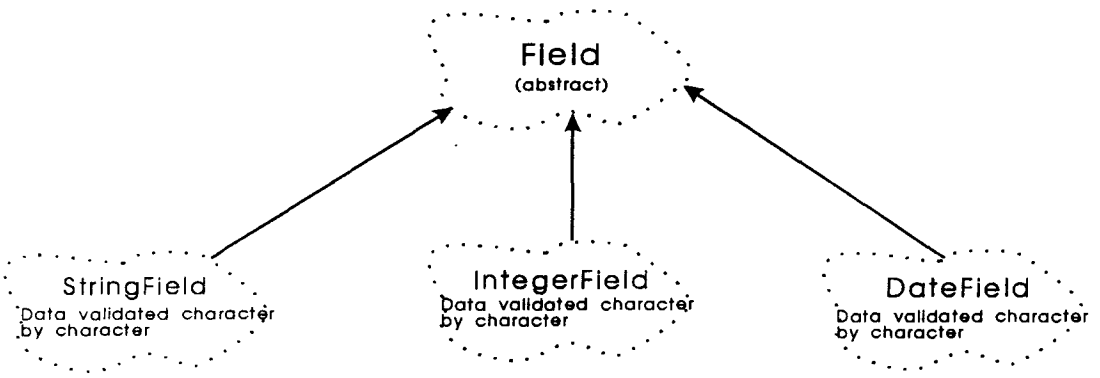


Figure 19.
A class hierarchy for an editing class to edit strings, integers and dates.

Shown in Figure 20, is an alternate proposal for editing different values. This arrangement is built on the the premise that a string class can handle all input and when completed, need only be verified by the child class (*DataField* or *IntegerField*). Thus, most of the code used to perform all editing operations is written only once. The problem with this arrangement is that data validation is only performed *after* the user presses the enter key. A user editing an integer value would be

permitted to enter an alphabetic character into the field but would only be informed of the error after pressing the Enter key. In the first arrangement (Figure 19), this scenario would never arise because the code that reads the keyboard is customized to that particular type of field. This highlights a problem that will often be encountered in differient situations. That is, there is a trade-off between code reusability (and therefore programme size) and getting the desired behavior from a class.

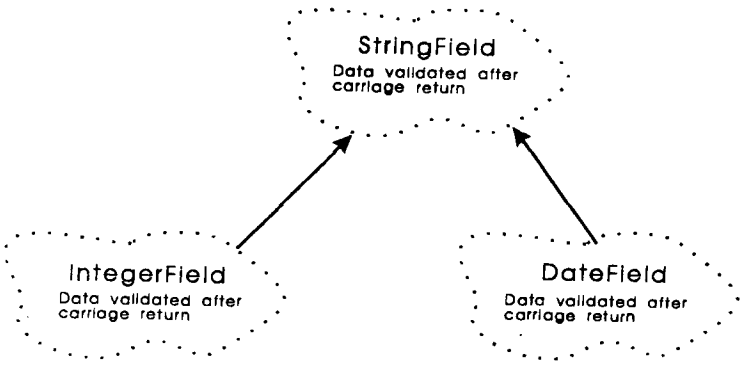


Figure 20.
Revised implementation to enhance code reusability. All editing is performed in StringField and validation is performed by the subclasses.

Duntemann refers to this problem as the case of *distribution* versus *extension*. In the first scenario, most of the behavior of the class is distributed throughout many subclasses with little embedded in the abstract class. In contrast, the second approach builds a fully functional base class which is then extended to support any required variations.

Designing for Reuse

Object-oriented programming promises to increase programmer productivity through software reusability through use of the techniques of encapsulation and inheritance. Two different approaches can be used in designing a system to maximize reusability. These are the *subclassing approach* or the *construction approach* [20]. In order to compare the two concepts, consider the example of a container class used to hold a list of strings. Assume that a generic container class, *Container*, holds generic objects that must be derived from the generic class *Object*.

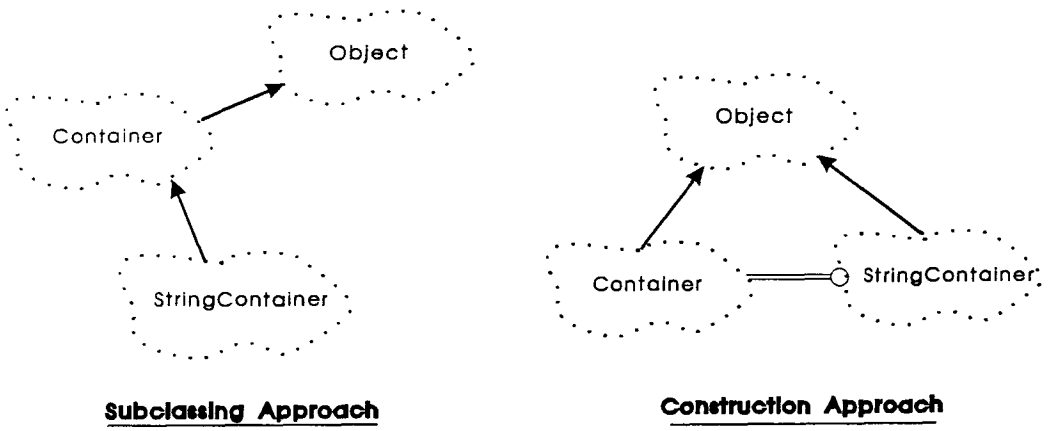


Figure 21.
Reusability using the Subclassing Approach or the Construction Approach.

The Subclassing Approach

The subclassing approach derives *StringContainer* from *Container* so that *StringContainer*, is given complete access to the interface and implementation of *Container*. Operations for adding or removing objects from *Container* need not be defined in *StringContainer* since they are already defined in *Container* and are be called directly. This is desirable from both reusability and flexibility points of view. On the other hand, subclassing may not be the best alternative in situations where the preceding hierarchy is very large. Here, the large number of methods that are inherited become difficult to manage (since one picks up both interface and implementation). Additionally, information that was cautiously hidden in the implementation part of the class suddenly becomes accessible to a derived class (that maybe should not have that access), thus defeating the whole concept of interface and implementation. The subclassing approach should only be used in instances where the class to be derived is really a *subtype* of the parent class.

The Construction Approach

Using the construction approach, the *StringContainer* is implemented as a derived class of *Object*. Included in *StringContainer* is a field which references (*uses*) the interface to *Container*. Unlike the subclassing approach, *StringContainer* now requires methods to be declared to interface with *Container* (ie. to Add or Remove a string from Container). This approach is

beneficial from a maintainability point of view since it only uses the interface to *Container* and is thus more resilient to any future modifications made in *Container*. However, many times the interface to the reference class (*Container*) is insufficient and the designer is forced to use the subclassing approach simply to gain access to a particular method hidden in the implementation part of the class. Additionally, the construction approach requires implementation of additional interface methods to perform operations such as adding and removing strings from the container (if inheritance were used, these methods are automatically included in the derived classes interface).

4 CROSS LINK: A UNIVERSAL STRUCTURAL ANALYSIS PREPROCESSOR

In the past, analysts using computer aided structural analysis programmes have had to work at an unnecessarily low level of detail. A typical analysis session would start with a paper sketch of the structure to be analyzed. On this sketch, the analyst would identify all *node points* and *elements.* Nodes and elements would be numbered in some particular sequence. From this sketch, the analyst would proceed to enter the data defining the node points and elements into a data file, its format dependent on the analysis programme being used. If the structure being analyzed requires the resources of several different analysis programmes, most times the data would have to be manually re-entered each time. Additionally, since most design offices now use some form of CADD (Computer Aided Drafting and Design), analysis data must be re-entered into the CADD programme for final detailing. Not only is this unproductive, but it also allows for more errors. Throughout the whole analysis procedure, the analyst often expends unnecessary time and energy having to remember correct file formats and maintain correct numbering and connectivity sequences for nodes and elements. Instead of concentrating on the problem at hand, the analyst has to worry about small details that can easily be taken care of by preprocessing software. Clearly, there is a need for structural analysis preprocessing software that allows for intuitive, high level model definition along with a facility that allows the model to be transferred between different analysis and CADD programmes.

The Cross Link Universal Structural Analysis Preprocessor, provides a graphical editing environment that allows structural engineers to perform preprocessing without becoming distracted with small details. Using the mouse, the analyst has the capability ot draw a structure on the display and then use a host of tools to perform editing operations to manipulate the structure. Cumbersome data file formats or node and element numbering schemes no longer need the attention they have received in the past. Cross Link also addresses the data transfer issue between alternate analysis and CADD programmes (and is hence called *Universal*). This been solved via the use of a unique macro programming language. Because the macro language has the capability to manipulate the internal database kept by Cross Link, its uses are limitless:

- Automated parametric structure definition,
- Mesh generation and refinement,

- Enhanced editing capabilities such as alignment and transformation,
- Design modules and,
- Import/Export of data in any format (Binary and ASCII).

The purpose of this chapter is to introduce and discuss some of the software design issues in Cross Link with particular emphasis on the data modelling aspects. This will be concluded with a tour through Cross Link including the use of the macro programming language.

4.1 Requirements

Overview

A software application is required to create and edit a graphical model of a structure which is then used in a numerical analysis programme to analyze deflections and stresses due to loads applied to the structure. Typically, the graphical model is composed of nodes and elements which define key locations and components of the structure. Node points are defined by some spatial coordinates and a code to represent allowable degrees of freedom. Springs may be applied to node points to model semi-rigid support points and can be applied to the same degrees of freedom as the node (x, y and rotational for a 2-D problem). Point loads can also be applied to node points in the same directions as springs. Since springs and loads are not common to every node, they are considered separate entities to the node.

Elements are used to model load carrying members in the structure. In the specification of an element, reference is usually made the lower and upper node numbers, a fixity code and the elements geometric properties (ie. cross sectional area, moment of inertia). An element cannot have zero length or be composed of two identical node points. Elements may also contain loads in the form of an offset point load and/or a uniformly distributed load (UDL). The UDL may be specified with either x,y components or as perpendicular to the element.

Several other types of elements are often used in analysis. One general type is the plate element, used to model surfaces. These elements are typically composed of a variable number of nodes (typically 3 to 8) defined by a particular numbering sequence. Although it is not required that these elements be supported in this specification, it is expected that the application will later be extended to support them.

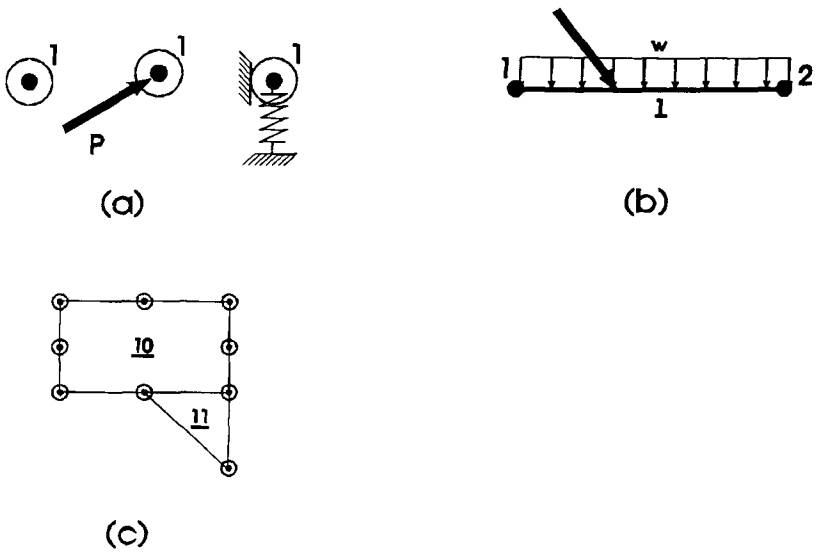


Figure 22.
(a) Nodes points with loads and springs.
(b) Beam element with UDL and offset point load
(c) Some plate elements

A session inside the editor would consist of describing the geometry of the structure, assigning preliminary cross sections to the elements, setting node/element boundary conditions and placing springs and loads on the structure. After saving the structure to a file with a format compatible with the analysis being performed, an analysis would be completed to calculate deflections and stresses. The results of the analysis would then be viewed inside a post-processor. At this stage, code requirement checks would be completed to ensure structural components are correctly designed. The user then proceeds back to the preprocessor to make the required changes to the structure and the cycle is repeated again (Figure 23).

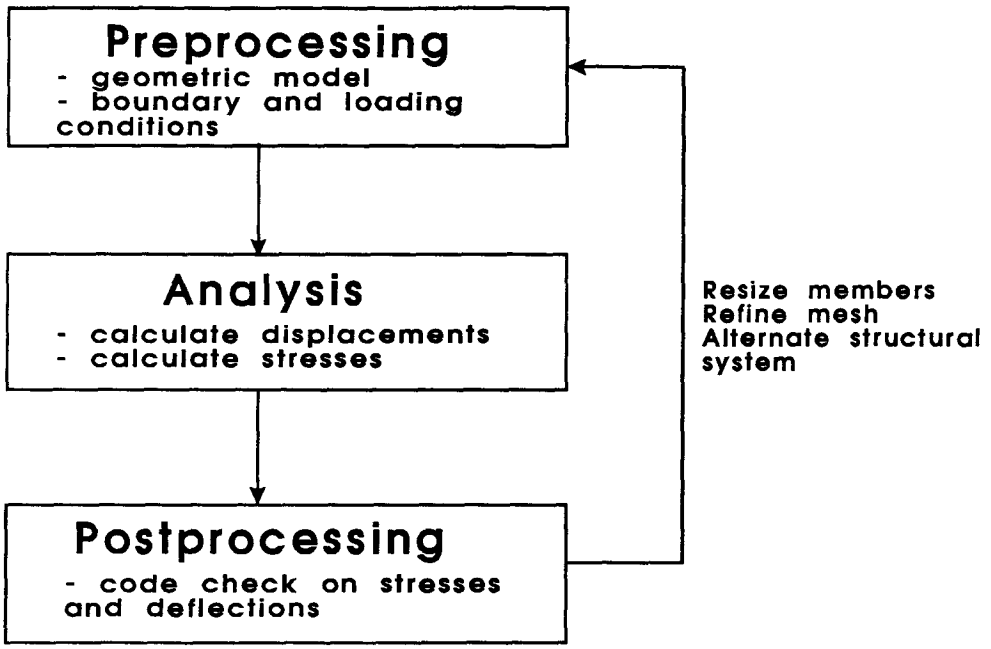


Figure 23.
The stages of Structural Analysis and Design.

Preprocessor Requirements

The first objective of the preprocessor is that it be used as tool for taking a structural concept and rapidly developing a design prototype. Hence its power will be in its inherent ability to quickly generate a model and perform basic editing such as moving or re-sizing components, changing boundary conditions or creating loading cases.

The second objective is that the preprocessor provide a generic link with other analysis programmes. This link must be user defined to allow complete flexibility in communication between the preprocessor and the desired application. The advantages offered by such a feature are twofold. Firstly, this will allow analysts to do prototyping in a single environment regardless of the type of analysis programme being used. Secondly, the information gap between engineer and draftsperson will likely be bridged due to the preprocessor's ability to import and export data via any CADD drawing interchange standard (ie. DXF or IGES). The first version of the preprocessor should provide support for the following structural components:

- Nodes with plane translations and rotations
- 2-D beam elements with fixity control
- Nodal springs and point loads
- Element loads - UDL and offset point loads

Basic tools should be provided to add, delete, re-size and move components within a user defined coordinate system.

4.2 Application Framework

From the requirements presented in the previous section, we may proceed to discuss some of the software implementation details. At this stage, let's look at the application framework for the Cross Link package. The application framework provides a library of highly flexible, pre-written functions to manage the interface between application and user, thus allowing developers to concentrate on problems specific to the application. In this case, a commercial software package called the Zinc Interface Library (ZIL), was selected [21]. This particular library was chosen because it was the first GUI (Graphical User Interface) based library written using object-oriented techniques since the introduction of the Borland C++ compiler. Additionally, the library is completely portable between DOS and Microsoft Windows.

The Zinc Interface Library

The purpose of the application framework is to provide an easy to use, customizable user interface to an application. Because ZIL is object-oriented, customization and modification can be achieved quite easily by simply deriving new classes from the existing framework. ZIL has three important classes that form the engine of the library. These are the display manager, the event manager and the window manager. Along with the these, the window class (UIW_WINDOW) and the window object class (UI_WINDOW_OBJECT) are also important in programme development. Before the role of each of these classes is explained, the foundation of the ZIL library must be explained.

Generic Classes

All of the generic classes in ZIL are built around the concept of the container. A containers is a linked lists which is capable of holding many different types of objects. The only requirement for an object to be registered in a container is that it must be derived from some common class. All containers in ZIL are built from the generic base class, UI_LIST and objects to be placed

in the container must all be derived from the UI_ELEMENT base class. In this configuration, the container class maintains pointers to the first and last objects in the list and the UI_ELEMENT objects provide pointers to the next and previous objects in the list (Figure 24).

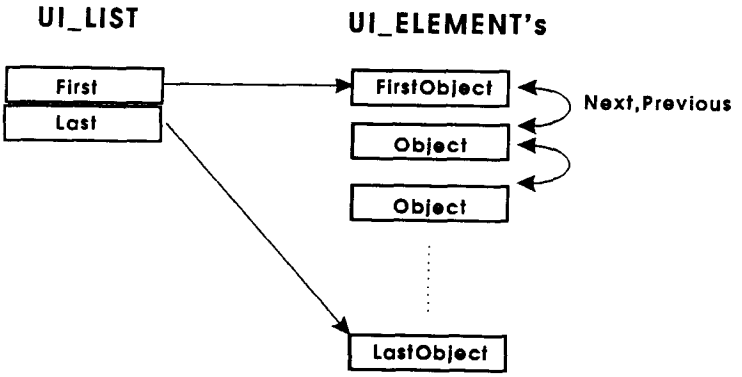


Figure 24.
The UI_LIST and UI_ELEMENT generic classes.

A UI_LIST object acts as a manager for a group of objects. Its basic functions include insertion and deletion of objects to and from the list. Additionally, the manager can traverse the list and send messages to each object. All classes within ZIL are derived from either the UI_LIST class or the UI_ELEMENT class (Figure 25).

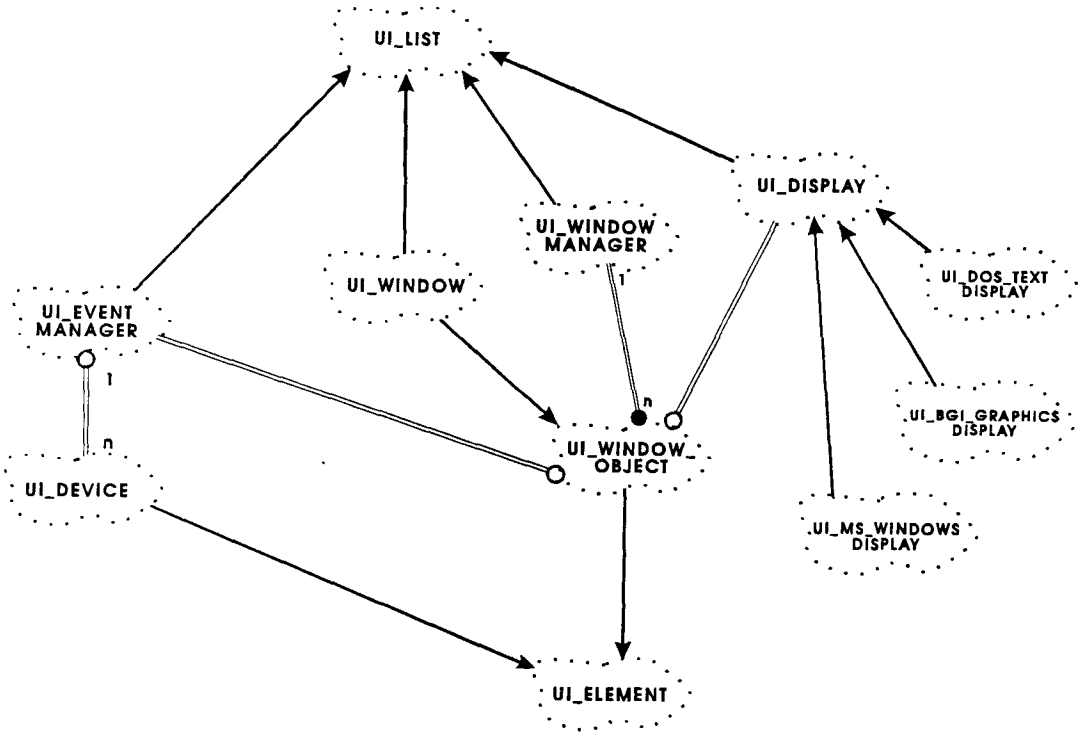


Figure 25.
The Zinc Interface Library Class Diagram.

The Display Manager

The purpose of the display manager (UI_DISPLAY) is to encapsulate all display input/output to provide a consistent application to hardware interface that is easily portable to other environments[11]. The first role of the display manager is to provide an interface between the software application and any hardware devices such as printers or video displays. The display manager also acts as a container for all object regions on the display. Whenever an object (ie. a window) is added to the window manager (discussed below), the window manager registers the region with the display manager. The window manager and the display manager then work

11 ZIL is packaged with several display managers that support the following environments: DOS text mode, DOS graphics mode, and Microsoft Windows. By simply changing the display manager the application ports seamlessly to other environments.

together to determine the areas of all objects in the region list that should be displayed (for features such as overlapped windows). They also determine which objects are affected by events such as mouse clicks.

The Event Manager

The event manager (UI_EVENT_MANAGER) is a container class for all devices (UI_DEVICE) attached to the system. UI_DEVICE objects (or anything derived from a UI_DEVICE) usually provide the communication medium between the user and the application and are generally responsible for generating *events.* Example devices include the mouse, the keyboard, a serial port, or a timer. The purpose of the event manager is to poll all of the devices that is holds, looking for events such mouse or keyboard actions. When an event is generated, the event manager informs the window manager of the type of event that occurred. The window manager then tries to determine which object on the display to dispatch the event to.

The Window Manager

The window manager (UI_WINDOW_MANAGER) is a container class for all window objects. Window objects include windows (UIW_WINDOW) and any objects (discussed below) associated with the window (ie. title bar, borders etc.). The window manager class provides the interface between the event manager and any window objects on the display. In other words, the window manager receives events from the event manager then dispatches them to the appropriate window objects.

Windows and Window Objects

The UI_WINDOW_OBJECT class is an abstract class from which all useable objects such as buttons and icons in ZIL are derived from (Figure 26). In ZIL, window objects are either used to enhance the appearance and behavior of a window or they are used as data fields in a window. This feature is very elegant because it allows the behavior and appearance of the window to be dynamically created and easily customized. Additionally, code size is smaller because only the features that are required are added. Trying to include every window feature provided by ZIL in one data structure or class would not only result in a very inefficient (speed and size) window, but would also be very difficult to maintain. With this dynamic system, each window object is only responsible for itself which results in very easily maintainable objects.

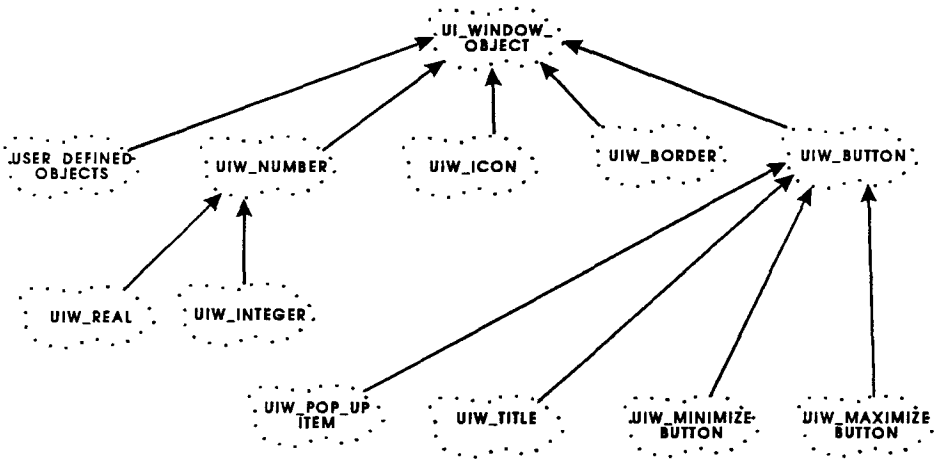


Figure 26.
A partial ZIL window object class hierarchy.

For example, the window shown in Figure 27 is built from several window objects that allow the window to exhibit a wide variety of behavior. The system button (top left corner), when pressed, opens a menu with options to expand the window to full screen or shrink it to an icon, close it or restore it to its original size. The border object allows the window to be re-sized and the title object not only displays the window title but allows the window to be moved when dragged with the mouse.

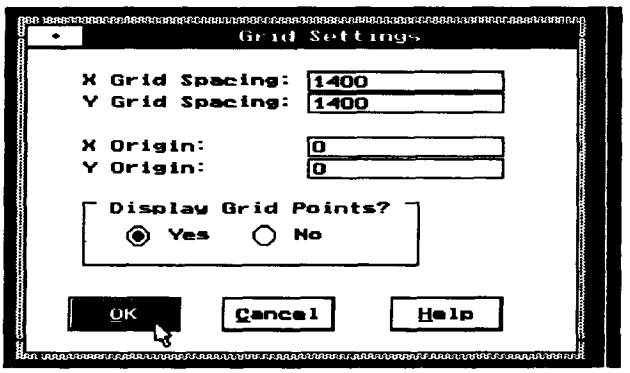


Figure 27.
A standard window created by ZIL.

The radio buttons (Display Grid Points) show another unique feature of the window class. The radio button class is also a window class (RADIO_CONTROL) which can be placed within another window (added to it). The RADIO_CONTROL class corresponds to the area around the two radio buttons (RADIO_BUTTON class) and its responsibility is to ensure that when one button is selected that the other is unselected. The RADIO_BUTTON objects then receive messages from the RADIO_CONTROL to display themselves as selected or unselected.

4.3 Implementation

This section briefly describes how the different features of Cross Link have been interfaced with the Zinc Interface Library using the C++ programming language. It also discusses the implementation of the Cross Link Macro Language that enable users to modify and customize the operation of Cross Link in many different ways. Detailed internal documentation on Cross Link can be found in reference [10].

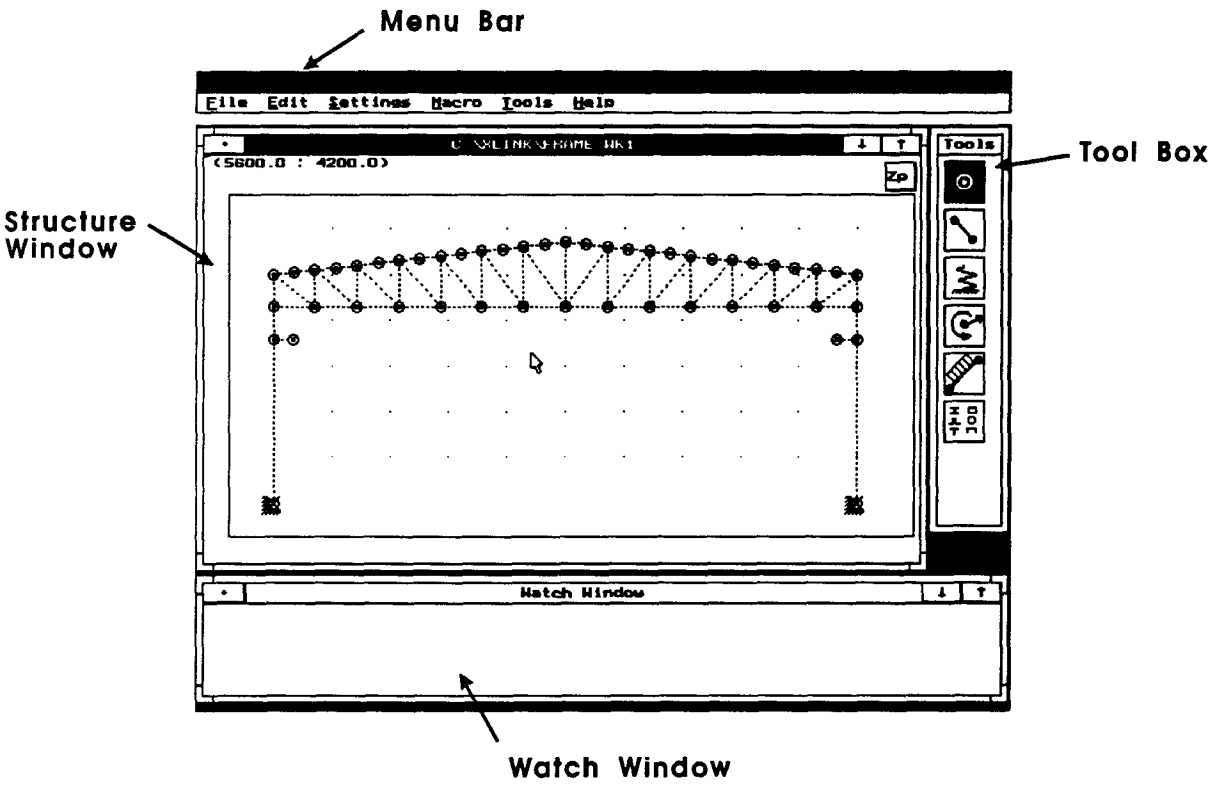


Figure 28.
The Cross Link User Interface.

Shown in Figure 28 is the Cross Link user interface. It is composed of four windows: the menu bar, the tool box, the watch window and the structure window. These features along with other utilities are also presented in Figure 29. The menu bar provides basic services for Cross Link including options to read and write the database to file, set drawing parameters such as the coordinate system limits, grid points and snap points. The menu bar also contains options to execute user macros, run external applications and obtain help. The tool box contains six icons that let the user select an appropriate editing mode. The watch window is a utility used by the macro language that allows macro programmers to write useful macro results to the display. Most importantly, the structure window, which is the engine of the preprocessor, provides the interface between the user and the database.

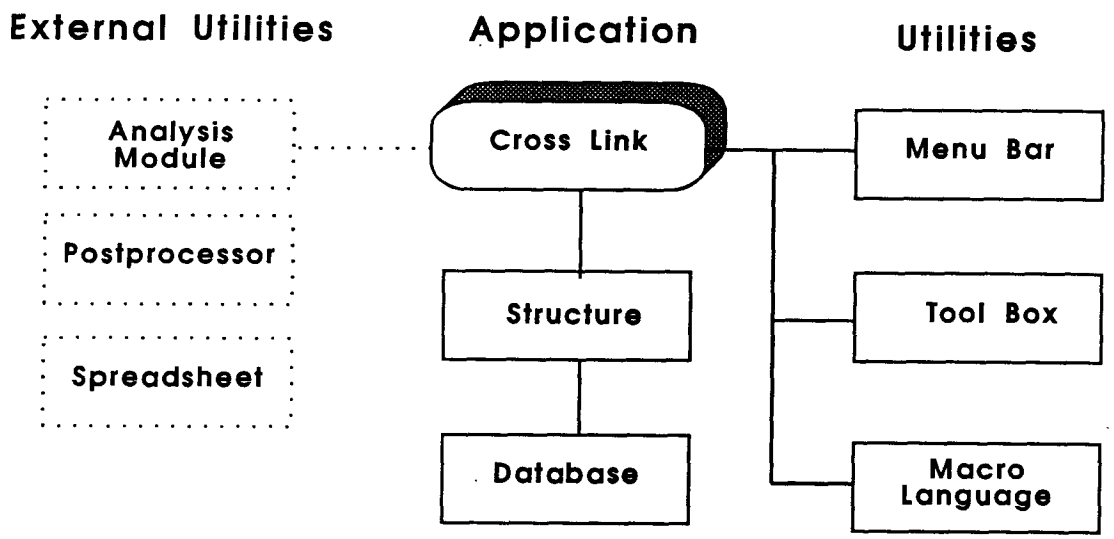


Figure 29.
The Cross Link Application and Utilities

The Structure Framework

Figure 30 shows the class diagram for the structure framework (using the Booch design notation). The four important classes to recognize in this diagram are the UIW_STRUCTURE class, the STRUCTURE class, the SO_NODE_CONTROL class and the SO_ELEMENT_CONTROL class. Referring to Figure 28, the UIW_STRUCTURE class corresponds to the structure window and the STRUCTURE class corresponds to the interior border

in the structure window. The structure database, which provides storage facilities for nodes and elements, is represented by the two node and element controllers. As already mentioned, the structure window and the structure object provide the interface between all utilities (ie. the menu bar or the macro language) and the structure database. In other words, utilities never communicate directly with the database.

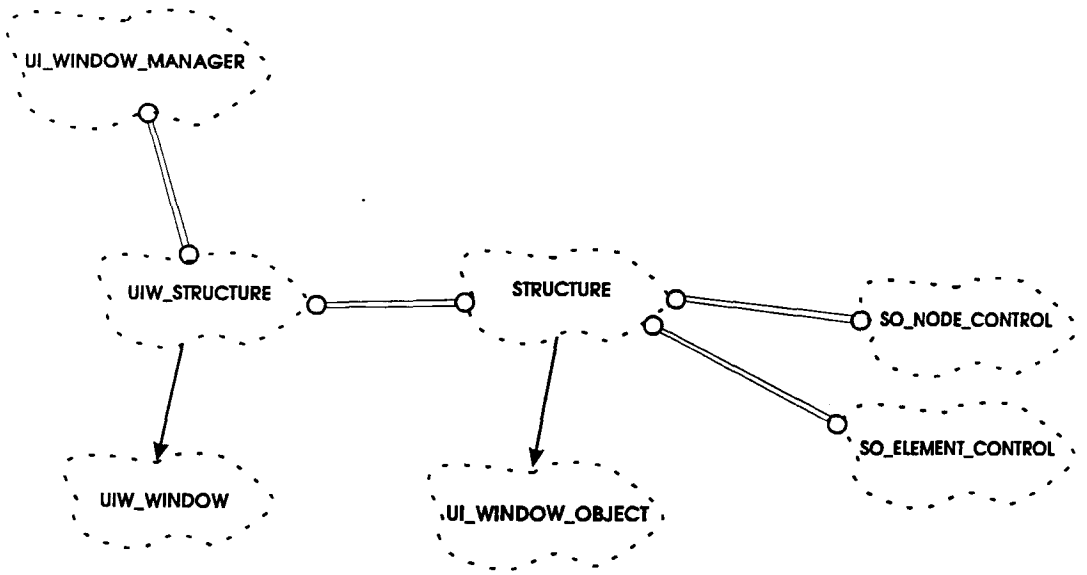


Figure 30.
The structure window and structure object framework (using the Booch Notation).

In order to demonstrate how the system works, consider the Booch object diagram shown in Figure 31. Shown on the diagram are six objects: a menu bar object (*aMenuBar*), a tool box object (*aToolBox*), a structure window object (*aStructureWindow*) and two controllers (*aElementControl* and *aNodeControl*)[12]. The two messages displayed on the diagram correspond to a user using the node tool to set the degrees of freedom on some selected nodes (*SetDOF*) and the using the *Edit...Delete* menu option to erase some selected nodes (*DelectObjects*). Following the *SetDOF* message, it is sent to the *aStructureWindow* object who in turn dispatches it to the *aStructure* object.

12 Container-type objects can be represented in the Booch notation with a series of smaller object icons placed inside the container icon.

The *aStructure* object then sends the message to the *aNodeController* object which sends the *SetDOF* message to all selected *aNode* objects. To erase all selected objects, the *DeleteObjects* message is dispatched in a similar manner except this time, the message is also sent to the *aElementController* object.

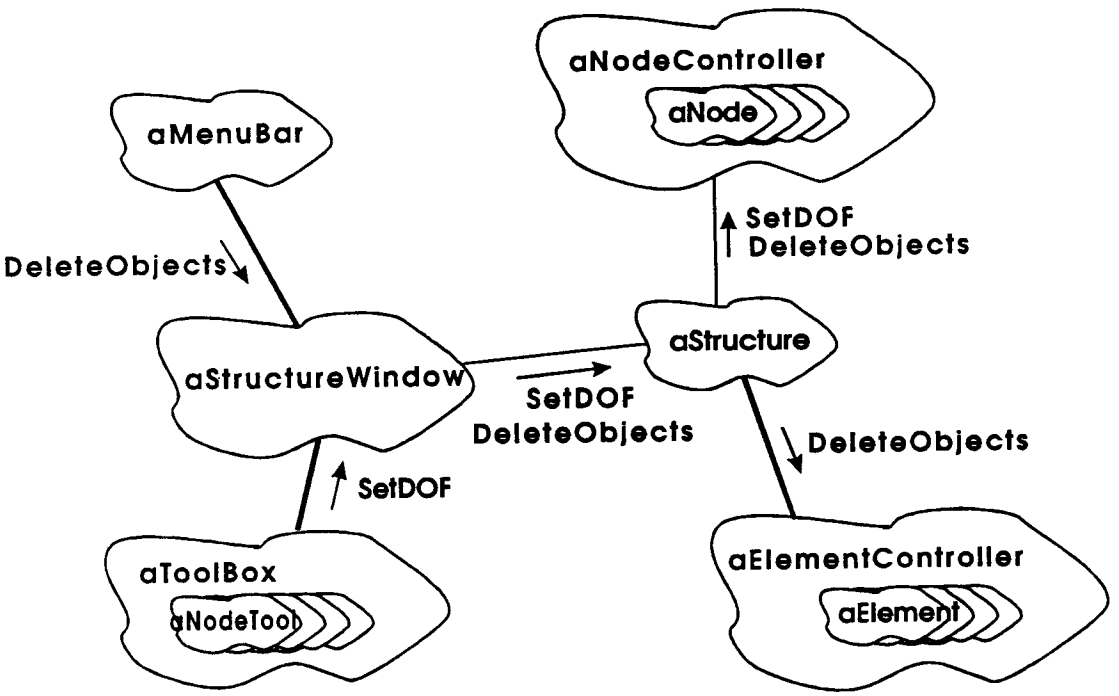


Figure 31.
Cross Link Object Diagram.

Controllers

The structure database is represented using the SO_NODE_CONTROL and the SO_ELEMENT_CONTROL controllers (Figure 32). Controllers provide a powerful, high level method of organizing information. Generally, it is not a requirement of the controller to know the type of objects stored in the container as long as the stored objects abide by the common message protocol defined by the controller. For example, it is perfectly acceptable for a controller to hold a

node object and an element object as long as the objects know what type of messages they can receive from the controller. In this implementation however, two controllers were used. Each one was customized specifically for the type of objects that would be contained.

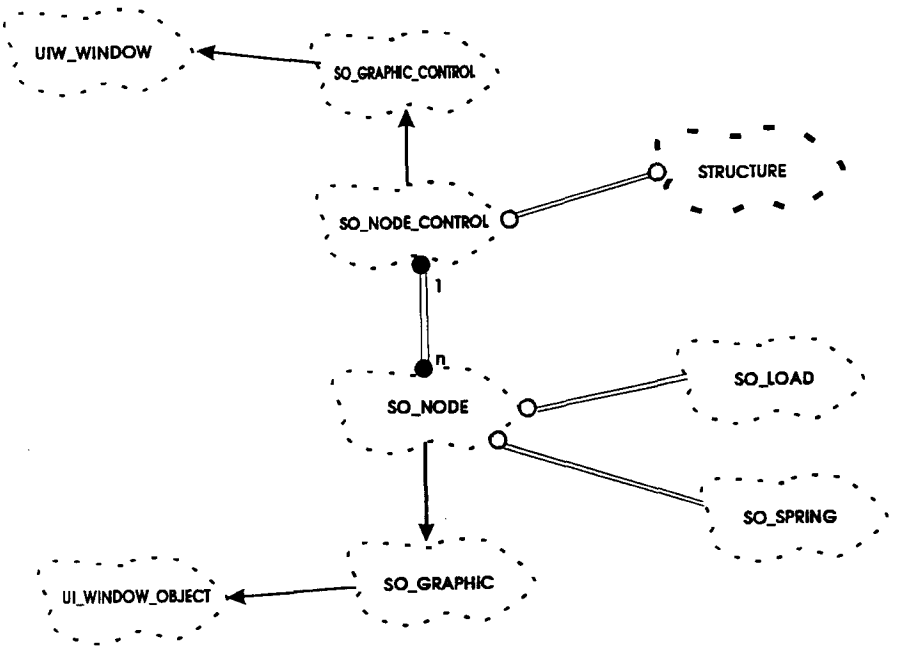


Figure 32.
Implementation of the SO_NODE_CONTROL class and the SO_NODE class. The SO_ELEMENT_CONTROL and SO_ELEMENT classes are similar.

Storing the nodes and elements in separate containers was done for efficiency reasons. With structures that consume a large amount of time in displaying themselves, a user has the option to suppress display of a given object (ie. nodes, elements or loads) in order to increase the overall programme efficiency. For example, when the flag is set to suppress the display of nodes, a draw message sent to both controllers is only accepted by the element controller. In a single controller implementation however, there is no way of filtering a message and sending it to a particular group of objects such as elements. This is because the controller now has to iterate through the complete container and for each object, either blindly send the message or determine if it is a candidate for the message.

The Macro Language

The Cross Link Macro Language (BOB) is a powerful object-oriented programming language that can be used by end users of Cross Link [1]. Through a set of callable library functions that access the Cross Link kernel, users can write macros to manipulate Cross Link in many different ways. Appendix A gives and overview of the language and also provides a complete listing of the macro library functions.

The macro language is implemented using three modules, the lexical scanner, the bytecode compiler and the bytecode interpreter, which are shown below (Figure 33).

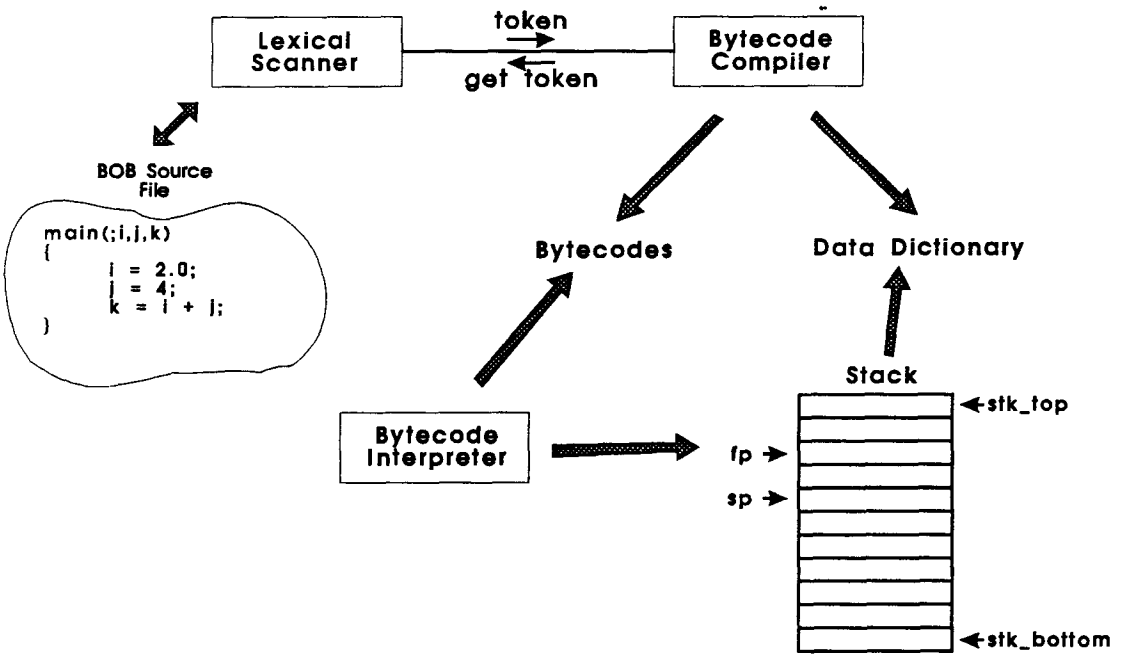


Figure 33.
Layout of the BOB Macro Language Compiler and Interpreter.

The Lexical Scanner

The purpose of the lexical scanner is to scan a macro file and parse it into *tokens* which are fed into the bytecode compiler. The lexical scanner does not perform syntax checking but it must distinguish between language keywords and operators (ie. class, if, else, nil, >=, =) and variables in order to send the correct token to the bytecode compiler (Figure 34).

```
T_INTEGER       integer value (12)
T_FLOAT         floating point value (1.234)
T_STRING        string value ("ab123.0")
T_EQ            equal (==)
T_LE            less than or equal to operation (<=)
T_IF            begin an if statement
T_AND           logical AND (&&)
T_NEW           new keyword
```

Figure 34.
Some example token values returned by the lexical scanner.

The Bytecode Compiler

The bytecode compiler calls the lexical scanner to parse the source file and then using its defined syntax rules, converts these tokens into a series of *bytecodes* which are later interpreted using the bytecode interpreter (Figure 35). The compiler also builds a data dictionary where all variables, functions, class definitions and class implementations are stored. This dictionary is referenced to using the gererated bytecodes. There is a class of possible syntax errors that are not trapped by the compiler and therefore must be located by the interpreter. These errors arise because of the fact that the language is *typeless*. Being typeless, any variables defined within a macro are not required to be explicitly declared as integers, floats or strings (variable types are set when an assignment is made to the variable - ie. x = 123.1; sets the *type* of x to a float). Therefore, the compiler cannot check for incorrect operations such as adding an integer to a string because this is unknown at compile time. If the language were typed, type syntax could be checked by the compiler.

```
OP_PUSH         push nil onto the stack
OP_ADD          add the top two stack entries
OP_DIV          divide the two top stack entries
OP_GE           perform "greater than or equal to" on two top entries
OP_SET          set the value of a variable
OP_CALL         call a function
```

Figure 35.
Some example bytecodes (or opcodes) produced by the bytecode compiler.

The compiler-interpreter arrangement has some advantages over a straight interpreter arrangement (such as BASIC) where syntax checking, compilation and execution are all performed in one pass. Not only is programme execution much faster (because language syntax is verified by the compiler) but development of a runtime-only system that does not include the

compiler could also be implemented. This would allow a compiled macro to be written to a file where it could later be retrieved and executed using the interpreter (the compiler could be packaged as a separate application). This feature would offer a significant speed advantage in instances where macros are never modified and do not require compilation every time they are executed.

The Bytecode Interpreter

The bytecode interpreter executes the series of bytecodes that are built by the bytecode compiler to create a *running* application. It is also responsible for the verification of type correctness at run-time.

To show how the bytecode interpreter uses the stack when interpreting bytecodes, consider the macro shown in Figure 36 which adds two numbers together. Shown in Figure 37 are the bytecodes (opcodes) created by the bytecode compiler.

```
main(;i,j,k)
{
    i = 2.0;
    j = 4.0;
    k = i + j;
}
```

Figure 36.
Macro to add two numbers.

```
TSPACE 03
PUSH
LIT  ; 2.0
TSET 02
LIT  ; 4.0
TSET 01
TREF 02
PUSH
TREF 01
ADD
TSET 00 ; k
RETURN
```

Figure 37.
Resulting bytecode instructions execute the function *main()* in Figure 36.

The interpreter uses a stack as a temporary storage facility for intermediate operations and values (Figure 33). The stack pointer (sp) is used to point to different slots on the stack. In the example programme, the TSPACE opcode allocates three temporary slots onto the top of the stack which are used the three local variables i, j and k. A PUSH opcode is then executed to push a slot onto the stack, followed by a LIT opcode which sets the value of the new slot to the value *2.0*. Next, a TSET opcode sets the value of the variable i to the current value pointed to by the stack pointer (2.0). The same LIT-TSET sequence is also performed for the assignment to variable *j*. The TREF opcode then pushes a slot onto the stack and sets the value of this slot to the value of variable *i* and the same is done for variable *j*. The ADD opcode is then executed

which adds the value pointed to by the stack pointer (j) with the value contained in the next slot above the stack pointer. The result is then stored in the slot above the stack pointer and the TSET opcode is issued to set the value of variable *k* to this value. The RETURN opcode then indicates execution of the function is complete.

Interfacing the Macro Language with other Applications

The macro language supports two different types of callable functions. The first is any function which is declared in a macro file (ie. a subroutine). The second type of callable function, specified by a developer of the system, is called an *internal* or *library* function. As part of the compiler initialization routine, a set of library functions can be added to the data dictionary (a storage dictionary for all macro variables and internal functions) which can then be executed from within a BOB macro. These internal functions are written in C and have to be compiled into the application's executable file.

Programming the internal functions requires minimal understanding of the compiler/interprete operation. The external function is written to accept one parameter in the function call, the number of arguments passed in the BOB function. For example, consider the BOB library function *Sin(value)*, which calculates the sine of *value*. Shown in Figure 38 is the equivalent C function that calculates the sine and returns the result back to the interpreter. Allone need to know in order to add a library function to the language is the state of the stack right before the external function is called. The interpreter will push all the parameters passed to the function onto the stack (in this case only *value*) so the internal function need only check that the parameters number of parameters were passed and that they are of the correct type (obviously, *value* can only be of type integer or float). Once the parameter type is verified, it can be extracted from the stack and the calculation can be performed. In order to pass the result back to the interpreter, the stack pointer must be incremented by the number of arguments in the function. After the stack increment, the result will be placed on the stack where the interpreter will insure it is assigned to the correct variable.

Using this method, a full complement of functions can be developed to interface BOB with functions and data that are internal to the application. This creates a very high level of flexibility

```
// init_functions - initialize the internal functions
void init_functions()
{
     ...
     // add the BOB Sin() function using a pointer to the sine function
     add_function("Sin",sine);
     ...
}

// sine calculates the sine of a value
static int sine(int argc)
{
     float value, retval;

     // make sure only 1 parameter was passed by BOB.
     argcount(argc,1);

     // ensure the parameter has the correct type
     if(!type(0,DT_INTEGER) && !type(0,DT_FLOAT))
          badtype(0,DT_NUMERIC);

     // perform the calculation
     if(type(0,DT_INTEGER)){
          value = sp[0].v.v_integer;
          retval = sin(value);  // call a function to do the calculation
     }
     else if(type(0,DT_FLOAT)){
          value = sp[0].v.v_float;
          retval = sin(value);
     }
     // increment the stack pointer by the number of params passed (1)
     sp += argc;

     // set the value pointed to by sp to the return value of sin
     set_float(&sp[0],retval);
     return(0);
}
```

Figure 38.
The internal function to calculate the sine of the BOB Sin() function.

because, with the proper interface to the *kernel* of the application, it is possible to shift much of the programming from internal development to external development which can be done by end users of the application.

4.4 A Sample Session in Cross Link

The purpose of this section is to provide a tour through CLUSAP. In the example, a model of the frame shown in Figure 39 will be developed. Then using the macro language, the frame will be exported to the ANSYS finite element programme for final analysis.

Start the Cross Link programme by typing *xlink<enter>* at the DOS command line. If this is not the first editing session, any structure that was edited in another session will automatically be loaded. A screen similar to that shown in Figure 40 should appear. The preprocessor environment consists of four windows on the display: the Menu Bar, the Tool Window, the Watch Window and the Structure Window. The Menu Bar contains menu options to open and save files, set environment

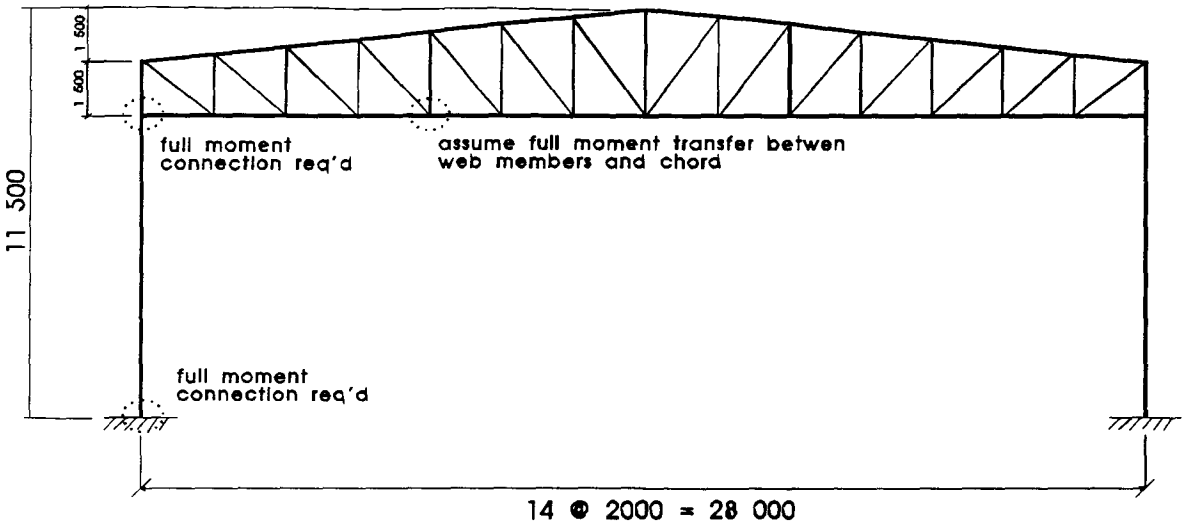


Figure 39.
The Example Frame to be Modelled using Cross Link.

parameters, run macros and obtain help. The Tool Window contains six icons or *tools* that determine the editing mode. The Structure Window is the window where the structure is displayed and edited. Finally, the Watch Window is used as an output facility for the macro language. Macro developers can use this window display information for users of the macro or use it as a debugging facility (by printing variable values to the window).

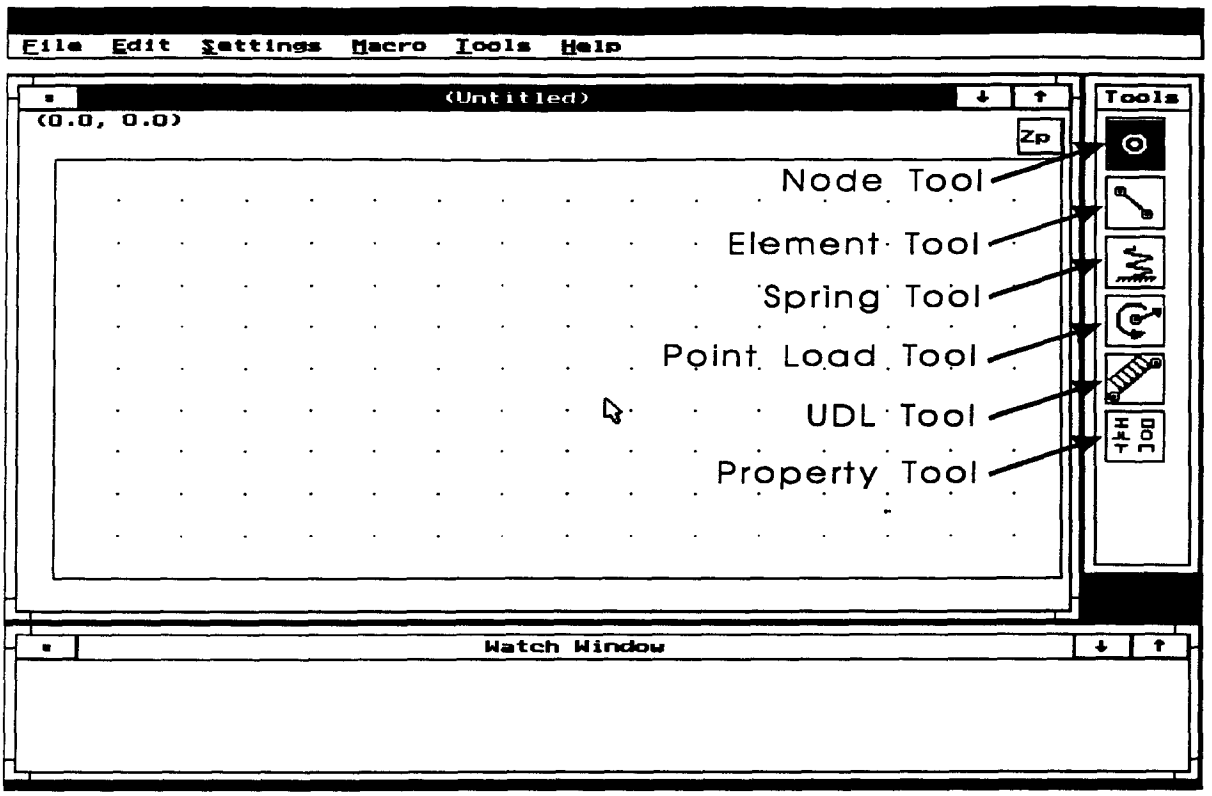


Figure 40.
The Cross Link Preprocessor Editing Environment.

The first step in a session is to set up a coordinate system and a grid sustem. To do this select *Settings...Limits* from the menu and enter the minimum and maximum coordinates of the frame (allowing for some margin space) in the dialogue box (Figure 41). To set the grid size, select *Settings...Grid* and enter 500 (mm) for the x and y grid spacings. The *Settings...Display* option can be used to set some of the structure display options on the screen. In this case, select the display of nodes and node numbers, and element and element numbers (Figure 42).

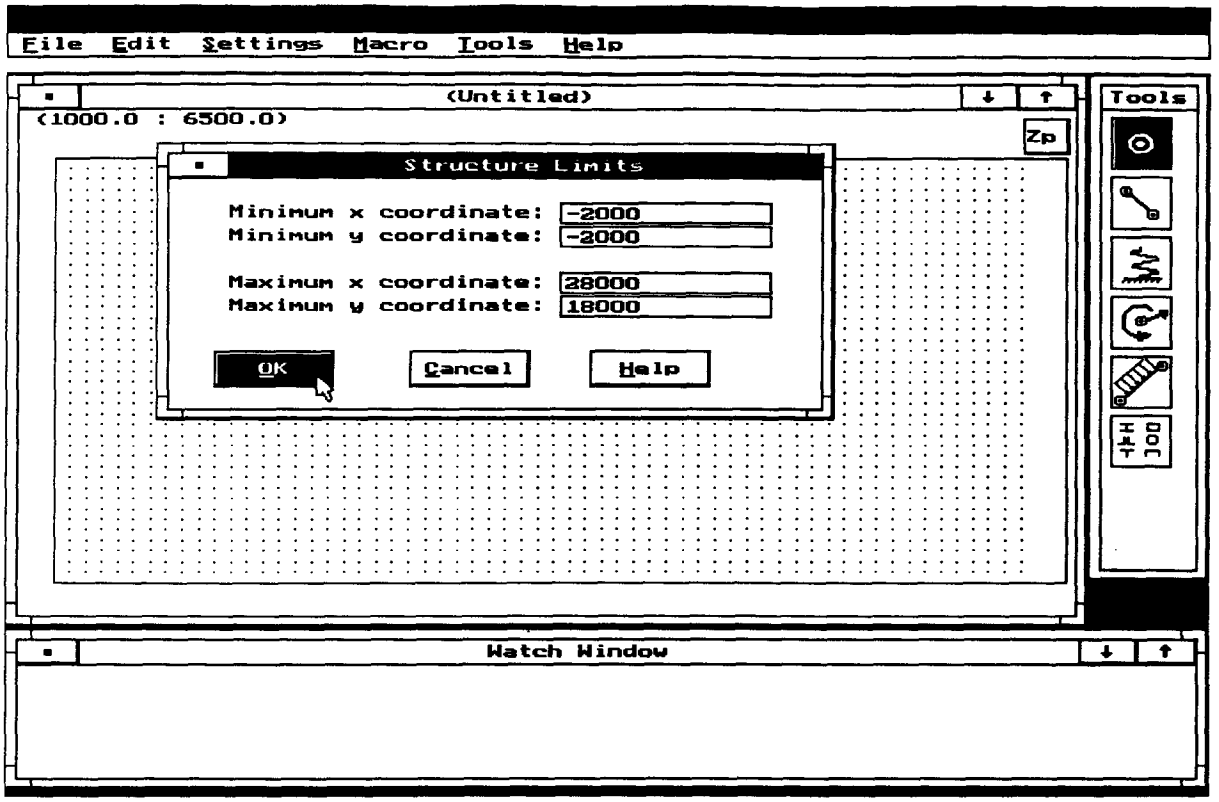


Figure 41.
Setting the Structure Limits.

The preprocessor is now ready to begin definition of the frame. This will proceed with the definition of the node points, then the elements and their properties and finally any boundary conditions (both node and element). To place nodes, select the node tool from the *Tools* window using the left mouse button. Move over the structure window and click the right mouse button to make it current (the window title bar will highlight). Now, using the left mouse button, add the node points for the two columns (Figure 43). The F9 key can be used to toggle the *snap-to-grid* option (the coordinate display will tell you if it is on or off). Similarily, F8 will toggle the grid display. If you wish to zoom into a region of the structure to place the nodes, drag the mouse with the *right button* pressed over the area you wish to zoom. To return to the previous view, click on the ***Zp*** ('Zoom Previous') button.

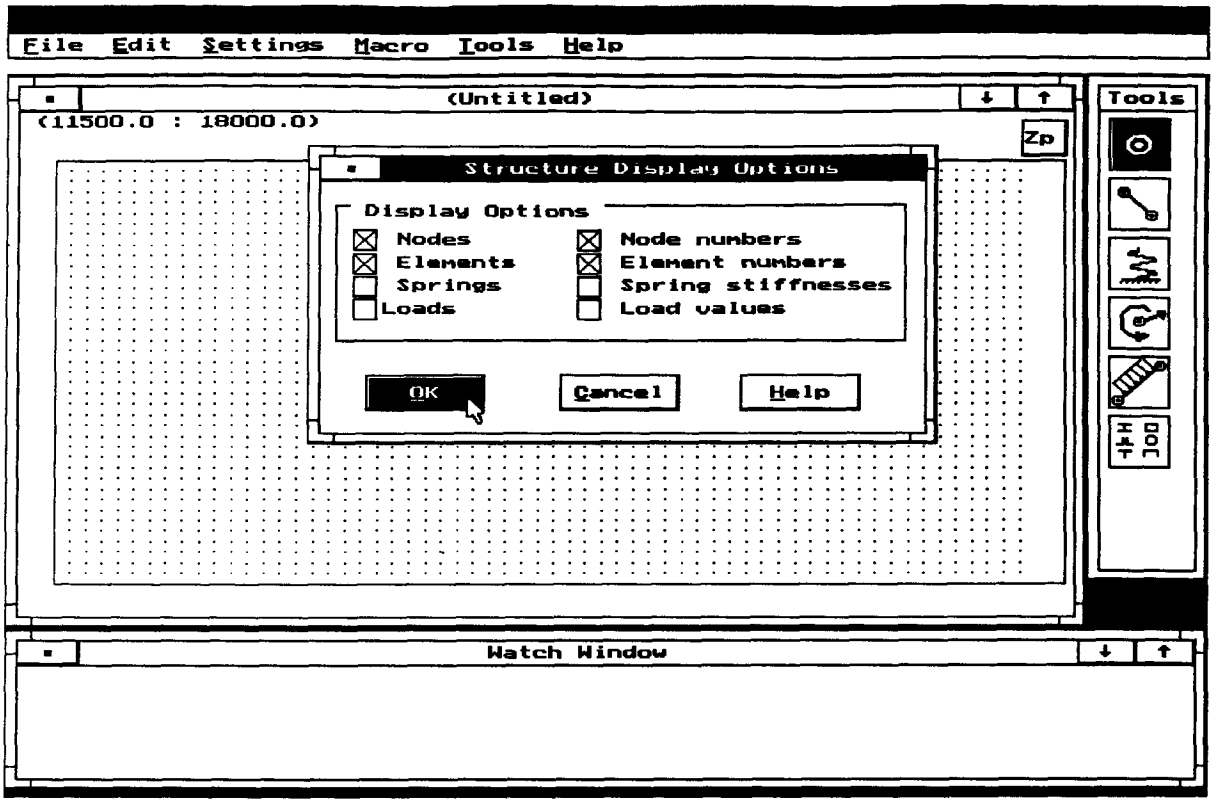


Figure 42.
Setting the Structure Display Options.

Next, select *Edit...Generate* to generate a line of nodes for the top and bottom chords (Figure 44). The node generator contains options to control node placement at the beginning and ending coordinates and optional element generation (with or without properties attached). For the bottom chord, start node generation at (0,8500) and end at (28000,8500) using 14 elements. Do not include generation of the first and last nodes because Cross Link will automatically look at these coordinates for a node to connect an element to.

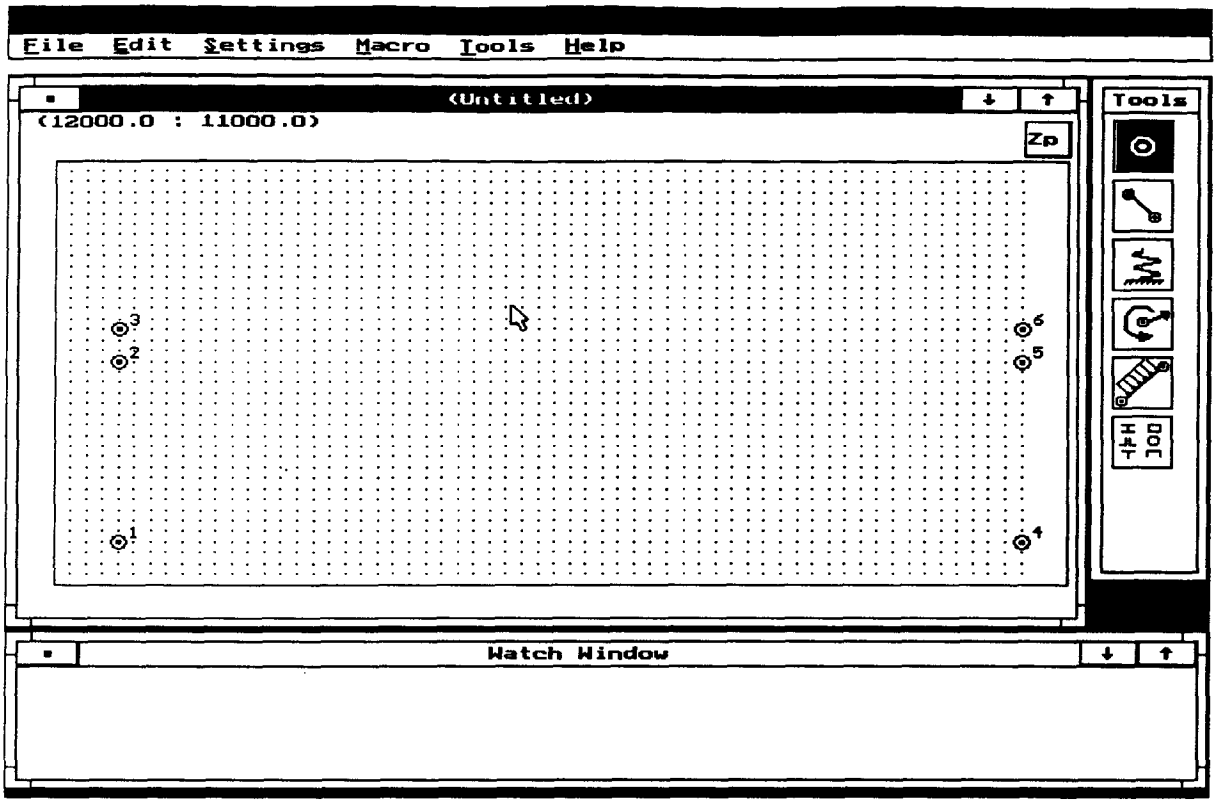


Figure 43.
Placement of the Column Node Points.

For the top left hand chord, start at (0,10000), stop at (14000,11500) and generate 7 elements. This time, include generation of the last node. For the top right hand chord, start at (14000,11500), stop at (28000,10000) and again generate 7 elements. Do not specify generation of the first and last nodes this time since they already exist. Placement of the remaining web elements can now be performed using the element tool. Select the element tool with the left mouse button, move over the frame and select the first node for an element. A rubber-banded element will follow the mouse cursor until the second node point for the element is selected. Element selection is continuous until you press the <Escape> key. Figure 45 shows the final frame model with all nodes and elements placed.

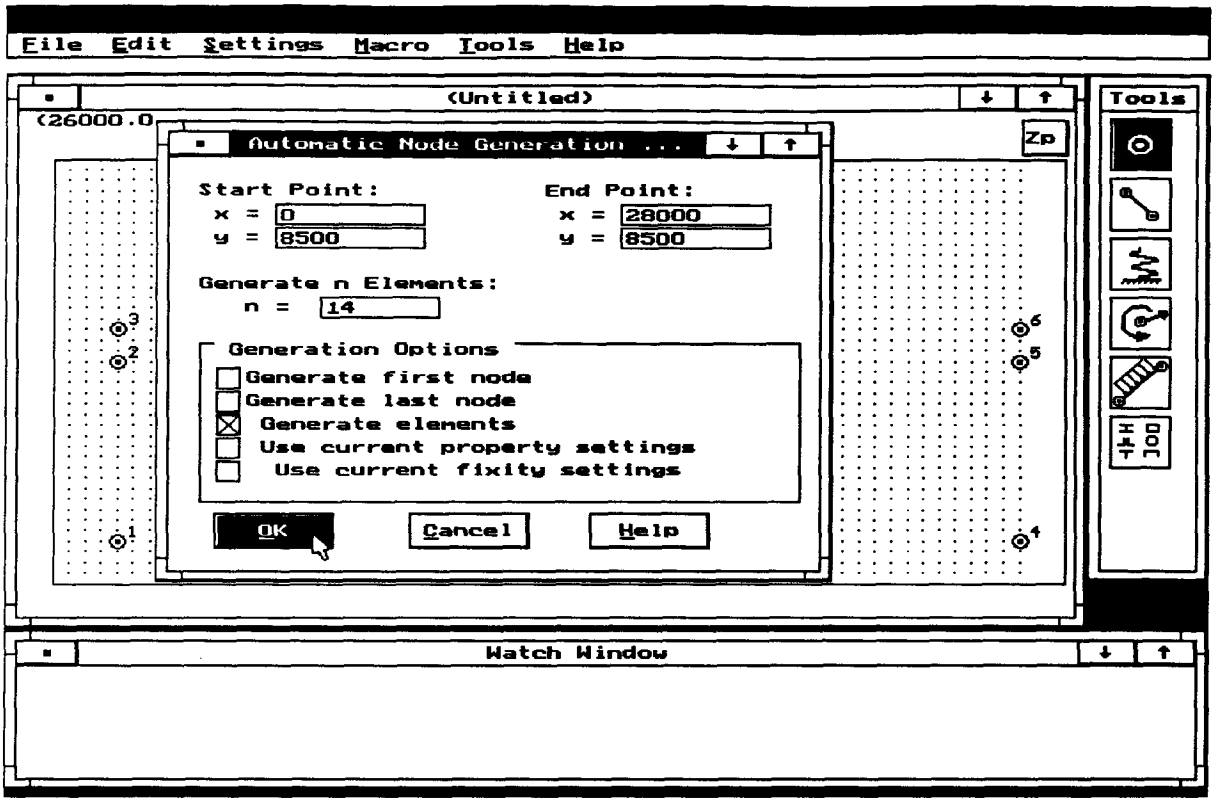


Figure 44.
Automatic Generator of Nodes and Elements for Top and Bottom Chords.

Now you can place the boundary conditions. Start by selecting the two column support nodes by dragging the mouse with the left button held down. Eight *selector icons* will appear on the display encompassing the selected objects. The selector icons can perform two operations when the mouse is moved over top of them and then dragged (a different mouse cursor appears when the mouse is over a selector). Dragging a selector icon with the *left mouse button* down will increase the box size in order to add or remove objects from the *selection set*. Dragging a selector icon with the *right mouse button* down will move all selected objects. The F4 key can also be used to add or remove elements from the selection set. The default selection criterion is to select all elements *inside* and *crossing* the selector box (the region inside the eight icons). Pressing F4 once will then proceed to select only elements *completely inside* the selector box. Pressing F4 yet again will proceed to select all elements *crossing* the selector box.

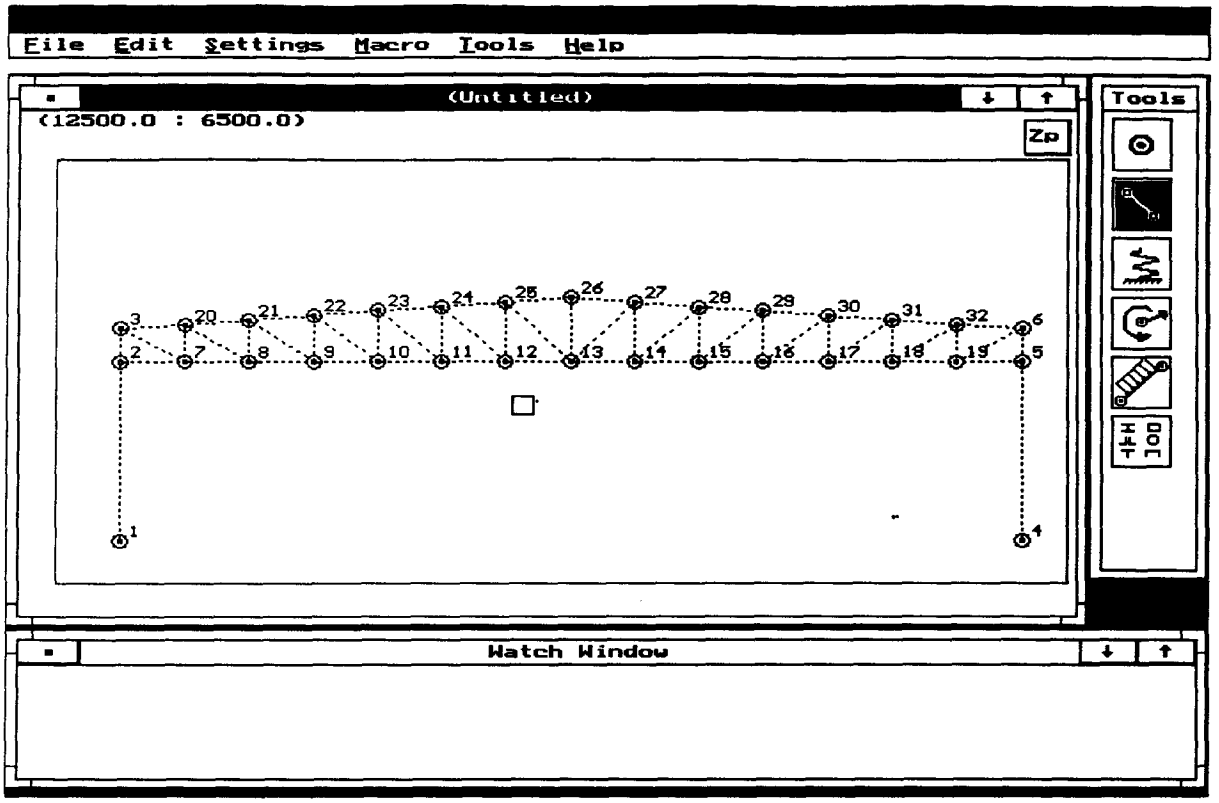


Figure 45.
Final placement of Nodes and Elements.

With the two column support points selected, select the node tool with the *right mouse button.* A window will appear that will display all possible nodal constraints (Figure 46). Select the icon representing a node with no translational or rotational degrees of freedom. Cross Link will proceed to set the boundary conditions on all *selected* nodes.

The frame geometry is now complete. If element releases are required for any elements, select the elements and then select the element tool using the *right mouse button* (The spring tool, the point load tool and the UDL tool all behave in a similar manner). A window will open presenting the available release options for an element (Figure 47). The convention for placing releases is based on the element's high and low node numbers. Note that the elements are displayed as dotted lines when they do not have properties assigned to them. In order to set the element properties, select the desired elements and select the property tool with the right mouse button. A window will prompt you to enter the required properties for the selected elements.

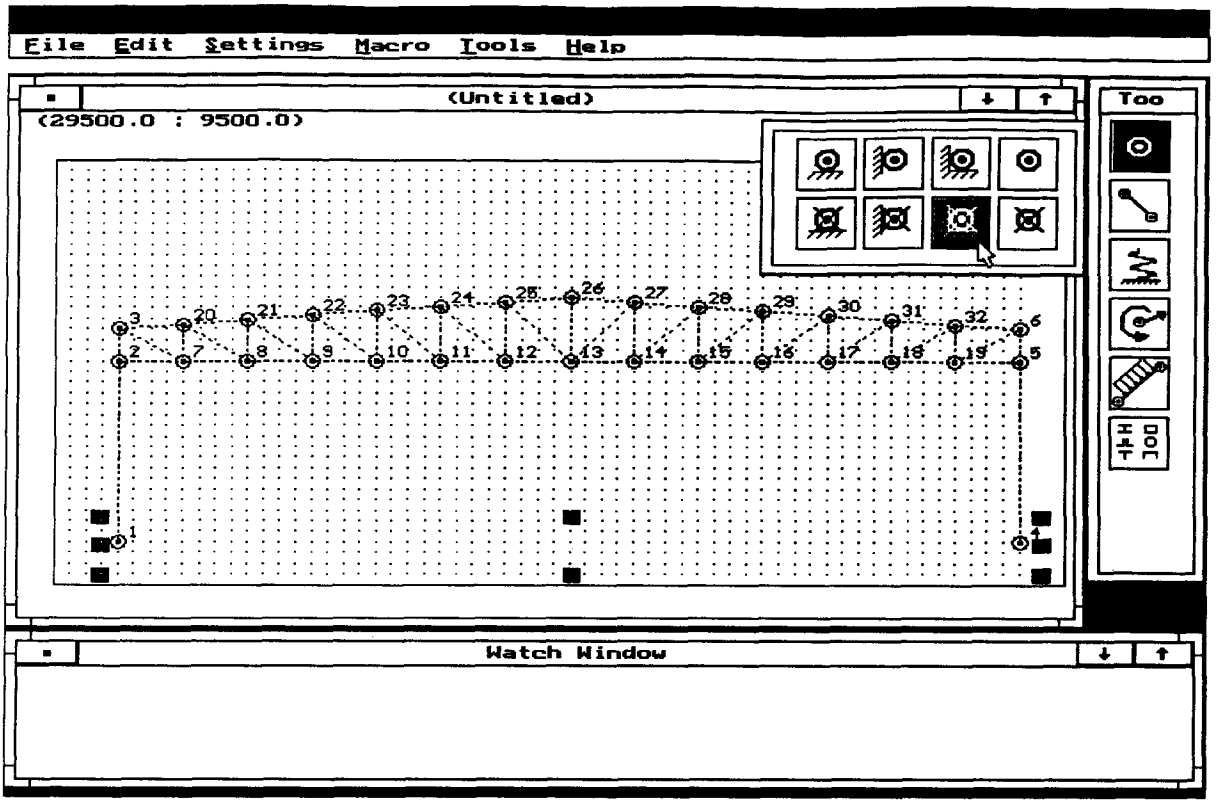


Figure 46.
Placing Node Boundary Conditions. Constraints are placed by selecting the node tool with the right mouse button. The selected nodes (below) will be affected by the selection.

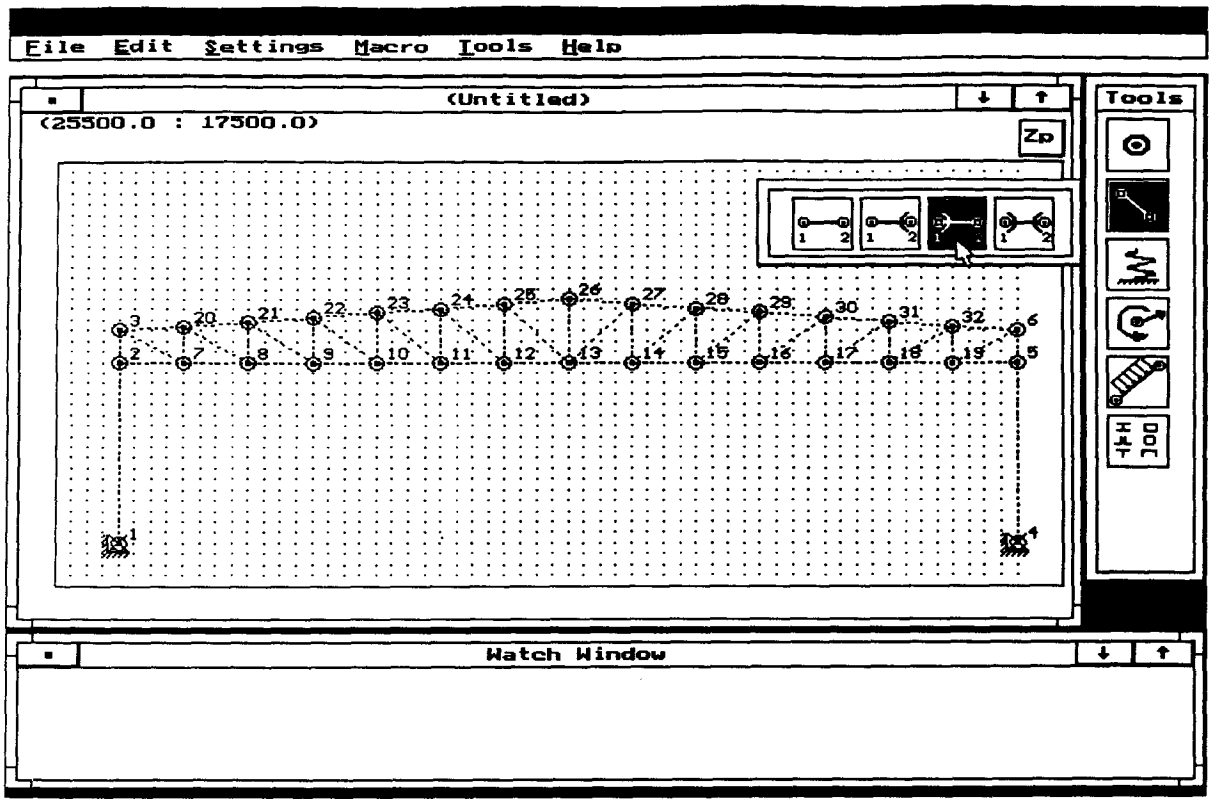


Figure 47.
Setting Element Fixity. Element fixity is specified by selecting the element tool with the right mouse button.

Exporting a Structure to the ANSYS Finite Element Programme

In order to export a structure to the ANSYS finite element programme, a macro must be written to extract the node and element data from the Cross Link database and to write it to an ASCII file in a format readable by ANSYS. Nodes are specified in an ANSYS data file using the N command which has the following syntax:

N,node_number,x_coordinate,y_coordinate[,z_coordinate]

Elements are specified using the E command. The ANSYS E command automatically takes care of the element numbering so its syntax is:

E,lo_node_number,hi_node_number

Nodal boundary conditions are placed using the D command. Its syntax is:

D,node_number,dof,displacement[,,,,dof2,dof3

where:

dof, dof1 and dof3 can be either UX,UY or ROTZ,

displacement = 0.0

For simplification, we will assume that only the node, element and boundary condition information must be provided for ANSYS. To start, using an ASCII editor open a file called 'ansys.bob'. Three functions are required to write the nodes, the elements and the boundary conditions to a file. The first function will be called *writeNodes()*. It takes one parameter, the *file* handle, and is specified as follows (review Appendix A for an overview of the BOB programming language and the Cross Link library functions):

```
// writeNodes - writes all node information to file
writeNodes(file;nodeNumber,x,y)
{
     nodeNumber = NodeFirst();
     while(nodeNumber){
          x = NodeGetX(nodeNumber);
          y = NodeGetY(nodeNumber);
          fwrite(file,"N,",nodeNumber,",",x,",",y,"\n");  \\ '\n' is linefeed

          nodeNumber = NodeNext();
     }
}
```

To extract the nodes, a call is first made to the *NodeFirst()* function which returns the number of the first node in the node database or *nil* if there are no nodes. The x and y coordinates are obtained using the *NodeGet?()* function. The *fwrite()* function is then used to write the information to *file* in an ANSYS compatible format (note that *fwrite()* can take a variable number of parameters). The *NodeNext()* function is then called to get the next node in the database. This is all performed inside a *while* loop until *NodeNext()* returns a value of *nil.*

The function *writeElements()* can be implemented using a procedure identical to *writeNodes()* except this time the two nodes points for the element are required. These are obtained using the *ElementGet??Node()* functions:

```
// writeElements - writes all node information to file
writeElements(file;elementNumber,loNumber,hiNumber)
{
     elementNumber = ElementFirst();
     while(elementNumber){
          loNumber = ElementGetLoNode(elementNumber);
          hiNumber = ElementGetHiNode(elementNumber);
          fwrite(file, "E,", loNumber, ",", hiNumber, "\n");
```

```
            elementNumber = ElementNext();
        }
}
```

To write the node boundary conditions, the node database must be traversed again, this time to retrieve the degree of freedom (dof) code for each node (using *NodeGetDOF()*). This code is compared with a set of constants which represent each allowable degree of freedom for a node. For example, the constant *ID_101* represents a node with no allowable x translations or z rotations. The value of ID_101 is 5 which is obtained from the binary representation 101. Hence logical operators (such as a bitwise OR [|]) can be used on the dof code to determine if a particular dof is free or not. For example, to set the x dof in an existing code (which may or may not have the code set) the dof can be OR'd with the ID_100 code (ie. dof = dof | ID_100). The *writeBoundaryCond()* function is implemented as follows:

```
writeBoundaryCond(file;nodeNumber,dof)
{
     num = NodeFirst();
     while(num){
          // get the node degree of freedom
          dof = NodeGetDOF(num);

          if(dof == ID_100)
               fwrite(file,"D,",num,",UX,0.0\n");
          else if(dof == ID_110)
               fwrite(file,"D,",num,",UX,0.0,,,,UY\n");
          else if(dof == ID_111)
               fwrite(file,"D,",num,",UX,0.0,,,,UY,ROTZ\n");
          else if(dof == ID_101)
               fwrite(file,"D,",num,",UX,0.0,,,,ROTZ\n");
          else if(dof == ID_001)
               fwrite(file,"D,",num,",ROTZ,0.0\n");
          else if(dof == ID_010)
               fwrite(file,"D,",num,",UY,0.0\n");
          else if(dof == ID_011)
               fwrite(file,"D,",num,",UY,0.0,,,,ROTZ\n");

          num = NodeNext();
     }
}
```

All three functions along with the following required *main()* function are placed in the same ASCII file (*main()* usually appears first). The *main()* function will open the ANSYS ASCII data file (harded coded to be 'test.dat') and call the three functions to write the node, elements and boundary conditions respectively:

```
main(;filehandle)
{
     // open a file (test.dat) for writing
     filehandle = fopen("test.dat","w");

     // write the nodes, elements and BC's
     writeNodes(filehandle);
     writeElements(filehandle);
     writeBoundaryCond(filehandle);
```

```
    // close the file
    fclose(filehandle);
}
```

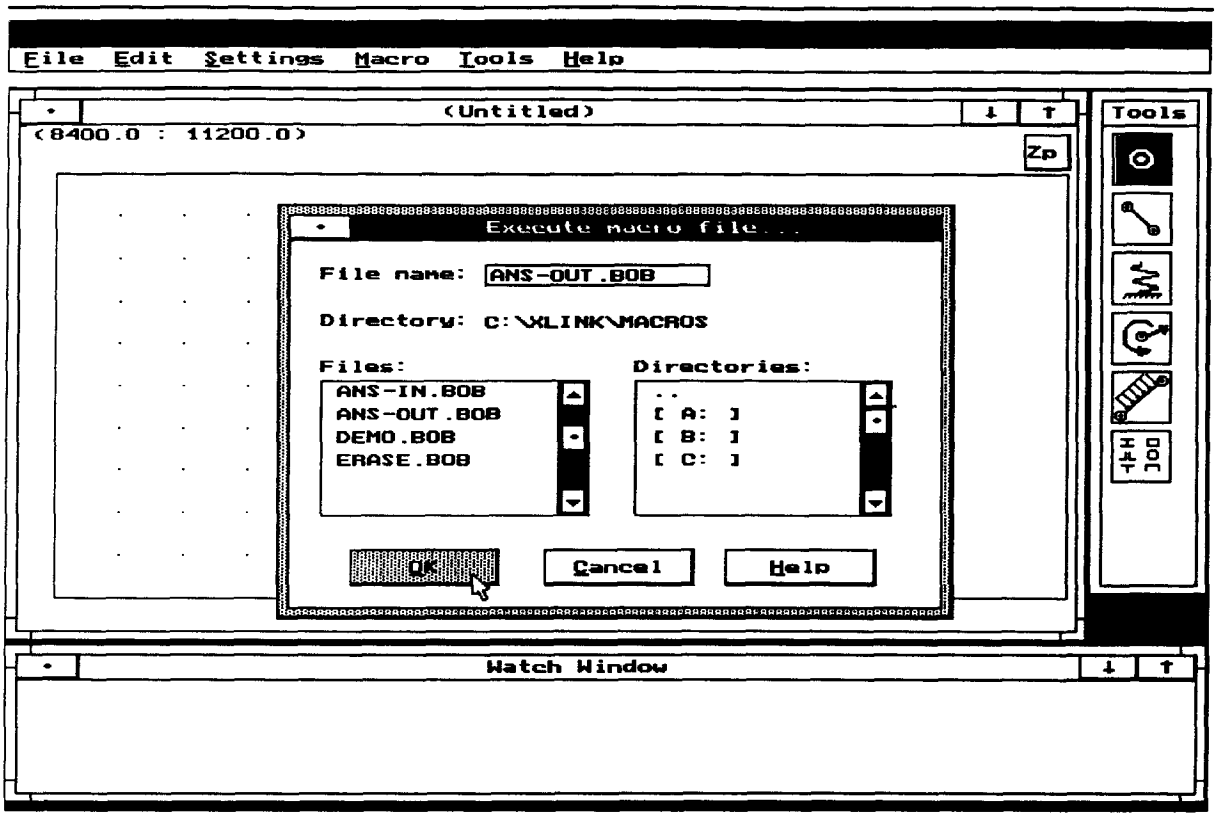


Figure 48.
Running a Macro in Cross Link.

Let's return back to the frame problem. The 'ansys.bob' macro can be run from Cross Link by selecting the *Macro...Run macro* menu option which will present a listing of all macro files (*.bob) in the current directory. Select the *ansys.bob* file and then the *OK* button (Figure 48). The frame will be written to the file 'test.dat' which is presented in Figure 49. Shown in Figure 50 is the frame as interpreted by the ANSYS preprocessor, PREP7.

Listed in Appendix A is a more complete ANSYS file export macro (ANS-OUT.BOB) which comments the data file and also includes a dialog box to get the name of the destination data file. Also presented is a listing for a macro to *import* a commented ANSYS data file (ANS-IN.BOB). Basicly, this macro uses the *fread()* function to read in a line of data from a file and the *token()* function is used to parse the fields from the string. It takes a two parameters; the first is the string

```
N, 1, 0.0000, 0.0000
N, 2, 0.0000, 8500.0000
N, 3, 0.0000, 10000.0000
N, 4, 28000.0000, 0.0000
N, 5, 28000.0000, 8500.0000
N, 6, 28000.0000, 10000.0000
N, 7, 2000.0000, 8500.0000
N, 8, 4000.0000, 8500.0000
N, 9, 6000.0000, 8500.0000
N, 10, 8000.0000, 8500.0000
N, 11, 10000.0000, 8500.0000
N, 12, 12000.0000, 8500.0000
N, 13, 14000.0000, 8500.0000
N, 14, 16000.0000, 8500.0000
N, 15, 18000.0000, 8500.0000
N, 16, 20000.0000, 8500.0000
N, 17, 22000.0000, 8500.0000
N, 18, 24000.0000, 8500.0000
N, 19, 26000.0000, 8500.0000
N, 20, 2000.0000, 10214.2861
N, 21, 4000.0000, 10428.5713
N, 22, 6000.0000, 10642.8574
N, 23, 8000.0000, 10857.1426
N, 24, 10000.0000, 11071.4287
N, 25, 12000.0000, 11285.7139
N, 26, 14000.0000, 11500.0000
N, 27, 16000.0000, 11285.7139
N, 28, 18000.0000, 11071.4287
N, 29, 20000.0000, 10857.1426
N, 30, 22000.0000, 10642.8574
N, 31, 24000.0000, 10428.5713
N, 32, 26000.0000, 10214.2861
E, 2, 7
E, 7, 8
E, 8, 9
E, 9, 10
E, 10, 11
E, 11, 12
E, 12, 13
E, 13, 14
E, 14, 15
E, 15, 16
E, 16, 17
E, 17, 18
E, 18, 19
E, 5, 19
E, 1, 2
E, 2, 3
E, 3, 7
E, 3, 20
E, 20, 21
E, 21, 22
E, 22, 23
E, 23, 24
E, 24, 25
E, 25, 26
E, 26, 27
E, 27, 28
E, 28, 29
E, 29, 30
E, 30, 31
E, 31, 32
E, 6, 32
E, 7, 20
E, 8, 20
E, 8, 21
E, 9, 21
E, 9, 22
E, 10, 22
E, 10, 23
E, 11, 23
E, 11, 24
E, 12, 24
E, 12, 25
E, 13, 25
E, 13, 26
E, 13, 27
E, 14, 27
E, 14, 28
E, 15, 28
E, 15, 29
E, 16, 29
E, 16, 30
E, 17, 30
E, 17, 31
E, 18, 31
E, 18, 32
E, 19, 32
E, 6, 19
E, 5, 6
E, 4, 5
D, 1,UX,0.0,,,,UY,ROTZ
D, 4,UX,0.0,,,,UY,ROTZ
```

Figure 49.
The resulting ANSYS data file for the frame.

to parse and the second is a string listing all possible field delimters in the string (the ANSYS delimiter is a comma [,]). The first call to *token()* returns the first parameter in the string. To parse the remainder of the string, successive calls are made to token with no string passed (ie. tok = token(0,delimiter)) until the *nil* is returned. The *strcmp()* and *val()* functions can be used to compare a string and convert it to a number respectively.

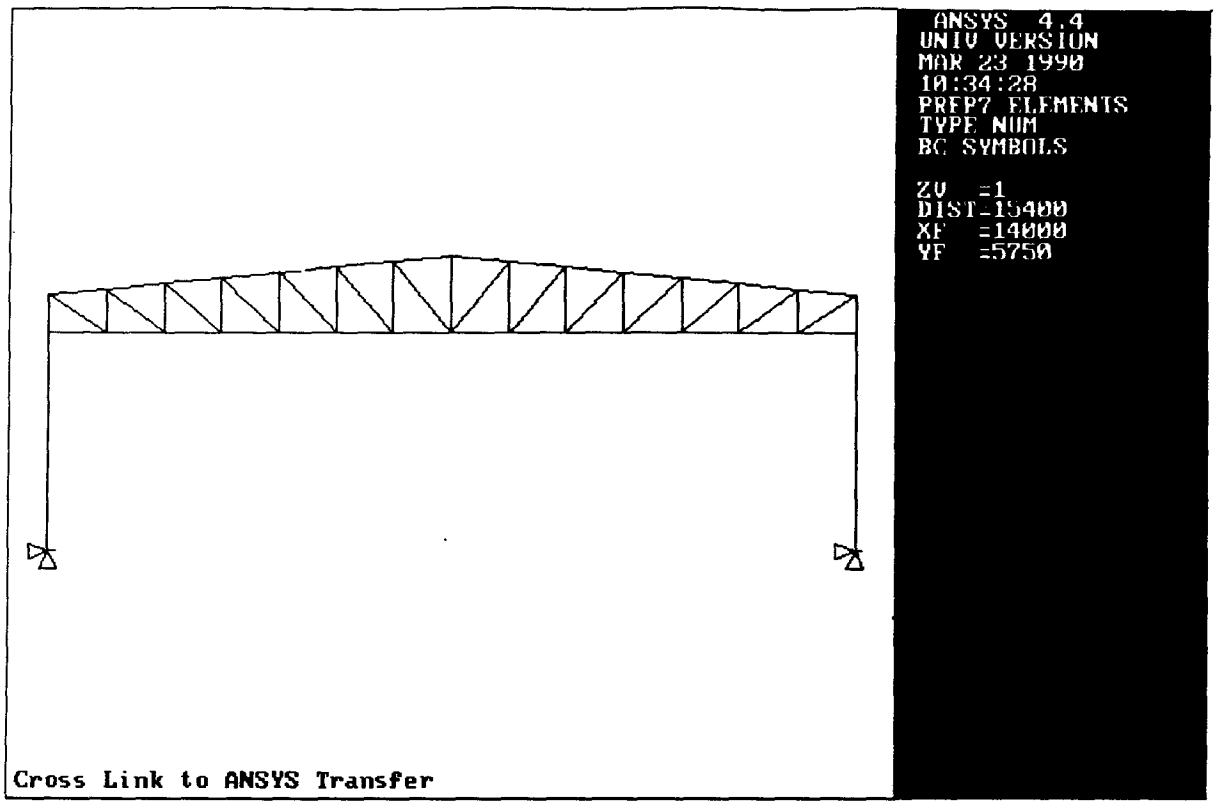


Figure 50.
Example Frame Exported to the ANSYS Programme.

4.5 Extending Cross Link

Several modifications could be made to Cross Link's editing environment and macro language. This section discusses some alterations that could be done to enhance the editing capabilities and also extend the preprocessor to a postprocessor.

Recognition of the Physical and Finite Element Models

Currently, a shortfall with the preprocessor is the fact that analysts still have to describe a structural model in terms of nodes and elements. No feature is yet provided to model a complete structural component such as a continuous beam, as a single entity and still recognize that it is composed of multiple nodes and elements. Structural analysis is performed at a level closer to the finite element model rather than the physical model (Figure 51). Clearly, an alternate data modelling technique is required that recognizes the requirements of the physical model and the finite element model. The physical model is most intuitive for structure description because it explains the structure

in terms of it structural components (ie. beam, column, slab). The finite element model provides a method of discretizing the physical model into data which is then used in forming the mathematical model of the structure.

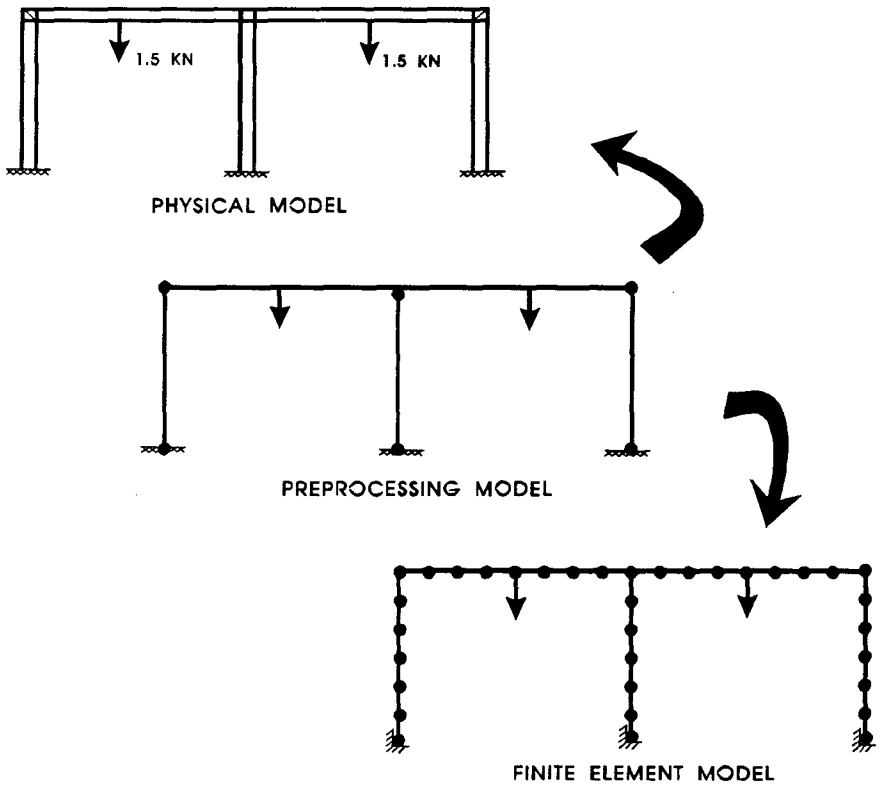


Figure 51.
The Physical and Finite Element Structural Models

To move from the finite element based model and back towards the physical model, modifications have to be made in the way nodes and elements are currently represented. Shown in Figure 52 is an alternate implementation of the element container which has been renamed to an *assembly container,* SO_ASSEMBLY_CONTAINER. The assembly container now holds assembly objects (SO_ASSEMBLY) which are container classes that model both the physical and finite element views of "vector type" elements such as beam and columns or "area type" elements such as slabs and walls. As shown in the figure, the assembly object will always contain, at the least, one element (SO_ELEMENT) which would represent the physical view of the structural component (the beam or column). Finite elements could then be added to the container (as SO_ELEMENT's)

but would be distinguished from the corresponding physical model element. Numerous other objects such as custom components or custom sub-assemblies (which are again container classes) can also be added to the assembly to model loads, boundary conditions or section table objects. Every component and sub-assembly in the assembly container has to behave according to a common message protocol defined by the container. Sub-assembly components could also define a message protocol to communicate object with the sub-assembly container.

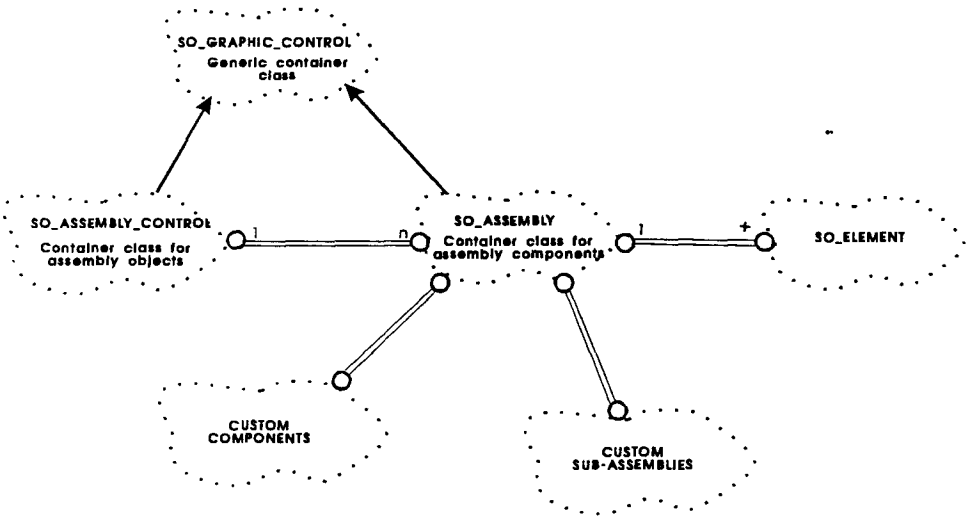


Figure 52.
Alternate Implementation of the Element Container. The element container is now called an assembly container.

For users of the application, most structural modelling would occur on the physical model level. However, a user would also have the ability to enter into a finite element model mode which would allow assemblies to be selected and discretized into finite elements. Returning to the physical model, all finite elements would disappear. An editing operation such as moving an assembly would occur on the physical model level; all finite elements would automatically be moved (so instead of selecting 10 elements representing one continuous beam or assembly, the user would select one assembly).

The concept of an assembly container that holds sub-assemblies and components could ideally serve as an extension of the preprocessor to a design programme (Figure 53). Objects such as connections or web stiffeners could be added to an assembly container to create detailed assemblies. Using polymorphism, generic messages such as *DesignYourself* would be dispatched from the assembly container to all objects. The objects in the container would then respond to a message such as this and proceed to design themselves. Because of the two way interaction between the container and the object, the object would also have the capability to query the container for required design information. For example, a moment connection object may query the assembly object to obtain the design moment and shear for the connection.

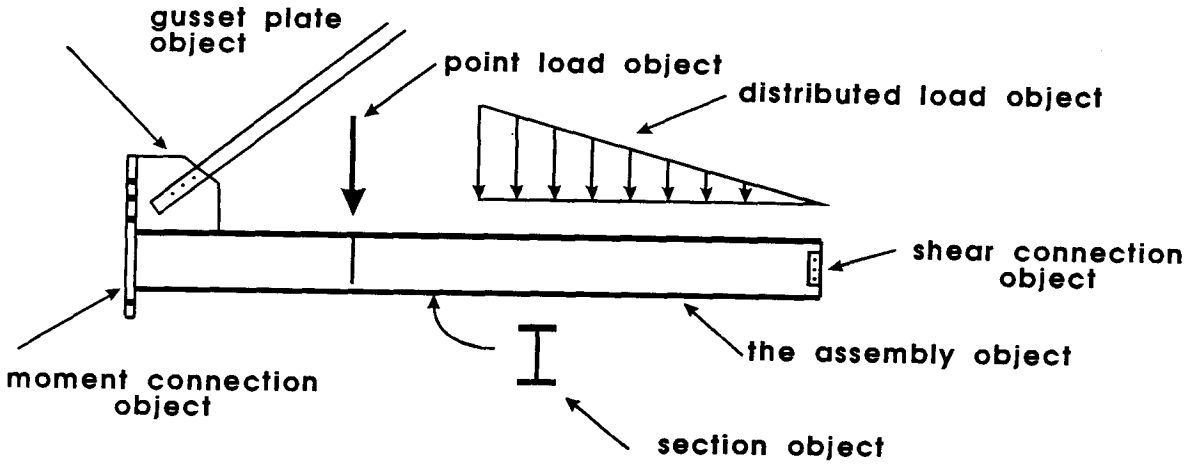


Figure 53.
Extending Assemblies for Postprocessing and Design

The benefits of this approach must be emphasized. Firstly, this model recognizes the needs of the end user of the system - a compromise in modelling capability between the physical and finite element views. Secondly, all assemblies, sub-assemblies and components are developed and debugged as separate entities, each adhering to the interface-implementation concept. This results in a set of objects that should be more resilient to changes made in the system. Finally, assemblies now have a very dynamic appearance because of the ability to modify and enhance behavior simply by adding components. Future objects can easily be implemented as long as they adhere to the message protocol defined by the assembly container.

Extending the Macro Language

Implementation of the aforementioned assembly concept would require modifications to the interface between the macro language and the Cross Link kernel. This is mainly because the containers are now nonhomogenous because they can hold many different types of objects. The existing implementation exploits the fact that the node and element containers are homogeneous. For example, the BOB library functions *ElementFirst()/ElementNext()* (see Appendix A) are used to traverse the element container by initializing and incrementing an internal pointer. Both functions return an integer value representing the current element number.

```
// example function to print all element fixity codes
PrintFixity( ; elementNumber,fixity)
{
     // get the first element
     elementNumber = ElementFirst();
     while(elementNumber){
          fixity = ElementGetFixity(elementNumber);
          print("Element ",elementNumber," -- ",fixity);

          // get the next element in the list
          elementNumber = ElementNext();
     }
}
```

Clearly, the above system used to manipulate the element container could not work adequately in an assembly container based system. Functions such as *ElementFirst()* and *ElementNext()* could still be implemented but the internal representation of the assembly container would not be fully accessible to macro users. Really, a macro user should have the ability to edit the contents of any object in the container. A feature such as this could be implemented using the *class* mechanism supported by the macro language. Pre-built classes could be designed into the language that would allow duplication of the objects data and provide a set of methods within the class to manipulate the data. Additional language keywords could also be created to allow determination of class types (in the existing implementation of the macro language, classes do not know their *type*). For example, an assembly container may be manipulated with a macro in the following way:

```
// get the first object in the container
object = AssemblyFirst()
while(object){

     // compare the object string type
     if(type(object,"Assembly")){
          subObject =  ObjectFirst(object);
          while(subObject){
               // look for element objects
               if(type(subObject,"SO_ELEMENT")){
                    // we've got an element stored in the assembly
                    elementNum = subObject->Number();
                    print("Found element number ",elementNum);
               }
```

```
                else if (type(subObject,"SO_PROPERTY")){
                    // we've get a property object in the assembly
                    Inertia = subObject->GetInertia();
                    Area = subObject->GetArea();
                }
                subObject = ObjectNext(object);
            }
        }

        // get the next object in the container
        object = AssemblyNext();
    }
```

Here, the *ElementFirst()/ElementNext()* functions have been replaced by the *AssemblyFirst()/AssemblyNext()* functions. These functions now return a ***handle*** which now represents a reference to any type of object stored in the container. A special language keyword, *type,* is then used to determine the class type of the object. The functions *ObjectFirst()/ObjectNext()*, which take an object as a parameter, can then be used to determine the contents in an assembly object. The above example looks only for element (SO_ELEMENT) and property (SO_PROPERTY) objects. Once the object's *type* is determined, member functions that are internally attached to the object can be used to manipulate the object.

5 CONCLUSIONS

This thesis has examined the use of object-oriented programming techniques in the development of engineering applications. The paradigm is very well suited to large, complex software applications where issues such as encapsulation, code reusability, portability and maintainability are important. These aforementioned features can be implemented using conventional paradigms such as structured programming, but only through programmer discipline. For example, most procedural languages provide only very limited support in the language syntax for concepts such as information hiding. Conversely, object-oriented languages imbed concepts such as information hiding rules directly into the language syntax and let the compiler provide the enforcement. Consequentially, software becomes more reliable and programmers become more productive because the compiler is now used as a more active tool in the development process.

Cross Link, a universal structural analysis preprocessor, provides an easy to use, interactive editing environment that, through the use of a macro programming language, links bi-directionally with *any* structural analysis programme. The unique macro facility also gives end users the exciting ability to customize the preprocessor to their own requirements, thus providing unprecedented flexibility. Conceivably, the macro language could also be used to provide an electronic link between structural design and drafting. Traditionally, this communication link is done through hand drawn sketches; engineers describe the geometry of a structure in their analysis and design programmes, produce a sketch of the final product and then pass this on to drafting personnel. These people then have to re-enter the information back into a CADD programme for final detailing. Cross Link could be used to provide a smoother, more effective communication link between engineer and draftsman. The link may be simple as a wire frame model or as complex as a fully detailed and dimensioned frame. Features such as these could be implemented using a drawing exchange standard such as IGES or DXF.

6 REFERENCES

[1] Betz, David. *Your Own Tiny Object-Oriented Language*. Dr. Dobb's Journal, September 1991, pp.26-33.

[2] Blair, Gordon. et al. 1991. *Object-Oriented Languages, Systems and Applications.* Reading, London:Pitman Publishing

[3] Booch, Grady. *Object Oriented Development.* IEEE Transactions on Software Engineering, Volume 12, Number 2, February 1986, pp.211-221.

[4] Booch, Grady. 1991. *Object Oriented Design With Applications.* CA:Benjamin Cummings.

[5] Borland International Inc. 1990. *Turbo C++ Programmer's-Guide.*

[6] Coad, Peter. And Yourdon, Edward. 1990. *Object Oriented Analysis.* NJ:Prentice Hall.

[7] Dale, Nell. and Orshalick, David. 1983. *Introduction to PASCAL and Structured Design.* MA:D.C. Heath and Company

[8] Dupont, G. et al. *A Functional and Object-Oriented Design Approach,* Fourth International Conference on Computer Aided Software Engineering, December 1990, pp. 356-372.

[9] Duntemann, Jeff. *Structured Programming.* Dr. Dobb's Journal, May 1990, pp.141-145

[10] Elbury, Kevin M. *Source Code for the Cross Link Universal Structural Analysis Preprocessor.* 1992, University of British Columbia, Vancouver, B.C., Canada.

[11] Hoy, Patrick. *A Comparison of Object-Oriented and Structured Development Methods.* ACM SIGSOFT, Volume 15,Number 1, January 1990, pp.44-48.

[12] LaLonde, Wilf R. and Pugh, John R. *Specialization, Generalization and Inheritance: Teaching Objectives Beyond Data Structures and Data Types*, SIGPLAN Notices, Volume 20, Number 8, August 1985, pp.88-92.

[13] Lee. H.S. and Arora, J.S. *Object-Oriented Programming for Engineering Applications.* Engineering With Computers. Volume 7, 1991, pp.225-235.

[14] Lieberman, Henry. *Using Prototypical Objects to Implement Shared Behaviour in Object Oriented Systems.* Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '86). 1986, pp. 214-223.

[15] Mullin, Mark. 1989. *Object Oriented Program Design.* Reading, MA:Addison-Wesley.

[16] Pascoe, Geoffrey A. *Elements of Object Oriented Programming.* Byte, Volume 11, Number 8. August 1986, pp.139-144.

[17] Snyder, A. *Encapsulation and Inheritance in Object Oriented Programming Languages.* Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '86). 1986, pp. 38-45.

[18] Sommerville, Ian. 1989. *Software Engineering.* Reading, MA:Addison-Wesley.

[19] Stroustrup, Bjarne. 1986. *The C++ Programming Language.* Reading, MA:Addison-Wesley.

[20] Taenzer, David., Murthy, Ganti and Podar, Sunil. *Problems in Object-Oriented Software Reuse.* Proceedings of the Third European Conference on Object-Oriented Programming (ECOOP 89). 1989. pp.25-38

[21] *Zinc Interface Library Programmer's Reference (Version 2.0),* Zinc Software Incorporated, Pleasant Grove, Utah.

APPENDIX A

CROSS LINK MACRO LANGUAGE
FUNCTION REFERENCE

Cross Link

Macro Language

Function Reference

Table of Contents

1 THE CROSS LINK MACRO LANGUAGE

The BOB macro language is a weakly typed, object oriented programming language that users of the Cross Link Preprocessor have access to within an editing session. The uses of the language are limitless: importing/exporting data files, manipulating the Cross Link node and element databases, and parametric structure definition to name a few.

BOB has several features that make it unique as a programming language. It contains flavours of C, C++ and LISP although its syntax is almost identical that of C. Being object oriented, it supports class definitions and inheritance. In a weakly typed language, basic data types such as integer numbers, floating point numbers and character strings are never explicitly *typed* by the programmer as integers, floats, or strings. This is advantageous since it makes the language more flexible and easier to use.

1.1 Basics

The basic program unit inside a BOB program is the function. Every BOB program requires a function called *main()* that must exist somewhere in the program file. A functions can take any number of parameters and perform operations on those parameters. The function may directly modify the values passed into it or it may only use them for calculations. Functions also have the ability to return a value. Hence a call to a function usually appears as an assignment.

```
main( ; x, x_squared)
{
     x = 5;
     4x_squared = Square(x);   // call the Square() function
}


// Function: Square(z)
// - returns the square of z
Square(z ; tmp)   // note x is a parmeter, tmp is a local variable
{
     tmp = z * z;    // calculate the square...
     return(tmp);
}
```

Figure 1.
A BOB program to calculate the square of a number. The function Square does the calculation.

From the example program in Figure 1, several observations can be made:

- The function *main()* has two local variables, *x* and *x_squared* which are declared after the semicolon in declaration of *main()*.
- Any function must enclose all of its associated code in curly braces {}.
- A semicolon (;) ends every line in which there is an assignment.
- The *Square()* function takes one parameter, *x* (declared as *z*).
- The Square() function has one local variable, *tmp,* declared after the semicolon in the function definition.
- The Square() function returns the value of *x* * *x*.
- Program comments are place by preceding the comment with two slashes (//).

1.2 Variables and Expressions

Data Types

As mentioned in the introduction, BOB is an untyped language. Thus as a user of BOB, you are not required to explictly declare the type of any variable. However, you must be aware of the basic internal representation of data types in order to use the language correctly. When a variable is declared, its default data type is type *nil.* Only when an assignment is made to the variable, is its type set.

Shown in Figure 2 is an example program emphasising BOB's internal treadment of data types. From this example we can make the following observations:

- Variables are of type *nil* before they they are used.
- In Part A, *i* and *j* are typed as integers because no decimal point appears in the number 2 or 5. The result of the integer division, 5 / 2, is 2.

```
main( ;i,j,k)
{
     // Part A: Division using integers
     i = 5;              // represented as integer
     j = 2;
     k = i / j;          // result is 2

     // Part B: Dividision using floats (deimals)
     i = 5.0;       // represented as float
     j = 2.;             // also represents as float
     k = i / j;          // result is k = 2.5
}
```

Figure 2.
A BOB program used to compare calculations using integers and floating point (decimal) numbers.

- In Part B, the variables *i* and *j* change type from integer to floating point (legal in BOB) as indicated by the **decimal point** (2.0). Here the result is different: 5.0 / 2.0 = 2.5.
- If the division involved a floating point number (ie i = 5.0) and an integer (ie. j = 2.0) the result would a float (k = 2.5).
- The above rules are applicable to all other basic mathematical operations.

Expressions

Listed in Figure 3 are the basic expressions supported by BOB. Since all of these expression are a subset of the C programming language, instructions and examples on their use can be found in any C programming reference (ignore anything to do with *pointers*).

```
<expression>,<expression>                    comma expression

<lvalue> = <expression>                      assignment
<lvalue> += <expression>                     assign sum
<lvalue> -= <expression>                     assign difference
<lvalue> *= <expression>                     assign product
<lvalue> /= <expression>                     assign quotient

<test-expression> ? <true-expression> : <false-expression>
      conditional operator

<expression> || <expression>                 logical OR
<expression> && <expression>                 logical AND
<expression> | <expression>                  bitwise OR
<expression> ^ <expression>                  bitwise XOR
<expression> & <expression>                  bitwise AND
<expression> == <expression>                 equal to
<expression> != <expression>                 not equal to
<expression> < <expression>                  less than
<expression> <= <expression>                 less than or equal to
<expression> >= <expression>                 greater than or equal to
<expression> > <expression>                  greater than
<expression> << <expression>                 shift left
<expression> >> <expression>                 shift right

<expression> + <expression>                  plus
<expression> - <expression>                  minus
<expression> * <expression>                  multiply
<expression> / <expression>                  divide
<expression> % <expression>                  remainder

-<expression>                                unary minus
!<expression>                                logical negation (NOT)
~<expression>                                bitwise negation
++<expression>                               preincrement
--<expression>                               predecrement
<lvalue>++                                   postincrement
<lvalue>--                                   postdecrement
```

Figure 3.
Operations supported by BOB. An <lvalue> is a left hand side value meaning the variable must be able to have the result of an expression assigned to it.

1.3 Program Control Statements

BOB contains all the program control statements of the C language (except for the *switch* statement). Listed in Figure 4 are the four program control statements used by BOB.

```
if( <test-expression> )
     <then-statement>
[else
     <else-statement> ]

while(<test-expression>)
     <body-statement>

do <body-statement>
     while(test-expression);

for(<init-expression>;<test-expression>;<increment-expression>)
    <body-expression>

break;
continue;

return(<result-expression>);
```

Figure 4.
Program control statements supported by BOB.

1.4 Objects and Classes

BOB supports class definitions and single inheritance. A class can be used to combine both data (called member data) and the functions (called member functions) that operate on that data into a single entity. Generally a complete class system includes two components. The first is the class template which defines the member data and the member functions of the class. Using inheritance, classes are able to *inherit* the member data and functions of other classes in order to extend or modify them. The second component of a class system is the implementation or definition of the member functions. These are coded outside of the class template; they look like regular functions except the function name is preceded by the class to which the function belongs. However, member data appears global to the member functions so member data is never passes as an argument to a member functions. The syntax for class definition is shown in Figure 5.

The class definition serves as a template for the physical definition of an object; an object is an instance of a class. An object is instantiated using the *new* operator. Member functions within the object are then referenced using the -> operator. However, all member data in a class definition is private so it cannot be referenced using the -> operator. Therefore, member data is typically referenced by users of the object via members functions that set or retrieve the value of member data. Figure 6 shows the syntax for creating objects and accessing them.

```
Class Definition:
class <class-name> [: <base-class-name>]
{
     <member-definition>
     <member-definition>
}

     where <member-definition> is any/all of the following:
     <variable-name>;
     static <variable-name>;
     <function-name>([formal-argument-list>]);
     static <function-name>([<formal-argument-list>] );


Member Function Definition:
<class-name>::<member-function>([<argument-list> ; <temporary-argument-list>])
{
     <statements>
}
```

Figure 5.
Class definition syntax.

```
Object Creation:
<object-name> = new <class-name>([<constructor-arguments>]);

Object Access:
<object-name>-><function-name>([<arguments>])
<object-name>-><data-name>;
```

Figure 6.
Object creation and access syntax.

Every object usually has a *constructor* as one of its member functions. Constructors have the same name as the class and are used for initialization on any member data. They are automatically called when an object is instantiated using the *new* keyword. Figure 7 shows an example implementation of a class system to represent a point in space.

```
// The Point class definition
class Point
{
     x,y;            // the coordinates of the point

     Point(x,y);          // Constructor
     Move(dx,dy);    // move the point

     GetX();              // get the data
     GetY();
}
```

```
// Constructor
Point::Point(a_x,a_y)
{
     x = a_x;
     y = a_y;
}

// move the point a distance (dx,dy)
Point::Move(dx,dy)
{
     x += dx;
     y += dy;
}

// get the x and y coordinates of point
Point::GetX()
{
     return x;
}

Point::GetY()
{
     return y;
}


// Example implementation of a Point class
main( ; aPoint, x)
{
     // instantiate a Point object at (0,0)
     aPoint = new Point(0.0,0.0);  // constructor is automatically called

     // offset the point a distance of (10.0,10.0);
     aPoint->Move(10.0,10.0);
     .
     .
     // inquire about its coordinates
     x = aPoint->GetX();

}
```

Figure 7.
Example implementation of class Point.

2 CROSS LINK PROGRAMMER'S REFERENCE

2.1 Programming Constants

Several constants are pre-coded into to the system. These are listed in the following table:

Name	Value	Comments
TRUE FALSE	0 1	Generally used as a return value from some functions.
PI	3.14159265	
ID_000 ID_001 ID_011 ID_111 ID_101 ID_100 ID_010	N/A	These are node degree of freedom codes used by NodeGetDOF(), NodeSetDOF() and NodePut()
ID_FF ID_PP ID_FP ID_PF	N/A	These are element fixity codes used by ElementGetFixity(), ElementSetFixity(), and ElementPut().

2.2 Input / Output Functions

---

fclose()

Synopsis

fclose(file)

file - file handle

Remarks

fclose() closes the file associated with the file handle *file*.

Example

```
outputfile = fopen("test.dat","rb");
.
.
fclose(outputfile);
```

fopen()

Synopsis

fopen(fname,mode)

fname - the file name
mode - the access mode for the file

Remarks

Use fopen to perform I/O to a file. fopen() returns the handle to the file *fname* opened. Valid *modes* are any combination of the following access modes and file translation modes:

Access Modes for File I/O

String	Interpretation
"r"	Open file for read only operations
"w"	Open a new file for writing. Existing contents will be overwritten.
"a"	Open a file for appending.
"r+"	Open an existing file for read and write operations. Error it the file does not exist.
"w+"	Create a file and open it for reading and writing.
"a+"	Create a file and open it for reading and writing.

File Translation Modes for File I/O

Mode	Interpretation
"b"	File is opened in binary mode. Every character is read as is without the changes described below.
"t"	File is opened in translated mode subject to the following: 1. Carriage Return-Line Feed combinations on input are translated to single linefeeds. 2. During input, the Ctrl-Z character is interpreted as the EOF character.

Example

```
// open test.dat for writing to
outfile = fopen("test.dat","w");
fwrite(outfile,"This file is called test.dat");
```

---

fread()

Synopsis

fread(file)

file- file handle (see fopen())

Remarks

Use fread to read a line from *file*. The line is read until a newline character (\n) is encountered. The maximum line length is limited to 255 characters.

If there are no errors, fread() returns the the string read; otherwise it returns *nil*.

Example

```
// program to print the file test.dat to the watch window
main( ; file, string)
{
     // open test.dat for reading
     file = fopen("test.dat","r");

     // read the first line
     string = fread(file);

     while(string){
          print(string);
          string = fread(file);
     }
}
```

fwrite()

Synopsis

fwrite(file,...)

file- the file handle (see fopen())

Remarks

Use fwrite to write data to a file. Numeric values are automatically padded with a blank space on the left hand side. Floating point numbers are output to four decimal point accuracy.

Additional characters can be used to control output:

\n - new line character
\t - insert a tab character

Example

```
// open test.dat for writing to
outfile = fopen("test.dat","w");
// write some text and add to line feeds
fwrite(outfile,"\tThis file is called test.dat\n\n");
```

getc()

Synopsis

getc(fhandle)

fhandle - the file handle

Remarks

Use *getc()* to read the next character from the file associated with *fhandle.* The function returns -1 when the end of file is reached.

Example

```
// read a character from file
char = getc(file);
print("Read the char: ",char," from file.");
```

putc()

Synopsis

putc(char,fhandle)

Remarks

Use *putc()* to write the character *char* to the file associated with *fhandle*.

Example

```
// write a character to file
putc("a",file);
```

print()

Synopsis

print(...)

Remarks

Use *print()* to print information to the Watch Window inside Cross-Link. Print takes any number of data types and is capable of distinguishing between different data types. This function is useful for debugging and displaying information to the user.

Example

```
print(ncount," nodes selected");
```

2.3 Math Functions

---

abs()

Synopsis

abs(number)

number - the number

Remarks

abs() returns the absoulute value of *number.*

Example

```
value = -10.3454
value = abs(value);  // value = 10.3454
```

---

acos() / asin() / atan()

Synopsis

acos(x)
asin(x)
atan(x)

Remarks

Use *acos(), asin(), atan()* to compute the arc cosine, arc sine and arc tangent of argument *x.* For *acos()* and *asin(),* x must be between -1 and 1. The returned angle is in RADIANS.

Example

```
angle = acos(0.5);  // angle = PI/3
```

atan2()

Synopsis

atan2(y,x)

Remarks

atan2() returns the arc tangent of *y/x* in radians.

Example

```
angle = atan2(y,x);
```

cos() / sin() / tan()

Synopsis

cos(theta)
sin(theta)
tan(theta)

Remarks

Use *cos(), sin(),* and *tan()* to compute the cosine, sine and tangent of angle *theta* (in RADIANS)

Example

```
val = cos(PI); // val = -1
```

exp()

Synopsis

exp(x)

Remarks

Use exp() to calculate the exponential of x.

Example

```
eval = exp(10.345);
```

hypot()

Synopsis

hypot(x,y)

Remarks

Use *hypot()* to compute the length of the hypotenuse of a right angle triangle given the length of both sides, *x* and *y*.

Example

```
length = hypot(3.0,4.0);   // length = 5.0
```

log() / log10()

Synopsis

log(x)
log10(x)

Remarks

Use *log()* and *log10()* to calculate the natural logarithm and logarithm to the base 10 respectively of argument *x*.

Example

```
y = log(2.0);  // t = 0.693147
```

pow() / pow10()

Synopsis

pow(x,y)
pow10(z)

Remarks

Use *pow()* to calculate the value of *x* raised to the power of *y*. Use *pow10()* to raise 10 to the power of *z*.

Example

```
x = pow10(2);   // x = 100.0
y = pow(2,4);   // y = 16.0
```

sqrt()

Synopsis

sqrt(x)

Remarks

Use *sqrt()* to compute the square root of value *x*. If x is less than zero an error will occur;

Example

```
y = sqrt(abs(-2.0);   // y = 1.414
```

2.4 String Functions

---

strcmp()

Synopsis

strcmp(string1,string2)
strcmp(string1,string2,n)

string1	- the strings to be compared
string2	
n	- number of characters to compare

Remarks

The first function compares *string1* and *string2* lexigraphically. The second function compares the first *n* characters of *string1* and *string2*. Both functions are case sensitive.

strcmp() returns a number less than, equal to, or greater than 0, depending on whether *string1* is less than, equal to, or greater than *string2*, respectively.

Example

```
if(strcmp(token,"Node") == 0)
     print("The token is a 'Node'");

if(strcmp("Kevin Elbury","KevinElbury",5) == 0)
     print("First 5 characters are the same!);
```

token()

Synopsis

token(string,delimiter)

string	- the number
delimiter	- a string containing the delimiting characters

Remarks

Use the token() function to parse the records out of a delimited string. The first argument, *string*, is a character string containing all the delimited tokens. The second string, *delimiter*, is a string describing the set of characters that delimit the tokens.

In general the *token()* function must be called several times to completely parse *string*. The first call to token() includes the complete string that requires parsing. Subsequent call to *token()* must pass 0 as the first parameter to indicate that parsing is still to be done on the old string. However, every call still requires the *delimiter* string (which can be changed between calls).

token() returns *nil* when there are no more records to parse.

Example

```
// parse each of the records from a string containing the
// following format (ANSYS Node command):
//              N,node_num,x,y,z
//
delimiter = ",";          // delimiter is a comma
string = "N,12,123.4,232.23,0.0";

// initialize the token parser
strtok = token(string,delimiter);

// check the first token to see if it is a node string
if(strcmp(strtok,"N"){
     // get the node number
     strtok = token(0,delimiter);
     num = val(strtok);

     // get the x, y and z coordinates
     strtok = token(0,delimiters);
     x = val(strtok);

     strtok = token(0,delimiters);
     y = val(strtok);

     strtok = token(0,delimiters);
     z = val(strtok);
}
```

val()

Synopsis

val(string)

string - the number in string form

Remarks

Use the *val()* function to convert *string* to a numeric value. If string contains any non-numeric characters, *val()* returns *nil,* otherwise *val()* returns the numeric value.

Example

```
number = val("123.23");          // number = 123.23
number = val("ab123");           // number = nil
```

2.5 Cross Link System Functions

SetGrid

Synopsis

SetGrid(grid_x, grid_y, x_origin, y_origin, snapflag, gridflag)

grid_x	- the x direction grid spacing
grid_y	- the y direction grid spacing
x_origin	- the x_origin of the grid
y_origin	- the y_origin of the grid
snapflag	- snap-to-grid flag (TRUE/FALSE)
gridflag	- grid display flag (TRUE/FALSE)

Remarks

The SetGrid() function is used to set the grid parameters.

Example

```
// set up grid with snaps off and grid display off
SetLimits(0,0,15000,10000);
snapflag = FALSE;
gridflag = TRUE;
SetGrid(1000,1000,0,0,snapflag,gridflag);
```

New()

Synopsis

New()

Remarks

Use *New()* to reset a structure by erasing all nodes and elements. Any work not saved will be lost.

Example

```
main()
{
     // start with a clean structure
     New();
     SetLimits(0,0,15000,1000);
     .
     .
}
```

Prompt()

Synopsis

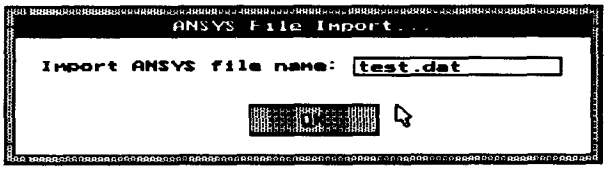
Prompt(title,prompt,value)

title - the title for the dialogue box
prompt - the prompt string placed within the box
value - the initial value to be edited (integer, float or string)

Remarks

Prompt() opens a dialog box centered on the screen. The diaglog box consists of a title bar with *title*, a *prompt* string, a *value* editor and an OK button. The parameter *value* must be initialized (type nil is not supported) with the type of value you wish to edit (ie. string, integer or decimal number) before it is passed into this function.

Prompt() returns the new value entered by the user.



Example

```
main()
{

     filename = "test.dat"
     string = Prompt("ANSYS File Import... ","Import ANSYS filename: ",filename);
     .
     .
     // edit an integer value
     intval = 1;
     intval = Prompt("Example Integer Editor","Enter an integer: ",intval);
     .
     .
}
```

PromptYesNo()

Synopsis

PromptYesNo(title,prompt)

title - the title for the dialogue box
prompt - the prompt string placed within the box

Remarks

PromptYesNo() opens a dialog box to retrieve a Yes or No answer from the user. The diaglog box consists of a title bar with *title*, a *prompt* string, a *Yes* button and a *No* button.

PromptYesNo() returns the TRUE if the Yes button was pressed or FALSE if the No button was pressed.

Example

```
main()
{
     // do we want to run the macro with a new structure?
     ans = PromptYesNo("ANSYS File Inport...","Replace existing structure?");
     if(ans == TRUE)
          New();               // erase the existing structure...
     .
     .
}
```

Repaint()

Synopsis

Repaint()

Remarks

Repaint() redisplays the contents of the structure database. Only items selected in the Settings... Display will be displayed.

Example

```
// add a node and redisplay the structure
NodePut(NodeHigh(),1000,2000,ID_000);
Redisplay();
```

SetLimits()

Synopsis

SetLimits(left,bottom,right,top);

Remarks

Use *SetLimits()* to set the current structure limits. The limits are defined as the *minimum* area required to fit a structure on the screen. Cross Link will provide a viewport with at least the specified limits. However, in order to maintain an aspect ratio of 1:1 the x or y limits may need to be internally adjusted. These adjusted limits are referred to as *'extents'* and are stored separately from the structure limits. Note that grids will only be drawn on the specified limits.

Example

```
// define limits big enough to hold a 10m wide by 20m high structure
SetLimits(-1000,-1000,12000,22000);
```

Speaker()

Synopsis

Speaker()

Remarks

Use *Speaker()* to sound the speaker on your computer. This is useful for debugging programs

Example

```
// select a node - make some noise if we can't find if
err = NodeSelect(num);
if(err == nil){
     Speaker();
     print("Can't find node ",num);
}
```

2.6 Node Functions

NodeErase()

Synopsis

NodeErase()
NodeErase(num)

num - node number to erase

Remarks

The first function removes all *selected* nodes from the node database. The second function erases node *num* from the node database. Both functions will additionally remove any point loads, springs and elements attached to the node.

Example

```
// erase the last node in the database
num = NodeHigh() - 1;
retval = NodeErase(num);
if(retval == nil)
     print("Node number ",num,"does not exist!");
else
     print("Node number ",num, "deleted");
```

NodeFirst() / NodeNext()

Synopsis

NodeFirst()
NodeNext()

Remarks

NodeFirst() returns the number of the first node in the node database. It is generally used in conjunction with the NodeNext() function to traverse the node database and perform operations on individual nodes. NodeFirst() returns nil if there are no nodes in the node database.

Example

```
// function to erase nodes that are NOT selected
MyErase(,num)
{
     num = NodeFirst();    // get the first node in the database
     while(num != nil){
          if(!NodeSelected(num))
               NodeErase(num);
          // get the next 'num' in the database
          num = NodeNext();
     }
}
```

NodeGetDOF() / NodeSetDOF()

Synopsis

NodeGetDOF(num)
NodeSetDOF(num,dof)

num - node number to get
dof - node dof code

Remarks

NodeGetDOF() returns the degree of freedom code for node *num.*
NodeSetDOF() sets the degree of freedom of node *num.* See *NodePut()* for a description of these codes.

Example

```
main()
{
.
.
// change all pinned nodes to fixed nodes
ModifyDOF(ID_110,ID_111);
.
.
}


// function to change all nodes with dof 'old_dof' with 'new_dof
ModifyDOF(old_dof, new_dof; num, dof)
{
     num = NodeFirst();
     while(num != nil){
          if(NodeGetDof(num) == old_dof)
               NodeSetDof(num,new_dof);
          num = NodeNext(); // proceed to next number
     }
}
```

NodeGetX() / NodeGetY()

Synopsis

NodeGetX(num)
NodeGetY(num)

num - node to get

Remarks

NodeGet?() returns the x or y coordinate of node *num* or returns *nil* if node *num* does not exist.

Example

```
// function to calculate the distance between two nodes
Length(num1,num2; dx, dy)
{
     dx = NodeGetX(num1) - NodeGetX(num2);
     dy = NodeGetY(num1) - NodeGetY(num2);
     length = hypot(dx,dy);
     return(length);
}
```

NodeHigh()

Synopsis

NodeHigh()

Remarks

NodeHigh() returns the number of the next available node number.

Example

```
// add a node but find out the next number we can use
num = NodeHigh();
NodePut(num,100,1200,ID_000);
```

NodePut()

Synopsis

NodePut(num,x,y,dof)

num - the node number
x,y - the x and y coordinates
dof - the dof code number

Remarks

NodePut places a node at the specified x,y coordinates with a dof degrees of freedom. Allowable dof codes are global variables set by the system. A code is specified by 'ID_' along with the x,y and rotational degrees of freedom represented by ones and zeros with 1 = free and 0 = fixed. Allowable codes are:

ID_000 - free joint
ID_001 - 0 rotation joint
ID_011
ID_111 - fully fixed joint
ID_100
ID_110 - pinned joint
ID_101

If node num already exists in the node database, Cross Link will update the existing node parameters with the supplied values.

Example

```
// generate a parabolic line of nodes
k = 0.00005;
for(x=0.0, x <= 10000; x += 500){    // space nodes at 500
    num = NodeHigh();
    y = k * x * x;
    NodePut(num,x,y,ID_000);        // ID_000 is a system variable
}
```

NodeSelect()

Synopsis

NodeSelect(num)
NodeSelect(left,bottom,right,top)

Remarks

The first function selects node *num*. The second function selects ALL nodes defined by a box with coordinates *(left,bottom)* to *(right,top)*. NodeSelect() returns the number of nodes selected. Use the NodeSelected() function to determine if a node is selected.

Example

```
// select all nodes within (100,100) to (500,800)
count = NodeSelect(100,100,500,1000);
print(count," nodes selected");
```

NodeSelected()

Synopsis

NodeSelected(num)

num - node number to inquire about

Remarks

NodeSelected returns TRUE if node *num* is selected and FALSE if node *num* is not selected.

Example

```
if(NodeSelected(num) == TRUE)
     print("Node ",num, "is selected");
```

NodeSetX() / NodeSetY()

Synopsis

NodeSetX(num,x_coordinate);
NodeSetY(num,y_coordinate);

Remarks

NodeSetX and NodeSetY are used to set the x and y coordinates of *num* respectively. Both return TRUE if succesful and FALSE if not.

Example

```
// set the x coordinate on node 12
num = 12;
ret = NodeSetX(num,10500.0);
```

NodeToggleSelect()

Synopsis

NodeToggleSelect()

Remarks

NodeToggleSelect() runs through the whole node database and toggles selected and unselected nodes.

Example

```
// delete all nodes except for number 5 and 7
NodeSelect(5);
NodeSelect(7);
NodeToggleSelect(); // flip selected/unselected nodes
NodeErase();
```

NodeUnSelect()

Synopsis

NodeUnSelect()

Remarks

NodeUnSelect() unselects all selected nodes in the node database.

Example

```
// unselect all unselecded nodes
NodeUnSelect();
```

2.7 Element Functions

ElementErase()

Synopsis

ElementErase()
ElementErase(num)

Remarks

The first function erases all **selected** elements. The second function erases element *num*. The return value for this function is TRUE if successful of FALSE if *num* does not exist.

Example

```
// erase all selected elements
ret = ElementErase();
```

ElementFirst() / ElementNext()

Synopsis

ElementFirst()
ElementNext()

Remarks

ElementFirst() returns the number of the first node in the node database. It is generally used in conjunction with the ElementNext() function to traverse the element database and perform operations on individual elements. Both functions return *nil* when there are no more elements in the element database.

Example

```
// traverse the element database
element = ElementFirst();
while(element){
     // perform any required operations on each element
     ...

     element = ElementNext();
}
```

ElementGetFixity() / ElementSetFixity()

Synopsis

ElementGetFixity(num)

ElementSetFixity(fix_code)
ElementSetFixity(num,fix_code)

Remarks

ElementGetFixity returns the fixity code for element *num*.
ElementSetFixity sets the fixity code for an element. If the first is used, all *selected* elements are affected. If the second is used, only element *num* is affected (if it exists).

See ElementPut() for list of *fix_code*'s.

Example

```
// set all selected elements to pin-pin
ElementSetFixity(ID_PP);
```

ElementGetLoNode() / ElementGetHiNode()

Synopsis

ElementGetLoNode(el_num)
ElementGetHiNode(el_num)

Remarks

ElementGetLoNode returns the lower node number of element *el_num*.
ElementGetHiNode returns the higher node number of element *el_num*.
Both return *nil* is *el_num* does not exist.

Example

```
// function to erase all elements that are attached to node 'num'
Spc_Erase(num; el_num,lonode,hinode)
{
     // run through the element database
     el_num = ElementFirst();
     while(el_num != nil){
          lonode = ElementGetLoNode(el_num);
          hinode = ElementGetHiNode(el_num);

          // do we have a match??
          if(lonode == num)
               ElementErase(el_num);
          else if(hinode == num)
               ElementErase(el_num);

          // get the next element
          el_num = ElementNext();
     }
}
```

ElementSetProp()

Synopsis

ElementSetProp(A,Av,I)
ElementSetProp(el_num,A,Av,I)

el_num	- the element number
A	- the Area of element *el_num* (returned)
Av	- the Shear Area of element *el_num* (returned)
I	- the Moment of Inertia of *el_num* (returned)

Remarks

ElementSetProp() sets the sectional properties for the element. The first function set the section properties of all *selected* elements. The second function sets the section properties for element *el_num*.

Example

```
ElementSetProp(el_num,A,Av,I);
```

ElementHigh()

Synopsis

ElementHigh()

Remarks

ElementHigh() returns the number of the next available element number.

Example

```
// add an element but find out the next number we can use
num = ElementHigh();
ElementPut(num,lonode,hinode,ID_FF);
```

ElementPut()

Synopsis

ElementPut(el_num,lonode,hinode,code)

el_num	- reference number for the element
lonode	- the lower node number
hinode	- the high node number
code	- the fixity code

Remarks

ElementPut add an element to the element database. The *lonode* and *hinode* numbers must already exist. The fixity codes precode into the system as follows:

ID_PP- lonode Pinned, hinode Pinned
ID_FP- lonode Fixed, hinode Pinned
ID_PF- lonode Pinned, hinode Fixed
ID_FF- lonode Fixed, hinode Fixed

Example

```
// generate parabolic line of nodes and elements
Parabola(dx,dy,num1,num2,elnum)
{
     num1 = NodeHigh();

     // place the starting node
     NodePut(num1,0,0,ID_000);

     // generate the line at increments of 1000
     for(dx=1000.0; dx <= 11000.0; dx += 1000.){
          num2 = NodeHigh();
          dy = 0.00008 * dx * dx;
          NodePut(num2,dx,dy,ID_000);

          // get the next available element number and add the element
          elnum = ElementHigh();
          ElementPut(elnum,num1,num2,ID_FF);
          num1 = num2;
     }
}
```

ElementSelect()

Synopsis

ElementSelect(el_num)
ElementSelect(left,bottom,right,top)

Remarks

ElementSelect selects an element. The first function selects an element by its number, *el_num*. The second function selects any elements inside a box with coordinates *(left,bottom)* to *(right,top)*.

Example

```
ElementSelect(1000,1000,5000,10000);
```

ElementSelected()

Synopsis

ElementSelected(el_num)

Remarks

ElementSelected returns *TRUE* if if element *el_num* is selected of *FALSE* otherwise.

Example

```
// see if element 12 is selected
if(ElementSelected(12) == TRUE)
     print("Element 12 is selected");
else
     print("Element 12 is NOT selected");
```

ElementToggleSelect()

Synopsis

ElementToggleSelect()

Remarks

ElementToggleSelect toggles selected and unselected elements in the element database.

Example

```
// erase element all UNselected elements using ToggleSelect()
ElementToggleSelect();
ElementErase();
```

ElementUnSelect()

Synopsis

ElementUnSelect()

Remarks

ElementUnSelect unselects all selected elements in the element database.

Example

```
// unselect all the elements
ElementUnSelect();
```

3 EXAMPLE APPLICATIONS

Following is the BOB source code for two sample applications

3.1 Demonstration Program

```
// ---------------------------------------------------------------
//  File:          DEMO.BOB
//  Description:   Demo of the BOB programming Language
//  Written By:    KME
//  Date:          Decenber 15, 1991
//
//  Copyright (c) 1992, by Kevin Michael Elbury
// ---------------------------------------------------------------
main()
{
     // start with a clean structure
     New();
     print("Cross Link Demo Program - (demo.bob)");
     // set up the grid and limits
     SetGrid(1000,1000,0,0,TRUE,TRUE);
     SetLimits(-1000,-1000,15000,12000);
     // perfom the demostration
     Circle(3000.,5000.,3000.,20);
     SineWave(8000,5000,14000,5000,2000);
     // update the display
     Repaint();
}

// Circle()
// %% Creates a circle out of node and elements with origin at x,y
//
Circle(x,y,radius,points; i,num1,num2,elnum,dx,dy,dt,theta,start)
{
     start = num1 = NodeHigh();         // rememeber the first node
     num2 = 0;
     i=1;
     theta = dt = 2.0 * PI / points;
     NodePut(num1,x+radius,y,ID_000);
     while(i<points){
          theta = dt * i;
          dx = x + radius * cos(theta);
          dy = y + 1.30 * radius * sin(theta);
          i++;
          num2 = NodeHigh();
          NodePut(num2,dx,dy,ID_000);
          // get the next element number
          elnum = ElementHigh();
          ElementPut(elnum,num1,num2,ID_FF);
          num1 = num2;
     }

     // add the element
     ElementPut(ElementHigh(),num2,start,ID_FF);
     Repaint();
}
```

```
// SineWave()
// %% Creates a sine wave out of node and elements with origin at xo,y0
//    and ending at xf,yf. Amplitude of wave is A.
//
SineWave(xo,yo,xf,yf,A;num1,num2,spc,w,i,di,dx,dy)
{
     num1 = num2 = 0;
     di = 80;
     spc = (xf - xo)/ di;
     w =  12 * PI / (xf-xo);
     num1 = NodeHigh();
     NodePut(num1,xo,yo,ID_000);
     i = 1;
     dx = xo;
     while(dx < xf){
          dx = xo + i * spc;
          dy = yo + A * sin(w * i * spc);
          i++;
          num2 = NodeHigh();
          NodePut(num2,dx,dy,ID_000);
          elnum = ElementHigh();
          ElementPut(elnum,num1,num2,ID_FF);
          num1 = num2;
     }
}
```

3.2 ANSYS File Transfers

ANS-IN.BOB

The following program provides limited file compatability with the ANSYS finite element program.

```
// ---------------------------------------------------------------
//  File:          ANS-IN.BOB
//  Description:   Imports an ANSYS script file into Cross Link
//  Written By:    KME
//  Date:          January 14, 1992
//
//  Copyright (c) 1992, by Kevin Michael Elbury
// ---------------------------------------------------------------
main(; filename,line,tok,dx,dy)
{
     // set this to the ANSYS file you wish to import
     filename = "test.dat";
     delimit = ","; // field delimiter is a comma - global var
     file = fopen(filename,"r");
     // read in first line and get the first token in the string...
     line = fread(file);
     tok = token(line,delimit);
     max_x = max_y = 3.4 * pow10(-38.0);   // smallest floating point number possible
     min_x = min_y = 3.4 * pow10(38.0);    // largest  ...
     // compare first token
     while(line){
          if(strcmp(tok,"N") == 0)
               readNode();
          else if(strcmp(tok,"E") == 0)
               readElement();
          else if(strcmp(tok,"D") == 0)
               readBoundaryCond();
          else
               print("Ignoring command: ",tok);
          // read and tokenize next line
          line = fread(file);
          if(line)
               tok = token(line,delimit);
   };
     // add 30% margin to limits
     dx = max_x - min_x;
     dy = max_y - min_y;
     dx *= 0.30;
     dy *= 0.30;
     min_x -= dx;
     max_x += dx;
     min_y -= dy;
     max_y += dy;
     SetLimits(min_x,min_y,max_x,max_y);
     Repaint();
}

// readNode()
// %% continues parsing the ANSYS 'N'ode command and adds the node
//    to the node database
//
// Format:
//     N,node_num,x_coord,y_coord
//
```

```
readNode( ;num,x,y)
{
     // get the next token string and convert it to a number
     num = val(token(0,delimit));
     x = val(token(0,delimit));
     y = val(token(0,delimit));
     // add the node to the data base
     NodePut(num,x,y,ID_000);
     // evaluate structure limits
     max_x = (x > max_x ? x : max_x);
     max_y = (y > max_y ? y : max_y);
     min_x = (x < min_x ? x : min_x);
     min_y = (y < min_y ? y : min_y);
}

// readElement()
// %% continues parsing the ANSYS 'E'lement command
//
// Format:
//     E,lonode_num,hinode_num
//
readElement( ;lonode,hinode,num)
{
     // get the next token string and convert it to a number
     lonode = val(token(0,delimit));
     hinode = val(token(0,delimit));
     num = ElementHigh();
     // add the node to the data base
     ElementPut(num,lonode,hinode,ID_FF);
}

// readBoundaryCond()
// %% Parses the ANSYS 'D'isplacement command
//    and sets the degrees of freedom on the associated nodes
//
// Format:
//     D,node_num,UX,disp[,,,,UY,ROTZ]
//
readBoundaryCond(;num,disp,dof,tok)
{
     // get the next token string and convert it to a number
     num = val(token(0,delimit));
     tok = token(0,delimit);
     if(strcmp(tok,"UX") == 0)
          dof = ID_100;
     else if(strcmp(tok,"UY") == 0)
          dof = ID_010;
     else if(strcmp(tok,"ROTZ") == 0)
          dof = ID_001;
     else
          print("Error: No support for dof - ",tok," on node ",num);
     disp = val(token(0,delimit));
     if(disp > 0.0)
          print("Warning: Displacment > 0 on node ",num);
     // read in the rest of the degrees of freedom
     tok = token(0,delimit);
     while(tok){
          if(strcmp(tok,"UX") == 0)
               dof = dof | ID_100;              / bitwise OR to add additional dof
          else if(strcmp(tok,"UY") == 0)
               dof = dof | ID_010;
```

```
            else if(strcmp(tok,"ROTZ") == 0)
                dof = dof | ID_001;
            else
                print("Error: No support for dof - ",tok," on node ",num);
            tok = token(0,delimit);
        }
        // add the node to the data base
        NodeSetDOF(num,dof);
    }
```

ANS-OUT.BOB

```
// ------------------------------------------------------------
//  File:          ANS-OUT.BOB
//  Description:   Exports a Cross Link structure to ANSYS
//  Written By:    KME
//  Date:          January 14, 1992
//
//  Copyright (c) 1992, by Kevin Michael Elbury
// ------------------------------------------------------------
main(;file)
{
     file = fopen("test.dat","w");
     // write a file header, go directly to PREP7
     fwrite(file,"! --------------------------------\n");
     fwrite(file,"!  - Cross Link to ANSYS Transfer -\n");
     fwrite(file,"! --------------------------------\n");
     fwrite(file,"/PREP7\n");
     fwrite(file,"/TITLE Cross Link to ANSYS Transfer\n");
     // dump the data to 'file'
     writeNodes(file);
     writeElements(file);
     writeBoundaryCond(file);
     // close the file
     fclose(file);
     Speaker();
}

// writeNodes()
// %% Extracts all node info from the database and dump to file in ANSYS format
//
// Format:
//     N,node_num,x_coord,y_coord
writeNodes(file;num,x,y)
{
     print("Writing nodes...");
     fwrite(file,"! --------------\n");
     fwrite(file,"! ***Node Data\n");
     fwrite(file,"! --------------\n");
     num = NodeFirst();
     while(num){
          x = NodeGetX(num);
          y = NodeGetY(num);
          fwrite(file,"N,",num,",",x,",",y,"\n");
          num = NodeNext();
     }
}

// writeElements()
// %% Extracts all element info from the database and dump to file in ANSYS format
//
// Format:
//     E,lonode_num,hinode_num
//
writeElements(file;num,lo,hi)
{
     print("Writing elements...");
     fwrite(file,"! --------------\n");
     fwrite(file,"! ***Element Data\n");
     fwrite(file,"! --------------\n");
     num = ElementFirst();
```

```
        while(num){
            lo = ElementGetLoNode(num);
            hi = ElementGetHiNode(num);
            fwrite(file,"E,",lo,",",hi,"\n");
            num = ElementNext();
        }
}

// readBoundaryCond()
// %% Scans node database and extracts any boundary conditions in ANSYS format
//
// Format:
//     D,node_num,UX,disp[,,,,UY,ROTZ]  - [] optional
//
writeBoundaryCond(file;num,dof)
{
        print("Writing BC's...");
        fwrite(file,"! ----------------------\n");
        fwrite(file,"! ***Boundary Conditions \n");
        fwrite(file,"! ----------------------\n");
        num = NodeFirst();
        while(num){
            dof = NodeGetDOF(num);
            if(dof == ID_100)
                fwrite(file,"D,",num,",UX,0.0\n");
            if(dof == ID_110)
                fwrite(file,"D,",num,",UX,0.0,,,,UY\n");
            else if(dof == ID_111)
                fwrite(file,"D,",num,",UX,0.0,,,,UY,ROTZ\n");
            else if(dof == ID_101)
                fwrite(file,"D,",num,",UX,0.0,,,,ROTZ\n");
            else if(dof == ID_001)
                fwrite(file,"D,",num,",ROTZ,0.0\n");
            else if(dof == ID_010)
                fwrite(file,"D,",num,",UY,0.0\n");
            else if(dof == ID_011)
                fwrite(file,"D,",num,",UY,0.0,,,,ROTZ\n");
            num = NodeNext();
        }
}
```