THE INTEGRATION OF COMPUTER AIDED DESIGN AND FINITE ELEMENT ANALYSIS TOOLS USING A LOGIC-BASED APPROACH

JASON DAVID DE BEER

A THESIS SUBMITTED IN COMPLIANCE WITH THE REQUIREMENTS FOR THE MASTER'S DEGREE IN TECHNOLOGY IN THE DEPARTMENT OF MECHANICAL ENGINEERING AT TECHNIKON NATAL

APPROVED FOR FINAL SUBMISSION

DAVID JONSON MSc Eng (Natal) SUPERVISOR	MARK WALKER MSc Eng (Natal), PhD (Natal) CO-SUPERVISOR	21 MAY 2002 DATE

DURBAN, SOUTH AFRICA

JANUARY 2002

DECLARATION

I declare that this dissertation is my own unaided work except where due acknowledgement is made to others. This dissertation has not been submitted previously for any other degree or examination.

Jason David de Beer

January 2002

ACKNOWLEDGEMENTS

I would like to give special thanks to Mr. David Jonson and Prof. Mark Walker who gave me the opportunity to undertake this research project, and who's generosity allowed me to fulfil my objectives. I would like to thank David Jonson again for his continued support, wisdom and guidance that reduced a task that seemed impossible to one with attainable goals. I would also like to thank Nitin Daya for his assistance with the research and the concept development, and my parents for their endless love and support. Lastly, I would like to thank Fiona Scott, whose presence was my inspiration and whose words my motivation.

ABSTRACT

Today's powerful computer-aided engineering (CAE) products have reached ground breaking levels of sophistication when compared with the almost archaic technology used by our predecessors. Engineers are able to develop complex three-dimensional models, or *virtual prototypes*, using powerful 3D modelling capabilities, and from these models, generate manufacturing drawings, motion analysis models, and even finite element models. However, even though this technology brings the engineer one step closer to the realisation of complete *virtual prototyping*, the fulfilment of this goal still requires significant research and development, especially in the areas concerning aspects of product integration. The work presented here attempts to improve product integration, specifically between the design generation and finite element analysis (FEA) code, by introducing the concept of a centralised product data model. By improving the product integration, the amount of human interaction, previously needed to perform laborious, repetitive tasks and tasks requiring high levels of expertise, can be reduced, therefore resulting in a more efficient design process.

A methodology has been developed, using a logic based approach, that efficiently provides the integration between the product data model and the FEA code. The models contained in the product data model are represented by a hierarchical system providing the necessary relational constraints. All models are originally represented using *face sets*, a compact form of geometrical representation. Using the logic based methodology, the information required to create an FE model is automatically obtained from the product data model. By utilising the method of contact surface recognition developed for this application, the appropriate boundary conditions, existing due to the specified physical relations between adjacent models, are automatically determined. Communication is then established with the FEA code using a script to convey the relevant information. Through the interpretation of the script, a FE model is automatically created and analysed and the results obtained are subsequently channelled back to the database to be used at the engineers discretion.

TABLE OF FIGURES

		Page
Figure 2.2.1	Integrated system with program-specific interfaces.	12
Figure 2.2.2	Integrated system using a centralised product data model.	13
Figure 2.2.3	Blackboard architecture.	14
Figure 2.2.4	The MRA for design – analysis integration.	15
Figure 2.3.1	Tree-structure for hierarchical representation.	17
Figure 3.5.1.1	Graphical representation of family tree.	39
Figure 3.5.3.1	Predecessor relationship.	44
Figure 4.3.1	FE model of simply supported beam.	53
Figure 6.1.1	Graphical representation of face set.	65
Figure 6.1.2	L-shaped bracket and face set representation.	67
Figure 6.3.1	Hierarchical tree-structure.	73
Figure 7.3.1	L-shaped bracket defined by four face sets.	80
Figure 7.3.2.1	The normal of two vectors of a face set.	82
Figure 7.3.3.1	Face set contiguity and common planes.	85
Figure 7.3.5.1	Solid and surface representation.	90
Figure 7.4.1	Representation of contact surface areas.	92
Figure 7.4.1.1	Labelled face set.	93
Figure 7.4.1.2	Illustration of the area method.	95
Figure 7.4.2.1	Plate representation by closed boundary curves.	96
Figure 7.4.2.2	Line intersection.	99
Figure 7.4.2.3	Possible intersection types.	99
Figure 7.4.2.4	Intersection between primary and secondary set.	107
Figure 7.4.4.1	Counting intersection method.	115
Figure 7.4.6.1	Curve cleanup.	118
Figure 7.4.9.1	Labelled primary and secondary components.	129
Figure 7.4.9.2	Primary and secondary curve intersections.	131
Figure 7.4.9.3	The final curves for the selection process.	135
Figure 7.4.9.4	Free and contact surfaces.	137
Figure 8.1	The hierarchical tree-structure.	149
Figure 8.2	Components of the piston – crank mechanism.	150
Figure 8.3	The assembled piston – crank mechanism.	151

Figure 8.4	Sub-tree defining the parent and child components.	152
Figure 8.5	The free and contact surfaces on the piston.	153

LIST OF TABLES

Table 3.2.1.	Implication truth table.	32
Table 6.3.1.	*Microsoft Access* database.	71
Table 7.4.2.1.	The eight possible curve intersection types.	100
Table 7.4.2.2.	Intersection method results table.	103

TABLE OF CONTENTS

Page

DECLARATION i

ACKNOWLEDGEMENTS ii

ABSTRACT iii

TABLE OF FIGURES iv

LIST OF TABLES v

TABLE OF CONTENTS vi

CHAPTER 1 1

INTRODUCTION 1

CHAPTER 2 9

VIRTUAL PROTOTYPING 9

2.1 INTRODUCTION 9

2.2 DESIGN - ANALYSIS INTEGRATION 9

2.3 COMPONENT REPRESENTATION 16

2.4 MULTIBODY DESIGN SOFTWARE 18

2.5 MULTIBODY ANALYSIS SOFTWARE 19

2.6 ASPECTS OF IMPLEMENTATION 20

2.6.1 THE USE OF OBJECT-ORIENTATED TECHNIQUES 20

2.6.1.1 CONCEPTS IN OBJECT-ORIENTATED PROGRAMMING 21

2.6.1.2 AN OBJECT-ORIENTATED DESIGN METHODOLOGY 22

2.6.2 THE USE OF ARTIFICIAL INTELLIGENCE 24

2.6.2.1 IMPROVING PRODUCT DESIGN SYSTEMS 25

2.6.2.2 METHODS OF INFORMATION-PROCESSING 27

CHAPTER 3 30

LOGIC BASED PROBLEM SOLVING 30

3.1 INTRODUCTION 30

3.2 THE MATHEMATICS OF LOGIC 31

3.3 DEDUCTION USING PROPOSITIONAL LOGIC 34

3.4 PREDICATE LOGIC 35

3.5 LOGIC PROGRAMMING WITH PROLOG 38

3.5.1 DEFINING RELATIONS BY FACTS 38

3.5.2 DEFINING RELATIONS BY RULES 40

3.5.3 RECURSIVE RULES 42

3.5.4 REPRESENTATION USING LISTS 44

CHAPTER 4 46

THE FINITE ELEMENT METHOD 46

4.1 INTRODUCTION 46

4.2 THE FINITE ELEMENT PROCESS 48

4.3 LOADS AND BOUNDARY CONDITIONS 51

4.4 CONVERGENCE 53

4.5 MESH GENERATION 55

4.5.1 INTRODUCTION 55

4.5.2 OCTREE 57

4.5.3 DELAUNAY 57

4.5.4 ADVANCING FRONT 59

4.5.5 INDIRECT QUAD MESHING METHODS 59

4.5.6 DIRECT QUAD MESHING METHODS 60

CHAPTER 5 62

CONTACT SURFACE DETECTION 62

5.1 INTRODUCTION 62

5.2 EXISTING METHODS 63

CHAPTER 6 65

THE INTEGRATED SYSTEM (PART 1) ____ **65**
6.1 FACE SET REPRESENTATION ____ 65
6.2 THE SCRIPT FORMAT ____ 67
6.3 THE PRODUCT DATA MODEL ____ 70

CHAPTER 7 ____ **75**

THE INTEGRATED SYSTEM (PART 2) ____ **75**
7.1 INTRODUCTION ____ 75
7.2 DATA RETRIEVAL ____ 76
7.3 GEOMETRICAL REPRESENTATION ____ 79
7.3.1 FACE SET DATA CONVERSION ____ 80
7.3.2 FACE SET PLANE DEFINITION ____ 81
7.3.3 CONTIGUOUS FACE SETS ON THE SAME PLANE ____ 85
7.3.4 GROUPING OF BOUNDARY CURVES ____ 88
7.3.5 SURFACE OR SOLID REPRESENTATION ____ 90
7.4 CONTACT SURFACE DEFINITION ____ 91
7.4.1 THE AREA METHOD ____ 93
7.4.2 METHOD OF INTERSECTIONS ____ 96
7.4.3 DEFINING THE SECONDARY GEOMETRY ____ 108
7.4.4 INITIAL GEOMETRY GROUPING ____ 110
7.4.5 BOUNDARY INTERSECTIONS ____ 117
7.4.6 CURVE CLEANUP AND LABELLING ____ 117
7.4.7 DEFINITION OF THE FREE SURFACES ____ 121
7.4.8 DEFINITION OF THE CONTACT SURFACES ____ 125
7.4.9 TWO-DIMENSIONAL PLATE EXAMPLE ____ 128
7.5 THE SCRIPT GENERATION ____ 137
7.5.1 COORDINATE SYSTEM CREATION ____ 138
7.5.2 MESH GENERATION ____ 141
7.5.3 LOAD APPLICATION ____ 142
7.5.4 CONSTRAINT APPLICATION ____ 144
7.6 OUTPUT DATA RETRIEVAL ____ 146

CHAPTER 8 ____ **149**

APPLICATION EXAMPLE 149

CHAPTER 9 155

CONCLUSION 155

REFERENCES 159

APPENDIX 166

CHAPTER 1

INTRODUCTION

Today's technology is increasing at an exponential rate. The sheer level of sophistication inherent in the average personal computer, in terms of hardware performance and software capabilities, totally eclipses the almost primitive technology of yesterday. An article in a 1949 addition of the *Popular Mechanics Magazine*, forecasting the relentless march of science, quoted the following extract, "Computers in the future may weigh no more than one and a half tons" [1]. An optimistic quote, for its time, which falls far short of today's reality. This rapid advancement of technology has proved to be extremely beneficial in many, if not all, areas of science and its associated disciplines, one of which being the field of product design.

In the past, and to a large extent even today, the design of a component, or an assembly of components, has required the manufacture of expensive prototypes. The reason for developing these expensive physical models, is to gauge the performance of the desired component subjected to operating conditions simulating those experienced in reality. However, the availability of sophisticated Computer-Aided Design (CAD) and Computer-Aided Engineering (CAE) applications, is rapidly changing the face of product design by allowing the creation of complex three-dimensional models using the powerful modelling capabilities developed for these applications. These 3D models, appropriately called *virtual prototypes*, not only allow for efficient visualisation but also for the detection of interference between components and the ability to generate manufacturing drawing and models used for evaluation purposes [2]. The use of virtual prototypes eliminates the need for actual prototypes in the early stages of the design process. John Baker, product manager at Unigraphics, quoted the following, "Virtual prototyping originally evolved out of the desire to eliminate costly physical prototypes" [3]. Virtual prototyping basically involves the creation of a digital model, as apposed to a physically based one, for product evaluation and testing. On large projects, the virtual components can be stored in relational databases where the engineers can effectively query any component, contained in that database, at any given time, and be totally assured that they are receiving the current design data and not information that has

become obsolete. Although virtual prototyping will probably never totally eliminate the need of physical prototypes, its increased utilisation in the preliminary stages of the design process will definitely prove to be advantageous.

As mentioned previously, virtual prototyping systems are not merely pragmatic visualisation tools, they need to incorporate the potential to display "intelligent" behaviour, the ability to emulate, to a certain degree, the human reasoning process. Virtual prototyping systems also need the ability to incorporate third party software without having to make laborious changes to the design system, and yet still provide efficient integration between the appropriate applications. There is a need for contemporary CAD systems to provide more facilities for technical links between the components of an assembly and between the various engineering disciplines associated with the design process. "Virtual prototyping today must be centred around the entire product design life cycle, from conceptual design to manufacture, and ultimately product design and development must be 100% digital", quoted from Bernard Charles of Dassult Systems [3]. There is definitely a great demand for virtual prototyping systems, or "intelligent" CAD systems that can effectively replicate the decisions made by an expert designer. These design systems should always take the following considerations into account [4]:

- The human designer should ultimately maintain control of the design process but, the virtual prototyping system should simultaneously take part in the design process as far as possible.
- Design is an activity that attempts to produce components that satisfy the necessary requirements in a realistic environment. However, the requirements are not always fixed and often change during the design process. Therefore the virtual prototyping system should have the ability to adapt to accommodate these changes.
- The design of a mechanical system can be very complex, and therefore the virtual prototyping system must be able to efficiently handle large quantities of information, and at the same time be able to manipulate the data to adapt to the dynamic environment.

- As design is a complex activity involving different types of problems, the virtual prototyping system must be a general purpose system that can support all aspects of design.

Thus flexibility, adaptability, expandability, practicability and generality are the conditions that must be satisfied by virtual prototyping systems. The demands or requirements of these virtual prototyping systems are prodigious, and as a result, years or even decades may pass before all these requirements can be fully realised by a single virtual prototyping system. However, there are many benefits to be realised through the use of virtual prototyping, benefits which can ultimately be achieved through dedicated research. Some of these are:

- Typically, a virtual prototype costs less than a physical one in terms of time and materials. Eliminating even a single physical prototype, in the case of complex assemblies, can significantly reduce the design costs [3].
- A virtual prototype can be modified and re-evaluated in a far shorter time period, not only increasing the overall efficiency, but also allowing for a more "fluid" design process.
- Virtual prototyping has proved to be invaluable for checking initial concepts and performing analysis and testing prior to manufacturing.
- In situations concerning large complex assemblies, where the product design is broken down into a number of individual groups each concerned with the design of a particular aspect of the assembly, the use of virtual prototyping can be beneficial by allowing the individual groups to see each others work in progress [5].
- Design companies around the world can have remote access to the virtual models over the internet.
- Virtual prototypes can be used in computer - based simulations to evaluate the products behaviour in variety of probable environments, environments which would be difficult to replicate when testing physical prototypes. Virtual prototypes can also be tested to failure without having to remanufacture the model after testing, since the model only exists in the realms of the virtual or digital world.
- Virtual prototyping also aids the marketing campaign by allowing the virtual model to be visualised by the target market before the product has been manufactured.

These reasons emphasize the significance of virtual prototyping in its ability to improve the overall efficiency of the design process. However, even though the functionality of today's CAD systems has increased dramatically, resulting in improved product quality and increased productivity, these current systems do not yet provide the necessary functionality required by true virtual prototyping systems. Between 1991 and 1996 major corporations in America [2] reduced the design time requirements to prepare a finite element model by 27%, and the average time to complete a full analysis by 48% (a finite element model is essentially a mathematical representation of a physical component. By applying realistic loads and boundary conditions to this model, the designer can gauge the expected performance of the component). This increased efficiency can be attributed to improved automatic meshing facilities and advancements in CAD - finite element analysis (FEA) integration. Although these figures do represent an appreciable reduction in design time, the overall design process is still far from being efficient. Performing a finite element analysis for design verification takes on average seven days, however designers can make changes to their products at a much faster pace [2]. This fact often results in an unnecessary bottleneck disrupting the natural progression of the design process.

A problem reported by many design companies is that they do not have the levels of expertise required to generate acceptable finite element models and subsequently verify the results. Typically a designer is required to have expertise in both CAD and FEA programs to allow for an even remotely efficient design process. Initially conceptual models are created in a CAD environment, and depending on the particular software used by that company, the model would either be imported, using one of the supported interface specifications (ACIS, Parasolid, IGES, AutoCad, STEP, etc.), into the FEA software or totally recreated in the FEA environment. Even if the model is directly imported from the CAD software, the geometry usually needs to be significantly modified before the model can be meshed and eventually analysed. However, experience suggests that even if a solid model of the component exists, it is seldom used directly to create the analysis model but rather the necessary geometry is reconstructed. The most obvious reason for this geometry duplication is the difficulties associated with transforming the solid model into the abstracted and discretised form required to perform a FEA [6]. This need for constant human intervention to perform laborious, and often repetitive tasks naturally hinders the flow of the design process.

Other problems often encountered when considering large complex assemblies, consisting of thousands of individual components, are the requirements on the computer's resources in terms of both hardware and software. The data models used to represent these complex assemblies are invariably inefficient. There is great pressure on the software developers to determine a more efficient means of representing complex assemblies, and on the hardware manufactures to continuously improve the current levels of component technology. The realisation of these goals is an essential prerequisite in order to accommodate the ever increasing demands of virtual prototyping.

It becomes immediately apparent from the above information that there are two main areas effecting the development of virtual prototyping systems. The first concerns the integration between the design generation and evaluation software, and the second involves the method of information representation and storage, especially in complex assemblies. It was initially decided to focus on the first aspect, concerning the software integration, and develop an integrated conceptual design system. The broad field of product design and development encompasses a large collection of disciplines, including quality assurance, manufacturing planning, motion analysis and finite element analysis to name a few. Due to interests in the field of finite element analysis, and fortunately the availability of *MSC Nastran for Windows (MSCN4W)*, an FEA application, it was subsequently decided that the focus of this research project would be orientated specifically towards the integration of design generation and finite element analysis codes. Although this integration would represent only a small portion of an entire product design mechanism, the underlying principles and ultimately the final concepts developed could effectively be adapted to any one of the applicable fields associated with product design.

After conducting a preliminary literature survey concerning design - analysis integration, it was evident that there were basically two predominant methods of software integration currently being researched and developed. The first involves the sharing of information directly between applications. Examples of this type of integration are being developed commercially and are currently featured in *4D Nastran,* where a motion analysis and finite element analysis code are integrated, and the *Pro / Engineer* and *Pro / Mechanica* integrated software system, where design generation and

finite element analysis components work together. The interfaces are designed specifically in terms of the required information to be exchanged and for this reason this method does not lend itself well to changes in the components used. The second approach consists of using a unified representation of the design and analysis information which essentially constitutes the product data model, similar to the idea proposed by Wu et al. [7]. The product data model incorporates all the information required to describe the model both dimensionally and functionally and communicates this information to whichever application requires it. Should any one of the software components be replaced, only one communication route would require modification. This approach is flexible in that it makes it relatively straightforward to add additional functionality, for example machining or cost analyses. For this particular reason, the second approach was selected.

After recognising the need or importance of improving the integration between the design generation and analysis phases of the design process, and subsequently selecting a suitable method of integration, the main goal of this research became apparent. An efficient mechanism needed to be developed that would provide seamless integration, requiring minimal user interaction, between the respective codes. This mechanism would be required to automatically obtain the relevant geometrical and functional information from the product data model, convert this information into a form that could be directly interpreted by the FEA code, subsequently channel this information to the FEA code and finally obtain the results, possibly stresses and displacements, from the finite element analysis. The designer would then have direct access to these results and could then determine the appropriate course of action.

It was initially evident that knowledge would have to be acquired in a number of engineering and associated disciplines in order to allow for the development of the integrated mechanism. These initial disciplines or areas of research were subsequently divided into the following sections:

- The integration of the design generation and design evaluation phases of the product design cycle.
- The application and use of databases for the development of the product data model.

- Different methods of geometry representation as an efficient and compact means of representing the component's geometry.
- The selection of a suitable programming language that could be used effectively to control and process the necessary information.
- The implementation of the finite element method, not only the fundamental theory, but also the ability of MSCN4W to be controlled externally and automatically through the use of its inbuilt programming language.

It was also required to conduct research into the field of object recognition. More specifically, a method needed to be developed that would allow the contact surfaces, between adjacent components in a mechanical assembly, to be identified and defined. The reason for this contact surface recognition capability will be explained in the relevant chapter, but basically involves the discretisation of the component's surface geometry into areas which have no contact with adjacent components, and into areas which are in contact and consequently require the application of the appropriate boundary conditions. Due to the amount of research and development involved, this section became one of the major focuses of this research project.

The outline of the work presented for this research project can be summarised as follows. Chapters two to five are allocated to the research of the relevant literature. Chapter two contains a summary of the information reviewed concerning the field of virtual prototyping and includes some of the methods of integration commonly used. This chapter is followed by a relatively detailed introduction into logic and logic programming languages. Subsequently, some of the relevant fundamental information concerning the finite element method is discussed in chapter four. Chapter five closes with a brief discussion concerning some of the common methods of object recognition.

Chapter six and seven are devoted to the implementation of the integrated system. Chapter six initially focuses on the method of geometry representation which is followed by a section describing the product data model as a means of information storage. The chapter closes with a description of the functionality of the script. Chapter seven contains the methodology of the mechanism used to perform the integration between the product data model and the FEA code. The chapter commences with an introduction into the method of data retrieval from the product data model. The next

section describes the method of geometry conversion from a face set to a boundary curve representation. Subsequently the contact surface recognition method is discussed. The penultimate section describes the generation of the script used to input the relevant information into the FEA code, and finally the chapter concludes with an explanation into the retrieval of output data resulting from the analysis of the finite element model.

Chapter eight contains an application example of a piston - crank assembly and ultimately chapter nine concludes the dissertation. The appendix includes a copy of the code used to perform the integration. Each of the rules in the code are conveniently numbered to allow for its structure to be described in chapter seven.

CHAPTER 2

VIRTUAL PROTOTYPING

2.1 INTRODUCTION

The title of this dissertation automatically suggests a strong correlation towards the field of virtual prototyping or rapid product development, and therefore this field of study forms the basis of this dissertation. However, virtual prototyping is itself a relatively broad field encompassing a plethora of related disciplines, and therefore certain key aspects, aspects that were identified as being of primary importance with regard to the selected topic, were subsequently highlighted and separated from the remaining aspects of lesser significance. The selected fields will be discussed in the sections that follow. The discussion commences with a description of some of the research that has been conducted in the field of virtual prototyping, primarily research concerning the integration of the design and analysis phases of typical virtual prototyping systems. Included in this chapter is also a description of the use of Artificial Intelligence (AI) in the design field. The field of virtual prototyping was introduced in the previous chapter, and therefore to avoid unnecessary repetition, an overview of the field will not be repeated. The key aspects of this chapter are concerned with methods of integration between the design generation and analysis phases of the product design process. The integration between the respective phases of the design process is usually achieved by utilising a certain model of data representation and by defining a means of data communication within this model. Various models of data representation have been developed over the past decade and will be described in the following sections.

2.2 DESIGN - ANALYSIS INTEGRATION

The predominant reason for integrating the design and analysis phases of the product design process is one of increased productivity. This increased productivity is usually realised through the reduction of time consuming and repetitive work typically

performed by the analyst. Initially analysis tools were used to determine why a product had failed and to identify what modifications were required to prevent further failure. However the recent advancements in product integration have resulted in more efficient design systems that are capable of analysing the components long before they go into service and the ability to respond rapidly to changes in the design process, resulting in shorter product development times, and hence increased productivity. In order for a virtual prototyping system to achieve the objective of increased productivity, the following requirements need to be satisfied [4, 8]:

(1) A robust data representation facility containing geometric and non-geometric data such as environmental conditions, design requirements, manufacturing methods etc.
(2) An intelligent, semi-automated means of transforming the geometric and non-geometric information into a suitable analysis model.
(3) Support for numerous diverse analysis models for each product type. The same kind of product often has analysis models from a variety of engineering disciplines that involve different solution techniques.

These requirements have been satisfied, at least partially, by the proposed development of a number of integrated design - analysis systems. Several approaches to the integration have been developed over the past years and can be roughly divided into the following categories [9]:

(1) *Direct translation approach* – Each application program's data is translated from its format to the specific format required by another program. The process usually involves preprocessing and postprocessing of multiple files to generate single or multiple files that satisfy the required formats. Examples of this type of integration are being developed commercially and are currently featured in *4D Nastran* and *visualNastran Desktop FEA*. These applications incorporate the integration of motion analysis and finite element analysis code. Another example is the *Pro / Engineer* and *Pro / Mechanica* integrated software system, where design generation and finite element analysis components are combined into a single system. Other commercial systems, such as *Virtual Product Model* and *Virtual Product Development*, support the integration of digital prototyping and data management methodologies [3]. A final example of this type of integration is the direct interface

between *DADS*, a motion analysis application, and *Pro / Engineer*. The interface allows for the automatic extraction of solid model body data (mass, inertia, second moments of area etc.) for use in the *DADS* analysis [5].

(2) *Neutral file approach* – All application programs read from and write to a file that is in a standard format. In other words, information in one data set is essentially translated into a common format required by other programs. This approach, based on *IGES* (An American National Standards Institute standard that gained acceptance in the early 1990's), was adopted by Abdalla and Yoon [9] based entirely on practical considerations.

(3) *Central database approach* – Data common to all applications is kept in a central facility or database where information can be directly accessed from the individual applications, using a database management system (*DBMS*). This approach has been utilised directly in the integrated systems proposed by Hardell [10], Arabshahi et al. [6], Remondini et al. [11], Gabbert and Wehner [12], and Mackie [13] and indirectly, or with slight modification, as proposed by Holzhauer and Grosse [14], Peak et al. [8] and Wu et al. [7]. Although the work presented by Abdalla and Yoon [9] focuses on a neutral file approach, the author incorporates the concept of a centralised model containing general analysis information, and therefore this method is very similar to the central database approach.

(4) *Distributed database approach* – Information is stored in local application databases. Information common to two or more applications is shared through a knowledge-based system.

The selection of a suitable method of integration was facilitated by a process of elimination. Firstly, the direct translation approach was disregarded primarily due to its method of data storage. The major disadvantage of this method of storage is that the data is stored in the specific database of the application. Therefore the use of the internal database makes the system application specific, or in other words, only a specific analysis application can be used with the design application. If for any reason the analysis application is required to be replaced, the entire system would probably require significant modification. For similar reasons, the distributed database approach was overlooked. This method of integration is unnecessarily segregated when compared to the central database approach. Eventually, the neutral file approach was eliminated principally due to the benefits that would be realised through the implementation of the

central database approach. For example, in Figure 2.2.1, a program-specific integrated system is shown.

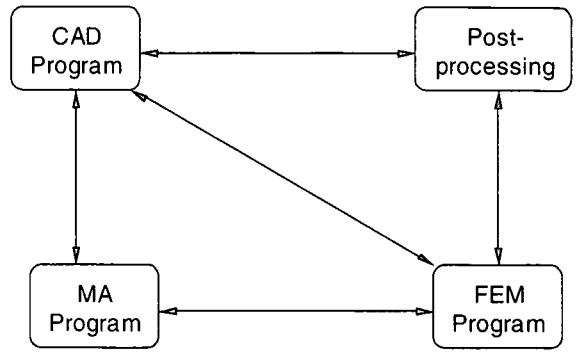


Figure 2.2.1. Integrated system with program-specific interfaces.

Each arrow corresponds to an interface that needs to be created. In this integrated environment, a change to the CAD program will require three new interfaces to be developed. Alternatively, if the central database approach is utilised, the integrated system immediately becomes more expandable and adaptable, important features that should always be incorporated into an integrated system. Using the central database approach, as shown in Figure 2.2.2, a change to the CAD program only requires a single new interface to be developed [10].

In general the integrated system must posses the ability to incorporate a variety of different engineering disciplines, disciplines that will require the application of a wide range software products predominantly developed by different suppliers. It is therefore important that the integration is achieved, where possible, by the use of standard protocols. This essentially allows one program to be replaced by another with the same functionality without affecting the complete design environment. As mentioned previously, the approach depicted in Figure 2.2.2 was adopted by Hardell [10], Arabshahi et al. [6], Remondini et al. [11], Gabbert and Wehner [12] and Mackie [13] to facilitate the development of their integrated systems. Holzhauer and Grosse [14] and Peal et al. [8] have proposed a number of useful conceptual modifications to the central database approach.

In the work presented by Holzhauer and Grosse [14], an approach using a *blackboard architecture* has been developed to facilitate the integration of diverse computational

systems into a single integrated system. The blackboard is essentially a database broken down into four spaces, each storing information corresponding to different phases of the problem solution.

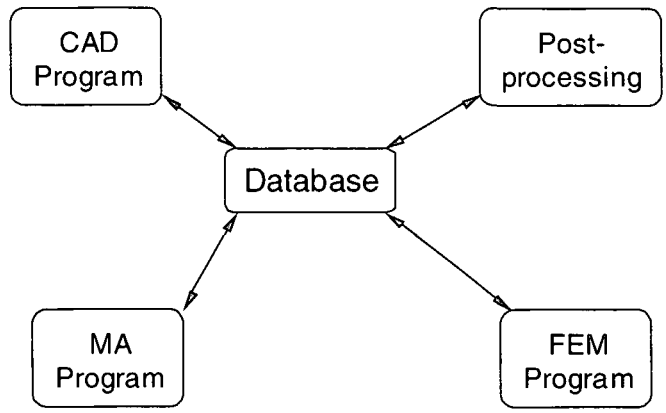


Figure 2.2.2. Integrated system using a centralised product data model.

A separate knowledge source (the concept of a knowledge source or knowledge base will be discussed in section 2.6.2) interacts directly with each of the partitions in the blackboard database performing particular functions that lead to the problem's solution. Figure 2.2.3 shows a simplified representation of an integrated system using a *blackboard architecture*. The partitioning of the problem into sections addressed by specific Knowledge Sources (KS) is analogous to the solution of a problem by several experts. Using this analogy, an expert (knowledge source) has access to the information, describing the problem, contained in the blackboard database. However, the experts are required to patiently wait in line until their expertise can be applied by taking information off the blackboard and writing new information onto the blackboard. In other words, there is no direct transfer of information between the individual knowledge sources, and the problem is solved in a sequential manner by multiple experts.

The method of queuing subsequent analyses, utilised in the blackboard architecture approach described above, represents a useful concept when dealing with product design systems containing many analysis modules. A good example would be a product design system that provides the integration of Motion Analysis (MA) and Finite Element Analysis (FEA) code with design generation software. For the purpose of this example, assume that a designer is interested in determining the stresses induced in one of the components of a piston - crank mechanism during normal operating conditions.

The designer therefore attempts to perform an FEA on that particular component. However, the FEA cannot be performed until the MA program has calculated the various forces resulting from the simulated operating conditions. As a result, the FEA must be placed in a queue until such a time as the results from the MA are available. This approach facilitates a natural progression through the various engineering disciplines of a product design system.

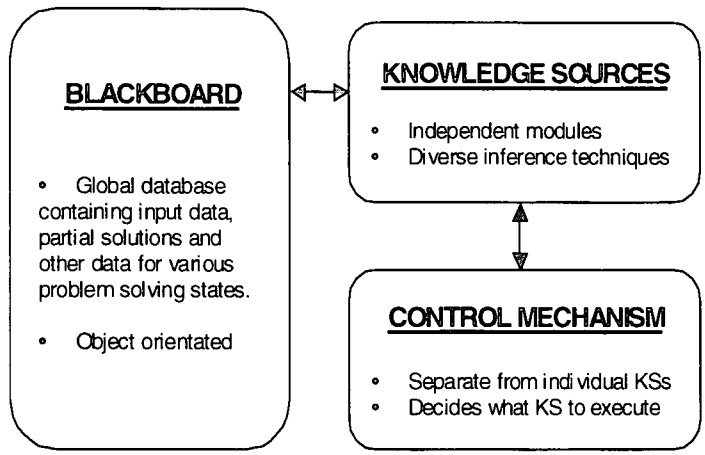


Figure 2.2.3. Blackboard architecture.

In the work presented by Peak et al. [8], a Multi-Representation Architecture (MRA) has been developed to address the problem of design - analysis integration. The MRA uses four information representations as stepping stones between the design and analysis extremities. On the one extremity is the P*roduct Model* (PM), which is considered the master description of a product, supplying information to other product life cycle tasks such as engineering analysis and manufacturing. To allow for the implementation of several analysis applications, PMs support idealisations that relate detailed, design-orientated attributes with simplified, analysis-orientated attributes. On the other extremity are the S*olution Method Models* (SMMs) representing analysis models in a relatively low-level, solution method-specific form. SMMs also combine solution tool inputs, outputs and control into a single information entity to facilitate automated solution tool access and results retrieval. SMMs are generated through transformations by A*nalysis Building Blocks* (ABBs), with the objective of obtaining results that are based on solution method considerations. ABBs represent engineering analysis concepts in a manner that is almost entirely independent of product application and solution method. The final model type involved in the integration is the P*roduct*

Model-Based Analysis Model (PBAM), which contains links that represent design - analysis associativity between PMs and ABBs. The associativity links represent the usage of idealisation for a particular analysis application. The MRA architecture can be seen in Figure 2.2.4.

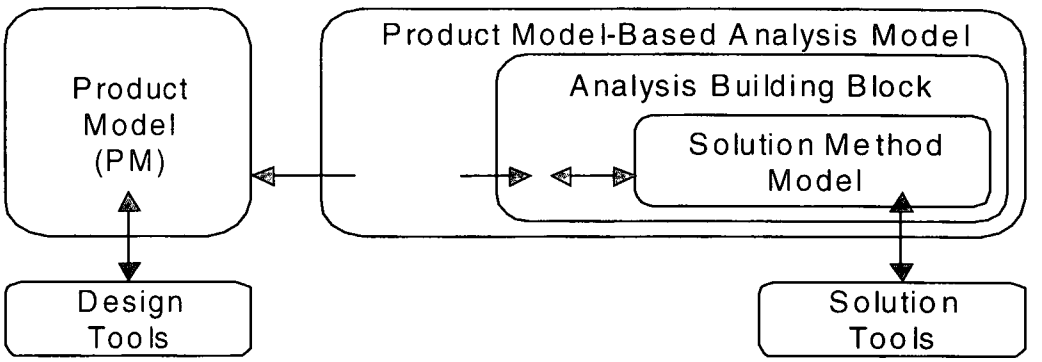


Figure 2.2.4. The MRA for design - analysis integration.

The MRA is an effective strategy in that it breaks the design - analysis integration gap into smaller sub-problems. The MRA is based on modular, reusable information building blocks and therefore provides flexibility and has the ability to support different design and analysis tools. These three paradigms of integration that have just been discussed have one feature in common, the central database or product data model. This feature allows general information concerned with various different analysis applications to be conveniently stored in a manner accessible to all other applications. For the purpose of this research project the concept of a centralised product data model has been adopted to represent the necessary design and analysis information. Although the work presented here is only concerned with the integration of design and FEA code, the very structure of the product data model allows for uncomplicated expandability and adaptability, and therefore further analysis models could be easily incorporated into the design system when required.

Product data models are essentially semantic models of a product representing the product geometry as well as functional, technological and other features associated with the product. In other words, product data models contain both aspects of the product and aspects of implementation [12]. From this fundamental data, other models can be automatically derived. For example, a finite element model can be created from the

information contained in the product data model. In terms of implementation, it was initially decided to use a neutral database (an application independent database), in this case *Microsoft Access*, to represent the product information. *Access* features a sophisticated query language, which allows non-expert programmers to query and update the database.

2.3 COMPONENT REPRESENTATION

A mechanical system is generally composed of an assembly of interconnected bodies or components, and thus lends itself well to representation by hierarchical methods. This representation, viewed as a hierarchy of components, can be conveniently represented using a tree structure. A tree structure usually consists of a set of nodes connected by directed paths [15]. In this hierarchy a node symbolises a component and its child nodes represent the subcomponents. The path connecting the nodes represents some physical relationship between the adjacent components in the mechanical assembly. In general, tree nodes have several incoming and outgoing paths. The node with no incoming paths is called the *root node*, and the nodes with no outgoing paths are called the *leaf nodes*. A node with one or more outgoing paths is called a *parent node*, and the nodes that it connects to are called the *child nodes*.

A simple tree structure is illustrated in Figure 2.2.1, where 'R' represents the *root node*, 'L' represents the *leaf nodes*, 'A' is a *parent node* and 'B' and 'C' are *child nodes*. A segment of a tree consisting of one or more nodes, and the interconnecting paths, is called a sub-tree, for example in Figure 2.3.1, 'A', 'B' and 'C' collectively form a sub-tree. The *degree* of a node is the number of outgoing paths from that node. The *degree* of a tree, or sub-tree, is the maximum number of outgoing paths from any node in that tree. A root node of a tree is said to be at *level* 0. The child nodes of the root node are at *level* 1, and in general, a node is at *level* N if its *parent node* is at level N-1.

Wu et al. [7] proposes a method that adopts this type of hierarchical system. The author suggests that the product data model, of a concurrent engineering environment, contains mechanical systems consisting of interconnected bodies and connectors. A body or a connector is modelled as an assembly composed of parts that are quasi-rigidly fixed

together. Each part is in turn defined by various features describing that part. These mechanical systems, bodies, connectors, assemblies, parts, and features form an object hierarchy consisting of five object levels: environment, mechanical system, assembly, part and feature.

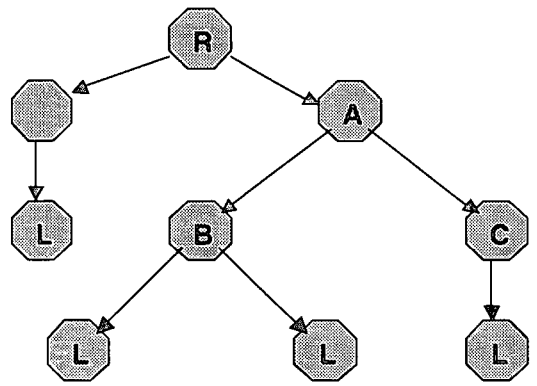


Figure 2.3.1. Tree structure using hierarchical representation.

The author claims that physical (constraint) relationships that are applied to objects naturally fit into these object levels. For example, at an environment level are the constraints that are applied to a mechanical system due to operation and manufacturing conditions, and at the mechanical system level are the constraints applied to the bodies such as degrees of freedom due to mating relations.

Abdalla and Yoon [9] and Shah and Tadepalli [16] use a similar method of hierarchical representation to define the classes of a specific problem. Initially the problem is abstracted into both physical and conceptual classes. Physical classes are those that represent element, property, connection, load etc. These physical classes contain subclasses that represent specific entities of a structural system. For example, the element class would contain subclasses defining the types of specialised elements that could be used for a particular problem solution. The conceptual class contains the conceptual abstractions of the application at hand.

For the purpose of the work presented here, it was decided that the hierarchical approach of representing components would be implemented due to its natural ability to represent mechanical systems. However, as opposed to the more structured methodology developed by Wu et al. [7], the researcher decided to use a more generic

approach, were the components of an assembly are represented on levels which are not explicitly defined. To a certain degree, this approach resembles the system assembly level proposed by [7]. However, instead of having further sublevels defining the part and the features of that part, all geometrical and functional information, concerning a particular component, is encapsulated into the representation of that component on that specific level. One of the predominant reasons for using this approach was its ability to adapt to the implementation of *Microsoft Access*, the database system used in this integrated product design system.

2.4 MULTIBODY DESIGN SOFTWARE

Today there are a wide range of computer programs available for computer aided design, offering various levels of functionality. On the one hand are the applications which are merely convenient replacements for the drafting board, while at the other end of the market are the sophisticated programs offering a variety of engineering activities such as solid modelling, finite element modelling and analysis and manufacturing to name a few. However, when modelling multibody systems (mechanical systems consisting of an assembly of interconnected components), suitable software for computer aided design should satisfy the following requirements [10]:

(1) *Solid modelling capabilities* - it should be possible to define complete and accurate properties of the components of the multibody system. For example, rigid body data for dynamic analysis and surface data for visualisation.
(2) *Mechanism design capabilities* – the system should be able to define attributes such as joints, dampers, springs, applied forces and moments to the components of the assembly.
(3) *Methods of extracting data describing the multibody systems* – the data describing the multibody system is usually dumped from the CAD program to a suitable temporary database or an ASCII file which can be efficiently accessed.

In terms of the design generation software tool used in the development of the integrated system described here, a commercial product was not used. Rather a design generation tool was developed at Technikon Natal based on the *Open Inventor* set of

graphics libraries. This approach was chosen due to the additional flexibility afforded by the low level use of the libraries. The tool is equipped with full featured solid modelling capabilities allowing realistic visual representation of the mechanical system as well as the calculation of all quantities required for the execution of motion analysis. The tool is provided with techniques for the specification of the mechanism specific features mentioned above [17].

2.5 MULTIBODY ANALYSIS SOFTWARE

The calculation of the internal forces (stresses) and the corresponding displacements, associated with the interconnected components that constitute a mechanical system, forms the basic purpose of the integrated system described here. Using numerical methods like finite element analysis, the complete static and dynamic behaviour of the individual components subject to known loads and boundary conditions may be accurately determined. Bearing in mind that one of the requirements of a virtual prototyping system is expandability, the integrated system described here was specifically designed to allow for the inclusion of other analysis applications. For example, in the case of complete systems, the inertial effects involving the motion of the entire mechanism may be more important [18]. In situations like this, the method utilised in determining the behaviour of the entire mechanical system involves initially analysing the mechanism as an ordered structure of interconnected rigid body components using analysis software equipped with *motion analysis capabilities* (typical examples of software displaying such capabilities are the commercially available analysis codes *Adams* and *Working Model 4D*). However, for the purpose of the work presented here, the focus of the research was on the integration of design generation and FEA code. Suitable software for the FEA of the individual components of a mechanical system should include the following functionalities [6]:

(1) *Intelligent meshing routines* – meshing routines with varying degrees of automation to suit the application. Other important considerations are the ability to suggest alternative meshing strategies, give instantaneous feedback on mesh quality and support adaptive mesh refinement.

(2) *Finite element solvers* – a variety of finite element solvers to suit the range of analysis problems. The software should have the ability to give error estimates of the errors incurred in the analysis and be capable of adaptive refinement.
(3) *Post-processing* – the software should have the ability to post-process the results.

In terms of the integrated system considered here, the commercially available analysis code *MSC Nastran for Windows* (*MSCN4W*) has been used. MSCN4W surpasses the requirements mentioned above as it is also equipped with a scripting feature which allows the input of mechanism data via a script which is basically a small program. This allows information to be supplied to the analysis code without any user interaction. The script is essentially the data channel through which analysis specific product data is supplied from the product data model to the analysis code.

2.6 ASPECTS OF IMPLEMENTATION

This section is concerned with the methodologies responsible for the communication of design specific data between the product data model and the analysis applications. This communication or transfer of information is one of the key aspects behind the software integration. The integration is essentially facilitated through the use of a suitable programming language that has the ability to efficiently communicate the required data with other software applications using the standard protocols.

Generally speaking, the two predominant methodologies used to provide the software integration involve using either *object-orientated* or *artificial intelligence* (AI) techniques, which will be duly discussed in the following sections. Holzhauer and Grosse [14] proposes a methodology using a combination of AI and *object-orientated* techniques to control the numerical solution process for a transient thermal analysis.

2.6.1 THE USE OF OBJECT-ORIENTATED TECHNIQUES

Object-orientated approaches have gained popularity in the field of virtual prototyping by providing a structured means of product integration and introducing several

abstraction techniques that close the semantic gap between real-world entities and computer representations of these entities. Various authors have presented work on design - analysis integration using an object-orientated approach to effectively represent and manage the diversity of the information involved. Vallis and Colton [19] used an object-orientated technique to organise information in engineering design applications, or more specifically to organise the information involved in "*under hood packaging*" (a process involving the spatial arrangement of components in an engineering system in order to meet the design constraints). Remondini et al. [11], Gabbert and Wehner [12] and Abdalla and Yoon [9] used an object-orientated approach to create and manage the representation of a designed component, or an assembly of components, in a mechanism. Mackie [13] highlights the advantages of using object-orientated technology to handle complexity in finite element programming. The most notable advantages are:

- Simpler program maintenance.
- Improved implementation of complex algorithms.
- Better integration of analysis and design.

In an attempt to meet the demands of today's virtual prototyping systems, the level of sophistication inherent in these integrated systems is constantly increasing, resulting in the development of programs that exhibit increased complexity. One of the most serious disadvantages of traditional procedural programming languages is that even relatively simple changes to the program structure can have pronounced ripple effects throughout the code. The use of object-orientated techniques eliminates this risk by encapsulating the data which can then only be accessed via specific methods or functions. It should also be noted that declarative programming languages offer a number of advantages over procedural languages with regards to this expandability. However these advantages will be discussed in section 2.6.2, which deals with AI.

2.6.1.1 CONCEPTS IN OBJECT-ORIENTATED PROGRAMMING

In an object-orientated environment, a program is defined as a system of objects, involving both data and operations, where the dynamics of the real world are modelled

as objects that interact with each other via messages. One of the most important features of an object-orientated design, is the design of the classes of objects and the methods of interaction between these objects. All entities (physical or conceptual) are essentially represented by objects, where each object is an instance of a class. A class describes a set of physical or conceptual objects having similar properties, constraints and operations. Each class is responsible for defining attributes that represent the state of the object, and methods that describe the objects behaviour. Each object has specific values that are assigned to the attributes defined for its class. All objects that belong to a specific class have the same methods and therefore behave similarly. The data attributes of a specific object are encapsulated within that object, and as a result only the method is shown. Generally speaking, attributes and methods can be public, protected or private. Public methods or attributes can be accessed by all objects. Protected methods or attributes can be accessed by the object of the class that defined them, and the objects of the derived classes. A private method or attribute can only be utilised by the object of the class that defined it, or by an object that it has given certain access [9].

An important concept in object-orientated programming is inheritance, which allows for the efficient reusability of code. A class may have several derived classes which inherit some of its attributes and methods. Objects communicate or interact with one another through messages by invoking the methods supported by the class of objects. An object (sender) communicates with another object (receiver) by sending a message, to which the receiver responds.

2.6.1.2 AN OBJECT-ORIENTATED DESIGN METHODOLOGY

Object-orientated design methodologies have been extensively researched over the past decade. These methodologies generally strive to incorporate criteria such as clarity, reusability, maintainability and efficiency with an objective of developing a system which is user friendly. Abdalla and Yoon [9] proposes a methodology that is applicable to diverse software design and implementation. The methodology consists of the following four stages:

(1) *Identification of classes and class hierarchies* – the problem is initially abstracted and categorised into physical and conceptual classes, where objects in the classes represent the real-world entities and the conceptual definition of the problem in hand. The classes represent the information required to construct a specific analysis model, for example element, property, load and various other classes. From these base classes, classes containing more specialised information are derived simultaneously inheriting the methods and attributes from the base class.

(2) *Identification of communication channels and protocols* – objects of physical and conceptual classes have many inter-relationships, resulting in the need for data communication and exchange. Therefore the system is required to identify the related objects and the information that is to be communicated. Communication is established by creating data channels between the related objects. The protocol is that an object always sends a message to another object in order to get the necessary information.

(3) *Method and attribute declaration for classes* – the function of each class is described by the declaration of the methods and attributes of the class. This information along with the class name is all the information required to either instantiate an object of that class, or to derive a new class having that class as its base class.

(4) *Use of aggregation for building more complex objects* – many engineering entities (mechanical assemblies) are complex and likely to be composed of several simple objects. Aggregation is a process of building complex objects from simpler ones. Consequently, this is analogous to the bottom-up approach in software design.

The implementation of such an object-orientated methodology allows for the development of reasonably complex, diverse systems. The encapsulation of objects provides flexible modularity in the program and the inheritance mechanism allows for the efficient reusability of specific sections of the program code. These two features of object-orientated programming provide an effective means of software integration and development. However, one of the downfalls of this approach is the run-time inefficiency when compared to traditional programming languages.

2.6.2 THE USE OF ARTIFICIAL INTELLIGENCE

AI can be described from a number of different perspectives [20]. From an intelligence point of view, AI is an attempt at making machines "intelligent", acting as we would expect humans to behave, or the inability to distinguish between human and computer responses. From a research perspective, AI can be likened to the study of how to make computers perform tasks that at the moment humans do better. The study of AI began in the early 1960's [20], initially as an attempt to improve the game playing ability of checkers. Other focuses of AI during its initial conception were theorem proving and general problem solving of simple tasks. Today the study of AI covers a wide range of subjects including robotics, game playing and expert systems. The subject area of expert (knowledge-based) systems, an area which forms an important aspect of the design automation field, can be broadly divided into two main sub areas, representation and search [21]. An expert system can be roughly described as ".... a computer program that behaves like a human expert in some useful ways." [22]. The two major components of an expert system are the knowledge base and the inference engine. The knowledge base contains a library of expert knowledge in the form of facts and rules. The inference engine attempts to find a solution or goal by searching through information stored in the knowledge base. The most flexible method for developing expert systems is through the use of declarative programming languages such as *Prolog* or *Lisp*.

Typically, in design an engineer is presented with a solution search space that is bounded only by the level of experience of the designer. Usually these boundaries are not very clearly defined. However, in an expert system, to maintain an acceptable level of performance and due to technological constraints, the field of information is limited and the boundaries are required to be well defined. These facts suggest that the type of design best suited to the use of AI is one where the solution principle can be determined or calculated before hand, allowing for an efficient search space to be developed.

Logic can be used to express statements, to infer additional statements and to rigorously prove (or disprove) statements [23]. If the concepts of reasoning and intelligent behaviour could be formalized using mathematical logic, it would appear that AI implementation would be relatively straight forward. Unfortunately, this is not the case.

One reason for this is that humans do not always reason by making logical inferences, and logic is too rigorous and to inflexible to be used in all AI domains. Another possible reason is that the typical complex conditions underlying real world design cannot always be taken unabatedly into consideration by these AI systems. However, these reasons are not sufficient to dismiss logic for all AI system development. Mathematical logic provides a stern mechanism for making inferences, which are, using the rules of logic, mathematically sound. Furthermore, logic forms the basis of rule-based systems, and extensions to classical logic provide many useful tools for the design of practical AI systems. One of the main reasons for studying mathematical forms of logic is to be able to develop mechanisms that posses the ability to both represent and manipulate entities known as statements, facts or possibly even knowledge. The semantics of logic and its variations will be discussed in chapter three.

Although there are many obstacles, in terms of implementation, to be surpassed, AI successfully manages to raise the expectations for advancing virtual prototyping technology. Inference mechanisms, knowledge bases and search engines realised in AI technology have made computers more intelligent in their problem solving abilities. Many people have successfully implemented AI into design systems and developed expert system shells used for solving engineering problems. AI definitely possesses the potential to create advanced product design systems [4]. For example Holzhauer and Grosse [14] used AI techniques to intelligently control finite element thermal analyses, resulting not only in gains in the analysis efficiency, but also the ability to optimise the analysis for specific characteristics of each separate analysis.

2.6.2.1 IMPROVING PRODUCT DESIGN SYSTEMS

Effective product design systems must exhibit conditions of flexibility, adaptability and expandability, as well as the capability to represent any design process that may be required. In order to meet these conditions, general purpose information-processing systems need to be developed. These AI systems need to provide capabilities such as inference, knowledge-base management and search mechanisms. The problem solving methodologies used by the AI systems should attempt to mimic those of expert designers. The AI systems must be able to efficiently represent and manipulate real-

world "objects", appropriately termed *object models*. The representation of object models includes information concerning geometry and also any functional information that may be related to the design process. Generally speaking, an object can be associated with an indefinite set of functionalities, however, a model is defined to include only the functionalities that may impact on the immediate problem [21]. As computers can only deal with functionalities that can be explicitly defined, a model description language plays a significant role in AI systems. The model-based method requires a powerful system management capability.

The search for the given knowledge, contained in the knowledge base, to solve the given problem occupies a large percentage of the solution time [4, 20]. Therefore it is quite appropriate to restrict the scope of the search only to areas which are directly concerned with the problem at hand. This can be achieved effectively by dividing the available knowledge into a collection of subclasses, where each subclass represents a certain aspect or domain of the design process. It should be noted that there is no standard design process, as this inevitably depends on the individual designer. However, it is important that the product design systems have the capability to represent any design process that the human designer has in mind. Ohsuga [4] has subsequently labelled the design process represented in computers as the Design-Process model (DP model). Although there is no standard design process, there is a typical DP model that can be applied to a number of problem domains. The design process can essentially be defined as creating an object model and gradually refining this model into its final form. The entire process is conveniently subdivided into a number of design stages, usually described as conceptual, preliminary and detail design. The conceptual design stage begins with a given set of requirements and the designer defines the initial object model. At this stage there is usually insufficient information to make a detailed analysis, but a qualitative estimation can be performed nevertheless. The object model is analysed and evaluated with regards to the given requirements. If for any reason the object model does not meet the requirements, the designer is then coerced into making the necessary modifications, otherwise the object model is determined to have reached a level where it can be refined by the addition of further information. Similar processes are repeated through the preliminary design stage until finally, the fine detail is added and the modelling process is complete. During these design stages the object model gains knowledge and complexity through the acceptance of external information.

There are a number of possible ways of enhancing product design systems using AI, of these, the following two are expected to be the most effective [4]:

- To computerise as many of the individual operations involved in the design process as possible.
- To integrate these different operations into an efficient process.

To achieve the first objective, computers need the capability to perform as many different operations as possible. For example, in analysing certain aspects of the design process, computational methods may have been established and the programs or applications utilising these methods may be available commercially. However, in other situations, the required computational methods may not yet have been developed, and as a result the AI system would then attempt to consult the knowledge base for possible solutions. In certain other situations, the knowledge may not be readily available and the AI system would then rely on the designer's experience and knowledge to determine the solution. To achieve the second goal, AI systems must be able to automatically transfer information between the relevant operations. Knowing that each operation, possibly involving separate computer applications, may be defined independently, the AI system would need to maintain information on what operations should follow the current operation and consequently how to communicate information between the corresponding operations.

2.6.2.2 METHODS OF INFORMATION-PROCESSING

From the viewpoint of product design, there are basically two different methods of information processing, namely conventional information-processing and knowledge-processing. On the basis of information-processing, there is essentially no difference between the two, as both are software systems realised on conventional computers. However, there is a difference in the application level owing to the different programming languages used to represent them. Typically, programming languages are divided into two predominant types, procedural and declarative, where the difference lies in the semantics (Today, most commercially available programming languages, in both the declarative and procedural domains, usually provide an extension that allows

for the implementation of object-orientated techniques. An example of each would be C++ and Visual Prolog). A procedural language can only describe the way a computer operates. There is no direct correspondence between the description in this language and the description of the real world problem, as the correspondence is made only in the programmer's brain. The characteristics of a procedural language are:

(1) A program must be written for each specific problem and is therefore difficult to adapt to the dynamic changes that may arise in the design process.
(2) It is only possible to represent problems to which the solutions have been previously determined.
(3) It is usually difficult to understand the programmer's interpretation of the problem as the meaning is partly retained by that person.
(4) Processing is efficient.

On the other hand, the semantics of a declarative language are defined as the direct correspondence between the language and the real world problem domain. The characteristics of a declarative language are as follows:

(1) Expressions can be stored to allow for the accumulations of knowledge for future problem solving.
(2) Objects or situations not known a priori can be partially described in advance and therefore it is possible to describe a hypothesis.
(3) The language is modular and therefore modification of the information or knowledge is simple, assuring adaptability and expandability.
(4) The language is more comprehensive than a procedural languages, because natural languages are mostly declarative, and therefore easier to learn.
(5) Shorter development time. Using declarative languages such as *Prolog*, the number of lines of programming code required to solve a typical problem is only a fraction of that required by procedural languages such as *C* or *Fortran*.
(6) Processing is inefficient.

Thus it is easy to recognise that declarative languages offer a number of advantages over the procedural type. The problems concerning efficiency have been recently addressed by the software vendors. In particular, *Visual Prolog*, developed by "Prolog

Development Center" or (PDC), claim that programs created in *Visual Prolog* are very fast, in fact almost as fast as equivalent programs developed in C++, because of the highly optimised compiler [24]. Visual Prolog was originally designed to be an AI language, and is consequently well suited to expert systems and similar AI applications. Visual Prolog is equipped with powerful features (frame or rule-based systems, forward and backward chaining, and pattern matching systems) that allow it to be an effective tool for intelligent product design systems. Therefore in terms of the aspects of implementation, it was decided that the AI approach as apposed to the approach using object-orientated technology, would be utilised in the development of the integrated system. This approach, facilitated through the use of the declarative programming language *Visual Prolog*, was selected primarily for the reasons stated above.

CHAPTER 3

LOGIC BASED PROBLEM SOLVING

3.1 INTRODUCTION

The origins of logical formulations can be traced back to Greek philosophy, where attempts were made to mechanise the human thought process. More recently mathematicians and philosophers such as Leibnitz, Boole, Russel, Frege, Peano, Hilbert, Herbrand, Godel, Tarski, and Gantzen have been involved in the development of mathematical logic [25]. The discipline of mathematical logic, more specifically symbolic logic, is a subset of discrete mathematics. Logic can be used to express statements or facts, infer (inference is the process of deducing new facts from existing ones) additional statements, and to rigorously prove or disprove statements. Through mathematical logic, a valid conclusion can be deduced from a set of axioms (truth statements). Several different types of logic are commonly used today, some examples of these are propositional logic, predicate logic, probabilistic logic and fuzzy logic.

The development of the digital computer has had a profound effect on mathematical logic by providing a standard for mechanical theorem proving. Logic programming and theorem-proving systems utilise deduction procedures through the use of computer programs. Although these systems are very powerful, due to the complexity of automatic theorem proving, they operate relatively slowly. However, compromises on expressiveness and completeness of these systems have resulted in the development of *Prolog*, a logic programming system that is powerful and efficient enough to remain practical. *Prolog* is a logic programming language that is based on first-order logic. It has a fixed execution model in which a set of inference rules is applied to a set of assumptions with a goal of drawing conclusions. As stated previously, one of the goals of AI systems is to mimic intelligent human behaviour such as reasoning. First-order logic is a commonly used model for the formalization of reasoning and knowledge representation. *Prolog* is suitable for developing AI applications such as expert systems

and knowledge representations. Together with *Lisp*, *Prolog* is one of the two most successful AI languages.

3.2 THE MATHEMATICS OF LOGIC

In symbolic logic, mathematical symbols are used to represent statements. A statement (a fact, a proposition or an assertion) is a declarative sentence that is either *true* or *false*. For example, the following statement, in the form of an English-like sentence, can be represented in the functional notation of first-order logic suitable for use in *Prolog*:

"the boy is walking to town" - English-like sentence
walking(boy, town) - functional notation using the predicate "walking"

A compound statement is an assemblage of statements using logical operators such as "*and*", "*not*", "*or*" and "*if*". The equality operator is another commonly used logical connective. A compound statement based upon equality does not mean that the individual statements are equal, but rather they have the same truth value [23]. The logical connective "*if*", also known as the implication operator, is one of the most commonly used, yet conceptually misunderstood, logical operators. It is used to represent conditional statements of the form:

IF antecedent
    THEN consequent

This conditional statement can be written in the semantics of first order logic as follows:

p "antecedent" :-
    q "consequent"

Where *p* and *q* represent certain symbolic or logical statements. Symbolic information represented by "antecedent" and "consequent" statements form the basis of many AI implementations. The term or statement on the left, the antecedent, implies the term on the right, the consequent. If the implication is *true* when the term on the left is *true*, then

the term on the right must also be *true*. The logical properties of implication are best summarized by means of a truth table, as illustrated in Table 3.2.1.

P	Q	p :- q
T	T	T
T	F	F
F	T	T
F	F	T

Table 3.2.1. Implication truth table.

If we assume that implication is *true*, we can utilise this information to constrain the allowable values of the antecedent or consequent. Consider the first row in the truth table. The value of the consequent *q* is constrained to be *true*, due to the fact that the antecedent and the implication are *true*. The importance of this result is that given an implication statement known to have the value *true*, and assuming that the antecedent is also *true*, forces the consequent to be *true*. This forms the logical basis for forward chaining in a rule-based system. The antecedent does not necessarily have to be a simple statement, it can represent a compound statement formed by the conjunction of a number of simpler statements. It must be remembered that implication is used in a weak sense in logic, and that a *false* hypothesis implies any conclusion. For example, the following assertion is *true*:

IF 10 + 10 = 18
THEN All domesticated animals can perform addition

The implication statement *(p :- q)* does in no way imply that *p* causes *q*, as there no cause-effect relationship existing between *p* and *q*. The implication operator merely relates the truth values of antecedent and consequent to the truth value of the implication. From the truth table, the fact that this implication is *true* implies that:

(1) p implies q
(2) q is TRUE if p is TRUE
(3) p is TRUE only if q is TRUE
(4) p is a sufficient condition for q

(5) q is a necessary condition for p

It is often necessary to determine the logical consequences of assertion-based statements where the antecedents and the consequents are themselves compound statements [23]. In knowledge representation, complex statements are often required to be converted into simple mathematical statements. This often results in certain difficulties such as determining the "meaning" and consequently the truth of a statement written in English, deciding the most efficient form of representing complex statements, or understanding the change in the truth value of a statement which has been modified. Consider the following example demonstrating forward chaining using implication. Variables p_i and q_i are used to represent the truth values of the antecedents and the consequents respectively.

Implication 1

p_1 An undergraduate student has to study on the weekends

q_1 The student cannot do anything else but study

Implication 2

p_2 The student cannot do anything else but study (same as q1)

q_2 The student does not have time to play sport

Implication 3

p_3 The student does not have time to play sport (same as q2)

q_3 The student is unhappy

Note that the antecedents of some implications are the same as the consequents of others. The following rule base can be constructed from the set of assertions (each assertion, and not necessarily each component statement, is assumed to be *true*) above:

$p_1 :- q_1$ $p_2 :- q_2$ $p_3 :- q_3$ $p_2 = q_1$ $p_3 = q_2$

The rule base may be rewritten substituting the relevant variables and simplifying the results, thereby producing the following three implications:

$p_1 :- q_1$ $q_1 :- q_2$ $q_2 :- q_3$

To determine the truth of the statement q_3 (the student is unhappy), given the fact that the statement p_1 is *true*, two strategies are employed:

(1) The truth table indicates that if (p :- q) is *true*, and p is *true*, then q is also *true*.
(2) A mechanism to "chain" production of new (*true*) facts from the knowledge that p_1 is *true* to arrive at the truth value of q_3.

Using a model known as forward chaining, the following definitions are developed:

(1) (p_1 :- q_1) and p_1 being TRUE, implies that q_1 is TRUE.
(2) q_1 and ($q_1 = p_2$) being TRUE, yields p_2 is TRUE.
(3) (p_2 :- q_2) and p_2 being TRUE, implies that q_2 is TRUE.
(4) q_2 and ($q_2 = p_3$) being TRUE, yields p_3 is TRUE.
(5) (p_3 :- q_3) and p_3 being TRUE, implies q_3 is TRUE.

3.3 DEDUCTION USING PROPOSITIONAL LOGIC

The connectives described at the start of this chapter allow for the generation or deduction of new information using the rules of logic. Logic provides a rigorous and formal method of creating new knowledge, in the form of logic statements, from existing knowledge through the development of one or more proofs. The process of reaching a conclusion from a set of propositions is called deductive reasoning. Logic enables the development of mathematically justifiable approaches to automating deduction. Approaches are developed for the manipulation of symbolic statements, such that a set *S* of axioms may be used to augment set *S* by forming new *true* statements from those in *S*, to verify the truth of a given statement using *S*, or to determine if any logical inconsistencies exist in *S* or a augmented version of *S*. This is the basis of proof by refutation. Therefore, the concept of "justification of a argument" or "proof of a principle" can be defined on a mathematical basis, where the implication connective assumes an important role as the basis of fundamental data structures and as a proof of validity of a strategy for knowledge expansion. The object is to prove that a new statement, *n*, logically follows from *S*, where *n* is the conclusion. Stated differently, the aim is to determine that the compound implication-based statement

{S} :- n

which represents a deductive argument, is *true*. {S} defines the conjunction of statements in *S*. This is a proof that the reasoning mechanism used to derive n is logically valid or *true*. Consider the following example written as a compound statement involving implication:

(p and (p :- q)) :- q

For a given statement *p* that is known to have the value *true*, and having an implication involving *p* that is also known to have the value *true*, using the principles of logic it can be established that the statement *q* is *true*. This is an example of M*odus Ponens* (MP). In order to prove the validity of arguments based on *Modus Ponens*, it is necessary to prove that the compound statement is a tautology, or in other words that the statement is true for all values of the arguments.

Propositional logic simply serves to determine whether the value of a statement is *true* or *false*. The statements or propositions do not contain any variables. Therefore no mechanism exists that would allow for a particular element of a statement to be substituted with another element. In other words, there is no means of relating a certain portion of the statement with the truth value. This inability of propositional logic to incorporate the use of variables into its statements was overcome by developing an extension to this logic known as predicate logic, defined in the following section.

3.4 PREDICATE LOGIC

A predicate can be described as being a proposition containing variables. Predicate logic is a branch of logic that allows modelling of the truth of statements based upon the values assumed by specific portions of those statements. Two predicates are assumed to be equivalent if they assume the same truth values for all possible variations of the variables. Two features of predicate logic make it suitable for the automation of problem solving, the first being the declarative nature, and the second being the relational nature of the language [26]. The declarative nature of predicate logic assists

problem solving by reducing the available information into a set of declarative statements without much concern as to how the problem will be solved. The processing of these logical statements to obtain valid conclusions is handled by a problem independent inference strategy. This is a viable alternative to conventional procedural problem solving approaches where the human problem solver is responsible for describing the problem and also developing a solution strategy for solving it. The relational nature of predicate logic promotes non-directional computation, which essentially allows for any value of a given logical expression or statement to be solved knowing the remaining values. It also simplifies the modelling process by permitting multiple uses of a formulation. The concept of predicates can be easily related to a more familiar formulation of mathematical equations. Consider, for example, the following algebraic statement:

$$x^2 + 4 = 20$$

This equation can be viewed as a predicate containing the variable x. The universe (or range) of *x* is assumed to be the set of integers. The values *(x = 4)* and *(x = -4)* cause this predicate to be true. In general, and in order to remain consistent with the syntax of *Prolog*, variables in predicates are represented by capitalising the first letter, and values of variables are depicted in lower case. Although predicate logic can be used to represent mathematical statements, it's strengths really lie in it's ability to represent non-numeric or symbolic statements. Consider coding the following statement, "the television in the lounge is on". All three of the following are valid interpretations in predicate logic, with corresponding general binary relations.

on(television, lounge) - on(What, Where)
television(lounge, on) - television(Where, Status)
lounge(television, on) - lounge(Equipment, Status)

It is often important to be able to represent more general relations. In some situations, a predicate is formulated whose truth value is unchanged irrespective of the values of the variables. Therefore, a necessity arises for the manipulation of quantified statements. For example, the statement "*All professors are underpaid*", relates to an entire set or class with the concept of underpaid. The statement "*Some professors are engineers*", is

more general, but requires the need to establish that there exists at least one professor who has an engineering background. These two statements are examples of categorical proposition. In order to represent the allowable universe or range of the variables in these quantified statements, two useful quantifiers have been devised. The first is the *universal quantifier* which is responsible for representing an entire set or class. The second is the *existential quantifier* which constrains at least one member of a class.

Before the use of logic as a mechanism for problem solving can be discussed, a few commonly used definitions need to be stated.

> A *term* is a constant, a variable, or an n-variable function.
> A *function* is an n-tuple of terms prefixed by a name (or functor) that satisfies the definition of a function.
> A *predicate* (a more formal representation of the definition stated previously), is an n-tuple of terms, prefixed by a predicate name.
> A *well-formed formulae*, abbreviated wff, is either a predicate or a (possibly compound) statement. A sentence or a closed formulae (a sentence containing no variables), is a wff in which every occurrence of a variable is within the scope of a quantifier for that variable.

Two important topics, often encountered in problem solving using logic, are *resolution* and *unification*. Resolution (or proof by refutation) is a powerful and indirect method of inference seeking to yield new clauses from an initial set. The functionality of resolution can be described as attempting to prove that a set of clauses are logically inconsistent. In other words, resolution produces a logical contradiction. If the process of resolution fails to find a contradiction, then the negative of what we seek to prove is logically consistent, and thus the clause cannot be *true*.

Unification is a systematic procedure for instantiation of variables in wffs. It is the process of attempting to make two expressions identical by finding suitable substitutions of variables in these expressions. Given that the truth value of a predicate is partly determined by the values assumed by their arguments, controlling the instantiation of the values therefore provides a means of validating the truth values of compound statements containing predicates. Unification is fundamental to most

inference strategies in AI. For example *Prolog* execution is essentially based on the unification mechanism. The basis of unification is substitution

3.5 LOGIC PROGRAMMING WITH PROLOG

Prolog is a goal orientated programming language that is vastly different from the conventional procedural (how - type) languages such as *Fortran*. *Prolog* is a language that beaks away from the procedural languages, and encourages the programmer to rather focus on describing the situations and problems, not on a method for solving them. In this sense can be described as a declarative (what - type) language. It is a programming language centred around a small set of basic mechanisms, including pattern matching, tree-based data structuring and automatic back-tracking. This small set constitutes a powerful and flexible framework [27].

The name *Prolog* is an abbreviation for *programming in logic*. It origins can be dated back to the early 1970's. *Prolog* has not been without it's historical controversies. It rapidly gained popularity in Europe and Japan as a practical programming tool. However, in the United States, *Prolog* was generally accepted with trepidation due to a number of reasons. There were reservations about adding practical features, not directly related to logic, to the programming language. It was also falsely understood that only certain solution algorithms, such as *backward chaining reasoning,* could be programmed in *Prolog*. In reality though, *Prolog* is a general programming tool that allows for the programming of any algorithms.

3.5.1 DEFINING RELATIONS BY FACTS

Prolog is especially well suited to solving problems concerning objects and relations between objects. Consider the following example of a family tree relation, illustrated in Figure 3.5.1.1. The family tree is represented by a simple knowledge-base containing six clauses, each in the form of a fact containing two arguments:

```
parent(pam, bob).
parent(tom, bob).
```

parent(tom, liz).

parent(bob, ann).

parent(bob, pat).

parent(pat, jim).

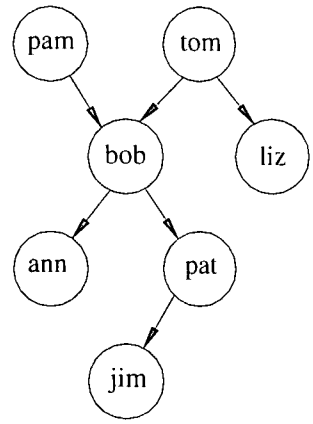


Figure 3.5.1.1. Graphical representation of the family tree.

As described previously, variables are represented by capitalising the first letter, and values of variables and predicate names are represented with lower-case letters. Each of these clauses declares one fact about the parent relation. For example, *parent(pam, bob)* is a particular instance of the parent relation. Knowing this information, *Prolog* could be queried about certain relations. Is *bob* a parent of *pat*? This query could be communicated with the *Prolog* system by typing

parent(bob, pat).

under the goal section of the program. Having found this as an asserted fact in the program, *Prolog* would answer (yes). A further query could be

parent(pat, bob).

and Prolog would answer (no), because the program does not contain any knowledge about *pat* being a parent to *bob*. More interesting questions could also be posed, for example:

parent(X, bob).

where *X* represents a variable, and *Prolog* would respond with the following conclusions:

X = pam X = tom

This example program could be asked some more complicated questions such as who is a grandparent of *jim*. Due to the fact that no grandparent relation is contained in the knowledge-base, this query has to broken down into a two-step process as described below:

- Who is the parent of *jim*? Assume that the parent is represented by the variable *Y*.
- Who is the parent of *Y*? Assume that this parent is *X*.

Such a composed or compound query can be constructed in *Prolog* as a combination of two simple ones separated by a disjunction (and) connective :

parent(Y, jim), parent(X, Y).

In other words, find the parent *Y* of jim, and subsequently find the parent *X* of *Y*. *X* is therefore the grandparent of jim. This example program has helped illustrate some important factors about *Prolog* programming:

(1) The user can easily query the system about relations that have been defined in the knowledge-base.
(2) A *Prolog* program consists of clauses, each terminated by a full stop.
(3) The arguments of relations can be: concrete objects (or atoms), constants, or general objects (or variables).
(4) Questions to the system consist of one or more goals.

3.5.2 DEFINING RELATIONS BY RULES

Maintaining the original parent relation knowledge-base from the previous section, additional information could be added to this knowledge-base in the form of an

offspring relation (inverse of the parent relation). This could be done by simply adding facts about the offspring relation:

```
offspring(liz, tom).
```

However, the offspring relation could be more elegantly defined by using the fact that it is the inverse of the parent relation and that the parent relation has already been defined. The relation would be defined through the use of the following logical statement:

```
For all X and Y,
        Y is an offspring of X if
                X is a parent of Y.
```

This statement can be represented using clauses, in the semantics of *Prolog*, as follows:

```
offspring(Y, X) :-
        parent(X, Y).
```

Clauses of this form are generally referred to as *rules*. There is an important distinction between facts and rules. Facts are always, unconditionally *true*. On the other hand, rules specify statements that are *true* if some condition is satisfied. A rule can be divided into a head (the conclusion part on the left of the implication connective), and a body (the conditional part on the right of the implication connective). If the condition *parent(X, Y)* is *true*, then a logical consequence of this is *offspring(Y, X)*. The workings of a rule in *Prolog* can be best described through a simple example, starting with the query:

```
offspring(liz, tom).
```

Because there are no facts about offspring in the program, the only method of solving this query is by using the offspring rule. The rule is general in the sense that it can be applied to any arguments *X* and *Y*, and therefore can also be applied to specific objects such as *liz* and *tom*. To apply the rule to *liz* and *tom*, the variables *Y* and *X* become instantiated with *liz* and *tom* respectively.

Y = liz and X = tom

This instantiation results in the following special case of the rule:

```
Offspring(liz, tom) :-
        parent(tom, liz).
```

And the conditional part of the rule has become:

```
parent(tom, liz).
```

Prolog then attempts to determine whether the conditional part is *true*, and therefore the original goal

```
offspring(liz, tom).
```

is then replaced by the subgoal:

```
parent(tom, liz).
```

The new subgoal, *parent(tom, liz)*, has already been defined as a fact contained in the knowledge-base of the program, and therefore the conclusion part of the rule is proven to be *true*, and *Prolog* will consequently answer the query with (yes). Rules provide an important mechanism in a knowledge-based system, as they allow for the inference of additional facts, not originally contained in the knowledge-base. Stated differently, this mechanism allows for the logical deduction of further information which can be subsequently added to the knowledge-base, thereby increasing the available knowledge.

3.5.3 RECURSIVE RULES

As another extension to parent relation knowledge-base, the predecessor relation will be added. This relation, like the offspring relation will also be defined in terms of the parent relation. The whole definition can be expressed as two separate rules. The

function of the first rule is to determine the direct or immediate predecessor and the second rule, the indirect predecessors. The indirect predecessor rule can be stated as follows, *X* is an indirect predecessor of *Z* if there is a parentship chain of people between *X* and *Z*. For example, t*om* is a direct predecessor of *liz*, and an indirect predecessor of *pat*. This concept can be seen in Figure 3.5.3.1. The first rule is straight-forward and can be formulated as follows:

For all X and Z,
    X is a predecessor of Z if
    X is a parent of Z.

The equivalent *Prolog* representation is:

```
(1) predecessor(X, Z) :-
        parent(X, Z).
```

The second rule is slightly more complicated due to the chain of parents that could possibly need to be searched to find the solution. The difficulty lies in the fact that this chain of parents does not have a specified length. In other words, the number of predecessors can take the value of any positive integer. Therefore, the predecessor relation is emphatically defined in terms of itself. This concept, known as *recursion*, is explained more precisely by the following statements:

For all X and Z,
    X is a predecessor of Z if
    There is a Y such that
(1) X is a parent of Y
(2) Y is predecessor of Z.

The corresponding *Prolog* clause representing the above statement is:

```
(2) predecessor(X, Z) :-
        parent(X, Y),
        predecessor(Y, Z).
```

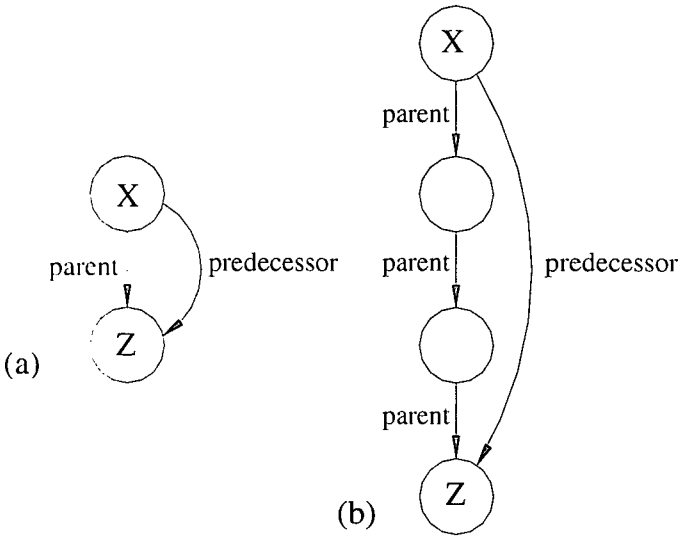


Figure 3.5.3.1. (a) X is a direct predecessor of Z. (b) X is an indirect predecessor of Z.

Rules (1) and (2) form the complete program for the predecessor relation. As stated previously, rule (1) handles the direct predecessors and rule (2) handles the indirect predecessors. The most important aspect of rule (2) is the use of the predecessor itself in its definition (recursion). Recursive programming is one of the fundamental principles of programming in *Prolog*. In fact, it is not possible to solve tasks of significant complexity without the use of recursion, as the programmer does not always have prior knowledge of the depth or complexity of the problem at hand.

3.5.4 REPRESENTATION USING LISTS

List processing, or handling objects that have an arbitrary number of elements, is a powerful technique often used in logic programming [28]. A list is basically an object containing an arbitrary number of other objects within it. In this respect it is similar to an *array* used in other languages. However, unlike an array, the size of a list does not have to be previously declared. There are other ways of combining several objects into one argument, but if the number of arguments is not initially known, this task can pose great difficulty. Lists can be defined to represent different types of information, for example: numbers, names, compound data objects or even other lists, as shown by the following objects (lists):

[1, 2, 3, 4]

[tom, liz, pat, bob, sam]

[parent(pam, bob), parent(tom, bob)]

[[20,30], [1,2,3,4], [101]]

Each of the items in the list are separated by a comma and are commonly known as elements. As mentioned above, elements can represent a variety of different objects. A list is essentially a recursive compound object which can either be empty or non-empty. An empty list is represented by *[]*, while a non-empty list can be viewed as consisting of two things:

(1) The first item, known as the *head* of the list. The *head* of a list is always the first element in the list

(2) The remainder of the list is referred to as the *tail* of the list. The *tail* of a list is always a list.

For example, in the list *[tom, liz, pat, bob, sam]*, the head of the list is represented by *tom*, and the tail of the list is represented by the list *[liz, pat, bob, sam]*. The head of the list, is this case *liz*, can then be removed from the tail leaving a new tail *[pat, bob, sam]*. This process can be continued until the tail contains an empty list. In other words, if the list was reduced to *[sam]*, the head of this list would be *sam*, while the tail would be an empty list *[]*.

The information contained in the preceding sections covers only the very basics of programming in *Prolog*. However, having this knowledge on the fundamentals of *Prolog* programming will allow a greater understanding of the "*program structure*", described in chapter seven, which has been developed entirely in the language of *Prolog*.

CHAPTER 4

THE FINITE ELEMENT METHOD

4.1 INTRODUCTION

Recent technological developments have totally revolutionised the engineering design process. Most modern personal computers are equipped with sophisticated hardware and powerful graphics processes that, through the use dedicated software applications, enable the design and development of complex assemblies. A method or approach that lends itself increasingly well to computer implementation, due to the sheer level of computational intensity involved, is the finite element (FE) method. The FE method was originally developed by engineers who realised there was a need for a method of solving complex stress analysis problems that could not be solved using traditional techniques. In general, the FE method is based on a theory that views the geometry of a physical component as an assembly of discrete building blocks (or elements). The application of the method involves dividing (discretising) the body of the component into an optimum number of elements, joined only at specific locations known as nodes, and using them as a basis for computations [29].

In all applications the analyst attempts to calculate a certain field quantity: in stress analysis this value could be a displacement or stress field, in thermal analysis a temperature field or a heat flux, and so forth. The FE method is a means of getting a numerical solution to a specific problem. A FE analysis does not provide a formula as a solution, nor does it solve a particular class of problem, but it rather provides an approximate solution (unless the problem is so simple that an exact formula is available) to a given problem [30]. The elements that are used to define the component are of a finite size and hence the name of the method. Generally speaking, the smaller the size of the element, the smaller the errors due to approximation, and as a result the solution obtained approaches the true solution with a greater degree of accuracy.

The FE method can be regarded as a *piecewise* polynomial interpolation, which essentially means that over an element, a field quantity such as a displacement is interpolated from the individual values obtained at the nodes. By connecting the elements together, the quantity becomes interpolated over the entire structure in a *piecewise* fashion. The power of the FE method lies in its versatility, specifically in its ability to analyse models having arbitrary shape, support (boundary conditions) and loads. Such a degree of generality does not exist in classical analytical methods.

The origins of the FE method can be dated back to around 1941, when Hrenikoff introduced the so-called framework method, in which a plane elastic medium was represented as a collection of bars and beams [29]. In 1943, Courant [31] presented work using *piecewise-continuous* functions, defined over a sub-domain, to approximate an unknown function. Courant used an assemblage of triangular elements and the principle of minimum total potential energy to study the St Venant torsion problem. Although certain key features of the finite element method can be found in the work presented by Courant and Hrenikoff, its formal inception is attributed to Argyris and Kelsey [32], and Turner et al. [33]. The term *finite element* was first used in 1960, and since its inception, the literature on the finite element method has grown exponentially. Today there are numerous books and journals that are devoted to the theory and application of the FE method [34].

With regards to the development of the integrated design - analysis system described in this work, the commercially available FEA program MSCN4W was used to provide the analysis capabilities. As mentioned previously, the objective of this research was to develop an efficient integrated system by providing a seamless integration between the product data model and the FEA code. Therefore, by creating an integrated system that required minimal user interaction, the only aspect of the FE process that the designer / engineer could manipulate, was the size of the elements (mesh density) used to represent the physical model. The remainder of the FE processes were automatically controlled by *the program* in conjunction with MSCN4W (from here onwards, the term *the program* refers to the program used to provide the design - analysis integration). By having the ability to adjust and hence refine the finite element mesh, the designer / engineer can effectively check the accuracy of the model by performing convergence tests, which will be discussed in a later section.

Although the analysis program MSCN4W controls a large fraction of the FE analysis, it was essential that the researcher understood the fundamentals of the FE method to allow for a suitable integrated system to be developed. There are literally hundreds of books and journals dedicated to the finite element method. Some of these books provide introductions and applications, while other work focuses on some of the more complex, specialised topics in FE modelling. However, to provide an in depth description of the FE method was not the intention of the work presented here. Instead, it was decided that an appropriate selection of material would rather incorporate some of the more practical aspects of FE modelling. Aspects such as: the application of loads and boundary conditions, convergence testing and automatic mesh generation. Nevertheless, a brief description of the FE method is included to provided a basic understanding of the relevant concepts and terminology.

4.2 THE FINITE ELEMENT PROCESS

A designer / engineer should always remember that the FE method is a way of implementing a mathematical theory of physical behaviour. Thereby the assumptions and limitations of the theory must not be violated by expecting the software to perform unreasonable or unrealistic functions. In many situations, the solution method by which the theory is implemented should be understood at a fundamental level, to avoid choosing an unsuitable solution method for a particular problem and to avoid misinterpreting erroneous results. Despite having a basic understanding it is still easy to make mistakes in describing the problem to the computer program [30]. Although the development of an integrated design - analysis system can effectively eliminate human error, the other problem areas described above are more difficult to over come. Therefore it is still essential that the user of such an integrated system has a good physical grasp of the problem at hand allowing judgement to be made as to whether the results are to be trusted or not. Bearing this in mind, this section serves to introduce some of the fundamental ideas that form the basis of the FE method.

The basic steps involved in the finite element analysis of a problem can typically be divided into the following areas [34]:

(1) *Discretisation* – discretisation (or representation) of the given domain into a selection of preselected finite elements. Finite elements resemble fragments of the structure. Nodes appear on element boundaries and serve as connectors that fasten the elements together. Nodes are always located at the vertices of an element, and in some cases when increased accuracy is required, nodes can also be included along the edges of the element (mid-side nodes). Each element is restricted in its mode of deformation to avoid unrealistic behaviour such as: the sliding of adjacent elements or even the formation of gaps between element boundaries during deformation. These are commonly known as conditions of *compatibility*. Discretisation is performed using the following steps:

a. Constructing the finite element mesh of the preselected elements.
b. Numbering the nodes and elements.
c. Generating the geometric properties (for example coordinates and cross-sectional areas) needed for the problem.

In terms of step a, there are several requirements for the quality of a mesh. For example, the proportions of the elements should be as close as possible to those with good aspect ratios (the aspect ratio is the ratio of the length of one side of an element to its adjacent side). In addition, the change in size between elements defined by the mesh should be gradual [35].

(2) *Derivation of the element equations* – the element equations for all typical elements in the mesh are defined.

a. Initially the variational formulation of the given differential equation over the typical element is constructed.
b. In order to obtain the element equations, assume that a typical dependent variable u is of the form

$$u = \sum_{i=1}^{n} u_i \varphi_i \tag{4.2.1}$$

and then substitute it into 2a to obtain the element equations in the form of

$$[K^e]\{u^e\} = \{F^e\} \tag{4.2.2}$$

c. Derive or select, if already available in the literature, the element interpolation function ψi and compute the element matrices.

(3) *Assembly of the element equations* – assemble the element equations to obtain the equations describing the whole problem.

a. Identify the inter-element continuity conditions among the primary variables. For example the relationship between the local and global degrees of freedom – connectivity of elements.
b. Identify the equilibrium conditions among the secondary variables. For example the relationship between the loads specified in local and global coordinates.
c. Assemble the element equations using steps 3a and 3b.

(4) *Application of the boundary conditions* – apply the appropriate boundary conditions to the FE model.

a. Identify the specified global primary degrees of freedom (displacements).
b. Identify the specified global secondary degrees of freedom (if this step has not already been performed in step 3b).

(5) *Solution of the assembled equations.*

(6) *Postprocessing of the results.*

a. Compute the gradient of the solution or other desired quantities from the primary degrees of freedom computed in step (5).
b. Represent the results in tabular and / or graphical form.

In terms of the integrated system considered here, once the script containing all the relevant information has been created and channelled to the analysis code, the above steps are automatically performed and the results are tabulated for the user's perusal. At this stage the model can be reanalysed with a finer mesh to check for the validity of the results obtained. This concept known as *convergence* will be discussed in a later section.

4.3 LOADS AND BOUNDARY CONDITIONS

The application of loads and boundary conditions is an important aspect of the FE modelling process. A model may be accurately defined in terms of the geometry and the selection of a suitable element type to portray the deformation characteristics, but the application of spurious loads and / or boundary conditions will result in unrealistic behaviour and erroneous results.

Mechanical loads are usually applied in the form of concentrated loads at nodes, surface traction and body forces. Traction and body force loads cannot be applied to the FE model directly, but instead they must be converted to equivalent loads acting on specified nodes. This load conversion is usually performed in one of two ways. The first method involves applying *work-equivalent* loads. In application this requires replacing the actual loads by nodal loads that perform the same work as the actual loads in moving through the nodal displacements. The second type of loading is known as *lumping*, which essentially involves applying the same load to each node. Work-equivalent loads usually provide greater accuracy than lumping, although in either case the exact results are approached with increased mesh refinement. Today most software is capable of automatically calculating equivalent nodal loads of proper magnitude and direction, from distributed loads of known intensity and orientation, and combining them at shared nodes [30]. In most circumstances standard software is not designed to accept non-nodal concentrated loads as input data. In practice, the mesh would usually be arranged so that there is a node at each location of concentrated load application. However, for the purpose of the research considered here, the integrated system being predominantly a conceptual tool allowed for the fact that certain approximations could be assumed. One of these approximations was the application of non-nodal loads, which are evidently applied to the node whose location is closest to that of the concentrated load. Naturally as the mesh density increases, so does the accuracy of this assumption. According to classical linear theories, a point loaded by a concentrated normal force causes:

(1) A finite displacement and finite stress in a beam.
(2) A finite displacement and infinite stress in a plate.

(3) An infinite displacement and infinite stress in a three-dimensional solid.

Physically, a concentrated load does not exist in reality, it is a mathematical representation of distributed load of high intensity acting on a small area. However, in a FE model a concentrated load will never cause infinite displacements or stresses, although infinite values will be approached as the mesh is repeatedly refined [30]. The integrated system considered here was designed for the analysis of *linear static* problems. The term linear refers to the fact that the loads applied to the FE model maintain their original orientations in space, regardless of the magnitudes of the computed displacements. In other words, a linear analysis is only accurate for small deformations of the FE model. If the applied loads caused considerable deformation to the model, a more accurate solution would be obtained by performing a non-linear analysis using *follower forces*, whose directions change as the structure deforms. The term static refers to an analysis where the load application does not vary with time.

Boundary conditions, also commonly known as constraints, are used to simulate the allowable (or realistic) movement of the FE model in a predefined space. However, misrepresentation of the input data, even minor changes to the support conditions, can have a profound effect on the computed results. Some support conditions are dictated by FE technology rather than by physical considerations. A constraint, such as an imposition of a zero displacement or rotation, must be defined at a node and not in between two nodes. Also the degrees of freedom that are not defined in a FE model must be suppressed, irrespective if they are on the boundary of the FE model or not. For example, typical plane elements resist two in-plane translational degrees of freedom per node, however software makes allowances for six degrees of freedom at every node. Therefore, to prevent singularity of the structure's stiffness matrix, the rotational and out-of-plane translational degrees of freedom must be suppressed, regardless of loads being applied to these degrees of freedom or not.

A structure that is either unsupported or inadequately supported will have a *singular* stiffness matrix, and as a result the FE software will be unable to solve the element equations for the nodal degrees of freedom. The term *stiffness matrix* originates from structural analysis. Early applications of the FE method were similar to matrix analysis of structures, and consequently the term was used to describe the matrix relation

between force and displacement. FE terminology defines two stiffness matrices: the *local* stiffness matrix relating to the individual element, and the *global* stiffness matrix corresponding to the assembly of all the local stiffness matrices. The latter defines the stiffness of the entire system [36]. To prevent singularity, supports must be sufficient to prevent all *rigid body motion* (motions that produce no deformations of the structure).

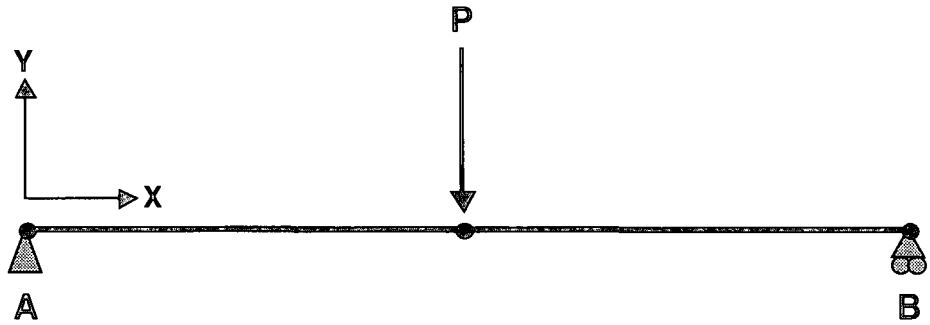


Figure 4.3.1. FE model of simply supported beam.

To demonstrate how loads and boundary conditions are applied to a FE model, consider the simply supported beam illustrated in Figure 4.3.1. The FE model has been created using two two-node beam elements to allow for the load to be located centrally. The load would be defined as having a magnitude P, and acting in the Y direction. To define the pin support at node A, the translational degrees of freedom in the X and Y directions would be suppressed (the rotational degree of freedom about the z-*axis*, perpendicular to the page, would not be suppressed to allow for the necessary rotation provided by the pin support). In terms of the boundary conditions at node B, only the degree of freedom associated with the Y direction would be suppressed, to allow for the required rotation and translation provided by the roller support. Although this example involved a simple two-dimensional model, the same principles apply to models created in three dimensions. However, the three-dimensional model would associate six degrees of freedom, as apposed to three as in this example, to each of the nodes.

4.4 CONVERGENCE

Irrespective of the analysis method used, a mathematical representation of the physical model is always analysed, and this introduces what is known as *modeling error*. This

type of error is reduced by selecting an element that can most accurately replicate the true behaviour of the structure. Consider an axially loaded tapered bar for which a mathematical model, based on a state of uniaxial stress, is available. An analytical solution can then be easily calculated and used as a means of comparison for a subsequent finite element analysis. The bar can initially be represented by discretising the mathematical model into a number of two-node elements of constant cross-section. Each element has a certain length and fixed area and accounts only for constant uniaxial stress across its length. Intuitively, it would be expected that as the number of elements were increased to model the length of the bar, the exact displacement would be approached. However, the exact displacement would never be reached due to *discretisation error*, which exists because both the physical and mathematical models have an infinite number of degrees of freedom, whereas the FE model has a finite number of degrees of freedom [30].

It is difficult, if not impossible to predetermine the exact number of elements required in a FE model. Consider a FE model that is created and analysed twice, the second time using a more refined mesh. The second FE model, having a greater number of elements, will have less discretisation error than the first, and will also represent the geometry better if the model contains curved edges. If the analyses of both models yield similar results, then it can be assumed that both results have only small errors. Or it might be necessary to perform a number of consecutive analyses, each time reducing the element size. By studying how the sequence converges, the designer / engineer should be able to estimate with confidence that the results obtained from the finest mesh are in error by no less than a predetermined amount.

Therefore it can be concluded that provided a *valid* element is used, the solution will converge toward the exact result with increased mesh refinement. An element is termed valid if it can pass the requirements of what is known as a *patch test* [30]. A patch test essentially indicates that when an element is used in a mesh, rather than in isolation, it is able to display: (1) a state of constant strain, (2) rigid body motion without strain and (3) compatibility with adjacent elements in states of constant strain.

4.5 MESH GENERATION

4.5.1 INTRODUCTION

This section is concerned with the discretication of the physical model into a model represented by a finite number of elements. The importance of this section cannot be over emphasised, as the quality of the mesh in the FE model has a major impact on the validity of the results obtained. Although the integrated system was intentionally designed to limit the amount of user intervention, including intervention during the creation of the finite element mesh, it was important to have at least a basic understanding as to how a finite element mesh is constructed in a typical FEA application.

As mentioned previously, the mesh generation process is automatically performed by the analysis code MSCN4W, which is equipped with a variety of powerful mesh generation algorithms to suit the automatic meshing requirements of a wide range of model types. The integrated system considered here was designed to accommodate geometry either composed of surfaces or solids, and as a result three different element types are typically utilised in the model creation. It is important to mention the type of elements used in the model creation as this generally dictates the method used to generate the mesh. In terms of plane models, the predominant element to be used is a *four-node linear quadrilateral.* However, in areas of the model where a quadrilateral element cannot be defined due to geometric constraints, a *three-node linear triangle* element is used accordingly. If the designer / engineer requires higher levels of accuracy (albeit at the expense of solution time and model size), the default quadrilateral and triangular elements can be modified to include mid-side nodes, transforming these elements into eight and six-node *parabolic elements* respectively. The essential difference between *linear* and *parabolic elements* lies in the mode of deformation.

When a linear element deforms, its sides deform as straight lines, whereas the sides of the *parabolic element* deform as quadratic curves and therefore allow a more realistic

approximation of the behaviour of the element [30]. If the component is represented by a solid model, then naturally the model requires to be meshed using solid elements. In this case, a *four-node linear tetrahedral element* is used irrespective of the geometry of the component. Once again, if increased accuracy is required, then these elements can be replaced by the equivalent *tetrahedral elements* having mid-side nodes. Triangular and tetrahedral elements are well-suited to the representation of irregular boundaries and allow a change in element size without excessive distortion. Therefore they are suitable candidates for mesh generation in the FE method. On the other hand, quadrilateral and *hexahedral* (brick-shaped) elements usually provide a more accurate representation of the physical model's behaviour but are more difficult to automatically generate. Unfortunately, discretising a computer representation of a physical model into a FE model is a problem that is more difficult than it originally appears. A useful mesh satisfies constraints that almost seem contradictory [37]. Nevertheless, there are a number of objectives that an effective mesh generation facility should surpass.

The first goal is the ability to accurately represent complex geometrical shapes, including shapes with curved surfaces and interior boundaries. A second goal is to offer as much control as possible over the size of the elements in the mesh. A last objective is the ability to create elements that have a regular shape, due to the reason that elements with large or small angles degrade the quality of the mesh.

There are basically two kinds of meshes which are characterised by the connectivity of the nodes. *Structured meshes*, also referred to as *grid point meshes*, have a regular connectivity which means that almost all the nodes have an equal number of adjacent elements. The algorithms used for these meshes generally involve complex iterative smoothing techniques that attempt to align the elements with the model's boundaries. Structured meshes are most commonly used in the *computational fluid dynamics* field, where there are severe restrictions on the allowable element shapes [38]. *Unstructured meshes* have irregular connectivity allowing each node to have a different number of adjacent elements [39]. For simple shapes, the choice between structured and unstructured meshes is determined by the discretisation method. Yet, for complex shapes, unstructured meshes are preferred to structured meshes for their efficiency and levels of automation. For this reason the focus of this section will be on the use of unstructured meshing techniques. The following three unstructured mesh generation

techniques are concerned with the decomposition of the original geometry into triangle and tetrahedral elements [38].

4.5.2 OCTREE

This technique was developed in the 1980s by a research group at Rensselaer [39, 40], and relies on an approach that recursively subdivides a geometric model, constructed from individual cubes, until the desired resolution is reached (the decomposition equivalent method in two dimensions is known as the *quadtree*). When a cube, or a square in the two-dimensional case, intersects the surface of the model, an irregular cell is created. This process does often require a significant number of surface calculations to determine the possible locations of intersections. The octree technique does not match a predefined surface mesh, as is usually the case with an *advancing front* or *Delaunay* mesh, but rather facets are formed whenever the boundaries of a cube (or square) intersect the model's boundary. Naturally the final mesh changes with the orientation of the octree structure. Smoothing and cleanup operations can be applied to improve initial element shapes.

4.5.3 DELAUNAY

Delaunay criterion represents the most popular of the triangular and tetrahedral meshing techniques. The Delaunay criterion, also known as the *empty sphere* property, simply asserts that any node must not be contained within the *circumsphere* of any tetrahedron within the mesh. A circumsphere can be defined as the sphere that passes through all four vertices of a tetrahedron. Although the Delaunay criterion has been known for many years, it was not until the work of Lawson [41] that the criterion was finally applied to the development of algorithms to triangulate a set of vertices. The Delaunay criterion is itself not an algorithm for mesh generation, but it does provide the appropriate criterion for which to connect a set of existing points in space. As a result it is necessary to provide a method for generating node locations within the geometry, and it is this method that distinguishes one Delaunay algorithm from another. The methods of inserting nodes into the existing geometry can generally be divided into the following two classes:

(1) *Point insertion* – the point insertion method can be roughly divided into the following six subgroups. (i) The simplest approach is to define nodes from a regular grid of points, whose size can be defined by a user specified sizing function. (ii) Another approach requires that nodes are recursively inserted at triangle or tetrahedron centroids providing that the underlying sizing function is not violated. (iii) An approach proposed by Chew [42] is to define new nodes at element circumcircle / sphere centers. (iv) A similar method is the so-called *Voronoi-segment* point insertion method. A Voronoi-segment can be defined as the line segments connecting two adjacent triangle or tetrahedral centers. The new node is inserted at a point along the Voronoi-segment. (v) Another method is an *advancing front* approach as proposed by Marcum and Weatherill [43]. Nodes are inserted incrementally from the boundary towards the interior. Each facet is examined to determine the most suitable location for the insertion of the forth node on the interior of the existing Delaunay mesh. (vi) The last type of method is commonly referred to as the *point insertion along edges* approach [44]. A set of candidate vertices are defined along the existing internal edges of the triangulation. Nodes are then carefully (ensuring that they are not placed to close to an existing node) inserted in an incremental manner.

(2) *Boundary constrained triangulation* – in many FE applications, there is a requirement that an existing surface triangulation be maintained. In most Delaunay approaches, a three-dimensional tessellation of the nodes on the surface geometry is produced before internal nodes are generated. In this process there is no guarantee that the surface triangulation will be satisfied. A common approach is to tessellate the boundary nodes using standard Delaunay algorithms without regard for the surface facets. A second step would then be employed to recover the surface triangulation, resulting in a triangulation that would no longer be strictly Delaunay, and hence the name boundary constrained Delaunay triangulation. In two dimensions the edges of a triangulation are recovered by iteratively swapping triangle edges. In three dimensions the process is more complex, since after recovering all edges in the surface triangulation there is no guarantee that the surface facets will be recovered. Two different approaches have been proposed by Weatherill and Hassan [45] and George et al. [46] for the recovery of boundary edges. (i) Edges are recovered by performing a series of tetrahedral transformations

by swapping two adjacent tetrahedrals for three. (ii) The second approach involves an edge and face recovery phase. Nodes are inserted directly into the triangulation wherever the surface edge or facet cuts non-conforming tetrahedrals.

4.5.4 ADVANCING FRONT

The *advancing front* or *moving front* is another popular approach to triangle and tetrahedrals mesh generation that was developed at the George Mason University and the University of Hong Kong [47]. This method progressively builds tetrahedrals inward from the triangulated surface, forming an active front where new tetrahedrals are created. As the algorithm progresses, the front advances to fill the remaining geometry with triangles or tetrahedrals. Intersection checks are also included to ensure that the triangles or tetrahedral do not overlap as opposing fronts advance towards each other. Another feature which can be incorporated is a sizing function to control the element sizes. When the geometry of the model allows it, quad mapped meshing usually produces more accurate results. Although mapped meshing is normally considered a structured method, it is quite common for unstructured codes to provide mapped meshing options. For mapped meshing to be applicable, the opposite edges of the area to be meshed must have an equal number of divisions. Unstructured quad meshing algorithms can generally be grouped into two main categories, which will be described in the following sections [38].

4.5.5 INDIRECT QUAD MESHING METHODS

One of the simplest methods for *indirect quadrilateral mesh* generation is to divide all triangles into three quadrilaterals. This method guarantees an all-quadrilateral mesh, but there is a high probability that the mesh will contain irregular shaped elements. Another approach is to combine adjacent pairs of triangular elements into a single quadrilateral. Although this method addresses the problem associated with the above method by increasing the quality of the elements, a large number of triangular elements may be omitted. This method can be improved by determining a specific order in which the elements are combined. Work as been presented by various authors attempting to

increase the number and quality of the quads. Some of the developed approaches have focused on local element splitting and element swapping strategies, while other work has included the advancing front approach used over the initial triangles. Another method recently introduced by Owen [38] is known as quad morphing. This method also utilizes an advancing front approach to convert triangles to quads, but it is also able to appreciably reduce the number of irregular shaped elements.

4.5.6 DIRECT QUAD MESHING METHODS

Indirect methods have the advantage of being very fast, but a typical drawback is the number of irregular shaped elements produced during mesh generation. Consequently, many direct methods have recently been proposed. These methods can generally be divided into two main categories: the first relies on some form of decomposition of the model into simpler regions and the second uses a moving front approach for the direct placement of nodes and elements.

(1) *Methods of decomposition* – Baehmann et al. [48] proposed one of the first methods using decomposition known as the *quadtree decomposition technique*. The two-dimensional space is initially decomposed into a quadtree based on local feature sizes. Quadrilateral elements are then fitted into the quadtree regions, adjusting nodes to maintain boundary conformity. Talbert introduced another decomposition technique where the model is recursively subdivided into simple polygonal shapes. Quadrilateral elements are then inserted into the resulting polygons. An immediately different approach uses what is known as a *medial axis decomposition* [49]. The medial axis can be thought of as a series of curves generated from the midpoint of a *maximal circle* as it rolls through the area. Once the area has been decomposed into simpler regions, a template is employed to insert quadrilateral elements into the model.

(2) *Advancing front methods* – Zhu et al. [50] proposed a meshing algorithm that initially places nodes on the boundary of the model, and then individual elements are formed by projecting edges towards the interior of the model. Another method introduced by Blacker et al. is referred to as the *paving algorithm*. It presents a method for forming complete rows of elements starting from the boundary and then

progressing towards the interior. The paving algorithm is currently implemented in several commercial packages including *MSC Patran*.

A number of mesh generation techniques have been presented, ranging from the more simple unstructured triangular mesh approaches to complex methods employed to generate accurate quadrilateral elements. Although there is an extensive collection of alternate approaches to automated mesh generation, the more common methods have been briefly described in this section as these are the method that are most likely used by the analysis code in this integrated system.

CHAPTER 5

CONTACT SURFACE DETECTION

5.1 INTRODUCTION

During the initial stages of this project, in light of the final objective of developing an integrated virtual prototyping system, the importance of the preceding chapters was immediately apparent, resulting in the relevant literature being reviewed. However, it was not until the early developmental stages of the integrated system that the importance or necessity of this section was fully realised. At this stage the basic conceptual ideas behind the representation of the components of a mechanical assembly had been roughly established. It had been decided that the inter-component relationships would be defined using a hierarchical system, where the components of the assembly could be depicted as the nodes of a tree-like structure, and the physical relations existing between the components could be associated with lines connecting these nodes. Using this hierarchical system, the locations of the components relative to each other could be efficiently calculated. But one problem still remained. A technique, that would allow for the formulation of the boundary conditions (the restrictions placed on a component's physical behaviour due to its connection to the other components) between adjacent components, needed to be determined. The simplest method would be to force the user to explicitly define the areas of contact between the adjacent components, and consequently apply the appropriate boundary conditions. However, in keeping with the objective of developing a system that essentially minimises human interaction, this method would be a direct violation of that objective.

Therefore it became apparent that a method, with the ability to automatically determine the areas of contact between adjacent components, needed to be incorporated into the integrated system. Further literature was reviewed accordingly, attempting to find a method that could effectively fulfil the following requirements:

(1) Determine, from the hierarchical system, the parent and child components of the specified component.

(2) Using the information obtained in (1), and knowing only the location and orientation of the relevant components, determine the areas of contact between the adjacent components.

(3) Using the information obtained in (2), segment the geometry of the specified component into areas of direct physical contact, and areas having no contact (in other words, the surrounding *free* areas).

(4) Apply the relevant boundary conditions to the areas of contact.

5.2 EXISTING METHODS

With the above goals in mind, the most obvious area or field of research to investigate would be a field concerned directly with contact surface recognition. Unfortunately, this initial search was unsuccessful in finding any useful or even relevant information. In an attempt to find all possible related information, the search was subsequently broadened to include the field of object recognition. Several papers in this field were reviewed, although for the most part the relevance of the information contained in these papers was rather limited. Belongie et al. [51] introduced a shape descriptor known as a *shape context* for shape-based object recognition. The method works by sampling a set of points from the contours of the object. Each point is subsequently associated with the shape context, which describes the coarse arrangement of the remainder of the shape with respect to that point. This paper introduced some interesting concepts, but the area of application was more suited to the recognition of geometrically similar objects. Aggarwal et al. [52] presented some of the more common methods of object recognition in the form of a comparative study orientated towards computer vision. Each of the methods had certain advantages and disadvantages, and the choice of a particular paradigm was ultimately dependant on the application at hand, and the amount and accuracy of the available information. This paper proposed an interesting overview of the use of knowledge-based approaches in object recognition systems. Nevertheless, the work presented by Aggarwal et al. [52] was orientated towards computer vision in "real-world" situations, rather than the recognition of specific contact areas of a mathematical representation of a physical component.

A further search, encompassing the field of collision detection, was then initiated. The first paper to be reviewed was concerned with collision detection in the apparel industry [53]. The authors stated that a fundamental problem in simulating the drape of material over solid bodies was that of collision detection. For example, classifying points, or elements, of the cloth as being interior to, being on, or being exterior to the surface of the body. Three alternative collision detection solution strategies were presented. The first method, known as *triangulation*, involved triangulating the front faces of the solid model and then calculating the outward surface normals to these triangular faces. The second approach was referred to as *surface interpolation*, and involved applying a specific interpolation function to each front face of the body. The resulting surface equations could then be used to classify the location of the foreign points. In terms of the integrated system considered here, a means of graphical representation using *face sets* was adopted (the method of face set representation will be described in the following chapter). The paradigm of contact surface detection employed for this integrated system adopted similar aspects from both the methods of triangulation and surface equations to calculate possible contact areas. The third approach used what is known as *topological mapping*. The idea behind this approach is to map each body sector (essentially a wedge-shaped volume) topologically onto a unit cylinder sector and then to classify each foreign point with respect to the transformed cylinder sector.

The last paper to be reviewed, also concerning methods collision detection, was orientated towards the simulation of motion in structures [54]. This paper highlighted an important idea that was suitably adapted for use in the integrated system. The author proposed a method known as a *global search* or a *pre-contact search* that essentially eliminates the bulk of the testing by putting the adjacent faces of two bodies into bounding boxes. These boxes are subsequently expanded to form capturing boxes which contain nodes of the faces that could possibly be in contact. The implementation of such a method increases the efficiency of a system by only testing faces / surfaces that are in close proximity, and therefore possibly excluding the bulk of unnecessary testing.

Although the literature reviewed in this chapter presented some interesting and possibly useful concepts, it was evident from the lack of fitting information that a contact surface methodology needed to be developed (see chapter seven).

CHAPTER 6

THE INTEGRATED SYSTEM (PART 1)

6.1 FACE SET REPRESENTATION

A *face set* is a means of graphical representation composed of individual *faces* [55]. A face is basically a three-sided plane shape or surface. It is therefore the simplest shape that can be used to represent a plane area. By using specific combinations of these triangular faces, or building blocks, even the most complex geometrical shapes can be closely approximated. (Any shape consisting only of straight edges can be exactly represented using face sets. However, a shape containing curved edges can only be approximated due to the fact that each individual face is represented by a shape bound by three straight edges. Therefore, the accuracy of the approximation can be greatly enhanced by increasing the number of faces used to represent the curved edge. Although using face sets results in approximated geometric representations in certain situations, this fact does not pose any immediate threat to using this method to represent geometry in finite element modelling. The reason for this being that all shapes in the finite element method are essentially "built up" from a finite number of elements (the simplest plane element is a triangular element bound by straight edges) as the name suggests.

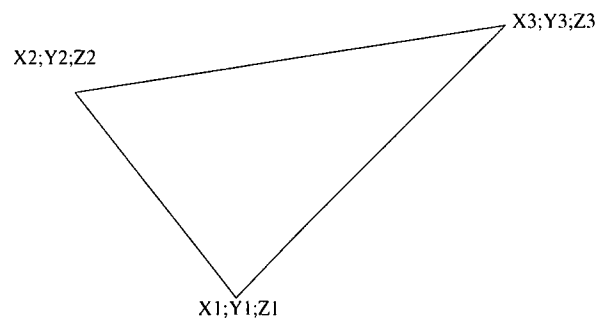


Figure 6.1.1. Graphical representation of a face set.

In Figure 6.1.1 above, an illustration of a typical face can be seen. The corresponding mathematical representation of this face, in the form of a fact in predicate logic, is as follows:

$s(X_1,Y_1,Z_1,X_2,Y_2,Z_2,X_3,Y_3,Z_3)$

Where: X_i, Y_i *and* Z_i – are the *x, y* and *z* coordinates respectively of each of the vertices (nodes) of the face set in a rectangular coordinate system.

It should be noted that the use of logic, as a means of shape representation, is not restricted to the method of face sets. Damski and Gero [56] define a logic formalism for representing shapes based on *halfplanes*. Chase [57, 58, 59] describe a method of shape representation using shape algebras. However, the method of face sets was justifiably selected for reasons that will be described shortly. The use of face sets for describing geometry can be most effectively demonstrated through a simple example. A dimensioned L-shaped bracket can be seen in Figure 6.1.2 (a). The corresponding graphical face set representation can be seen in Figure 6.1.2 (b). The mathematical representation of the L-shaped bracket is as follows:

s(0,0,0,1,0,0,0,0,-1)
s(1,0,0,1,0,-1,0,0,-1)
s(0,0,-1,1,0,-1,0,1,-1)
s(1,0,-1,1,1,-1,0,1,-1)

Using this format, the geometry of any shape can be easily stored as a compact *text-based* data file. The main reason for using face sets as a graphical representation tool is due to the fact that *Open Inventor*, the user interface responsible for displaying and manipulating the various components that may form part of a mechanical system, can efficiently convert the geometrical image into a data file containing the relevant face sets. However, the benefits were not only convenient data conversion. MSCN4W, the program responsible for performing the finite element calculations, is equipped with a scripting feature that allows the geometry and analysis specific information to be input directly into the solver via a *script*, which is essentially a small program. The script represents the geometry in terms of points and interconnected lines which can be easily obtained from the face set information. The method used to convert the face set information into the required boundary representation (ie. the shape containing only the necessary outside boundary edges / curves), and finally into a script format, will be discussed later in this chapter.

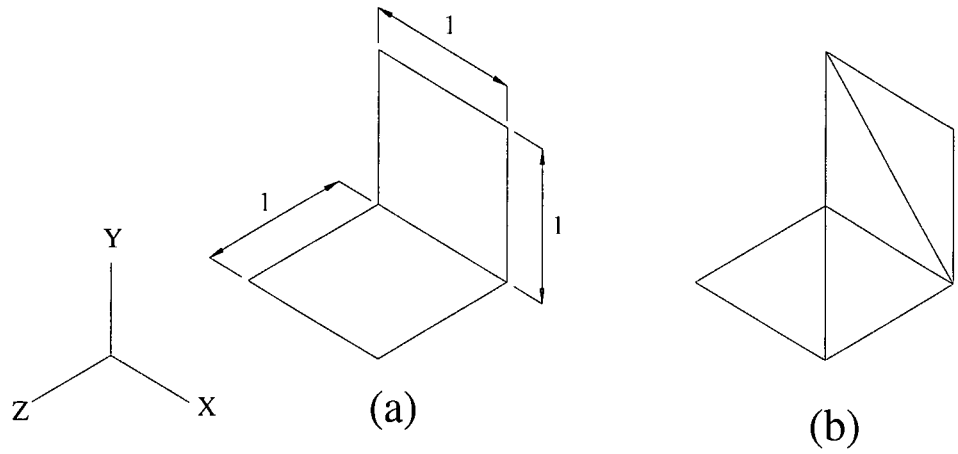


Figure 6.1.2. L-shaped bracket and face set representation.

6.2 THE SCRIPT FORMAT

MSCN4W is equipped with a powerful scripting feature which allows the entire model to be created and executed automatically via a script, which is essentially a simple ACSII text file. The script is written in a format which is a derivative of the *Visual Basic Scripting* language. All programs contain a main subroutine that acts as the entire program, or calls other functions and subroutines. The scripting language contains a library of built in functions which allow for the entire model to be created and analysed. The script is written in the following format:

Sub Main ()

(1) The variables to be used for the model creation and execution are declared as necessary.

(2) The points forming the vertices of the shape to be analysed are defined in terms of 'x', 'y', and 'z' rectangular coordinates.

(3) The interconnecting lines, each containing two endpoints, are defined.

(4) The boundary surfaces containing the relevant lines are defined.

(5) The material properties are defined.

(6) The element properties are defined. The research which has been performed to date allows for the recognition of solids and shells (hollow shapes) from the face set

information. Therefore the type of element utilised in the model is restricted to either a plate or solid element.

(7) The mesh on the surface or solid, depending on the model to be analysed, is then defined using the appropriate element type and a user defined element size. It was decided at the onset of the research project that the powerful automatic meshing facility provided with MSCN4W would be fully utilised as apposed to developing an automatic meshing facility, which could easily encompass an entire research project on its own.

(8) The load case and type of load (point load, distributed load along a curve, pressure load on surface or any combination of these loads) are defined.

(9) The boundary condition is then defined in terms of its location and constraint type (fixed, translation or rotation).

(10) Finally a function to analyse the script is defined.

End Sub

A simple example would best illustrate the format of the script, and some of the built in functions that are used to create and analyse a finite element model. For the sake of simplicity, a script will be created for a square plate having dimensions 1m x 1m, fixed boundary conditions on all curves, and a pressure load of 10 kN applied over the entire surface. The material has an elastic modulus of 200 GPa, and a Poisson's ratio of 0.3. The plate also has a thickness of 10mm and is meshed using 100 elements. The script will take the following form:

```
Sub Main ()
Dim p1 as esp_Coord
Dim p2 as esp_Coord
Dim p3 as esp_Coord
Dim p4 as esp_Coord
Dim c1 as long
Dim c2 as long
Dim c3 as long
Dim c4 as long
Dim listcurveID as long
Dim listcurve as long
```

```
Dim s1 as long
Dim mat as esp_Matl_Iso
Dim material as long
Dim prop as esp_Property
Dim property as long
Dim msize as long
Dim attrib as long
Dim size as double
Dim listsurfaceID as long
Dim listsurface as long
Dim mesh as long
Dim lset as long
Dim listnodeID as long
Dim listnode as long
Dim l as esp_Load_Value
Dim d as esp_Load_Dir
Dim load as long
Dim cset as long
Dim listcurvecID as long
Dim listcurvec as long
Dim constraint as long
Dim analyse as long

p4.x = 0: p4.y = 1: p4.z = 0
p3.x = 1: p3.y = 1: p3.z = 0
p2.x = 1: p2.y = 0: p2.z = 0
p1.x = 0: p1.y = 0: p1.z = 0
c1 = esp_LineEndpoints(colour,layer,p1,p2)
c2 = esp_LineEndpoints(colour,layer,p2,p3)
c3 = esp_LineEndpoints(colour,layer,p3,p4)
c4 = esp_LineEndpoints(colour,layer,p4,p1)
listcurveID = esp_ListNextAvailableID
listcurve = esp_ListAdd(listcurveID,c1)
listcurve = esp_ListAdd(listcurveID,c2)
```

```
listcurve = esp_ListAdd(listcurveID,c3)
listcurve = esp_ListAdd(listcurveID,c4)
s1 = esp_SurfBoundary(listcurveID)
mat.E = 200e9
mat.Nu = 0.3
material = esp_MatlCreateIsotropic(1,"Mat1",colour,layer,mat)
prop.val1 = 0.01
property = esp_PropCreate(1,"Prop",17,1,colour,layer,flags,prop)
msize = esp_MsizeDefault(0.1)
listsurfaceID = esp_ListNextAvailableID
listsurface = esp_ListAdd(listsurfaceID,s1)
attrib = esp_MAttrSurf(s1,property)
mesh = esp_MeshSurface(listsurfaceID,1)
lset = esp_LoadCreateSet(1,"Load")
listnodeID = ListSelectNextAvailableID
listnode = esp_ListSelectAll(listnodeID,Node)
l.z = 10000/121
load = esp_LoadNodal(1,listnodeID,0,l,0,d)
cset = esp_BCCreateSet(1,"Constraint")
listcurvecID = esp_ListNextAvailableID
listcurvec = esp_ListSelectAll(listcurvecID,curve)
constraint = esp_BCCurve(listcurvecID,1)
analyse = esp_FileNastranWrite(4,1)
End Sub
```

6.3 THE PRODUCT DATA MODEL

A database is a collection of information that's related to a particular subject or purpose, such as tracking customer orders or maintaining a music collection. For the purpose of this research project, *Microsoft Access* (or *Access*) was chosen as the database management system. *Access* is a relational database management system that stores information in tables (that is, rows and columns of information). A relational database

uses matching data from two tables to relate information in one table to information in the other table. Therefore, specific information is typically stored only once, which provides for an efficient database where the risk of entering incorrect data is minimized. Each table in the database includes a set of fields or columns, where a specific or unique type of information will be stored. The tables in *Access* databases can be efficiently modified allowing for easy expansion if required. This feature proved to be very useful as additional fields of information were added to the original table throughout the duration of the research project. An *Access* database was developed containing the necessary information that would allow a finite element analysis to be performed on any one of the components of the mechanical system contained in the database. The database incorporates all the information required to describe the model both dimensionally and functionally and communicates this information to whichever application requires it. As mentioned previously, the concept of a centralized product database, as described by Wu et al. [7], has been implemented for reasons of flexibility and ease of modification. Table 6.3.1 illustrates the database used to store the appropriate information.

ID	Name	Type	E (Gpa)	G (GPa)	Nu	Connected	Load	BC	V-Gauge	MRF	ExtLoad
1	Global	global				chd(10,0,0,0,0,0,0,fixed)					
2	Cylinder	faceset	210	75	0.27	chd(1,1.85,-5,-0.77,0,0,0,fixed) chd(3,1.85,-0.77,5,0,0,0,trans)	na	na	na	3	
3	Piston	faceset	72	35	0.33	chd(4,0,2,15.1,0,90,0,rot)	na	na	15,1.5,9.74	2	
4	Pin	faceset	200	72	0.28	na	na	na	na	3	
5	Bush	faceset	200	72	0.28	na	na	na	na	3	
6	Shell_1	faceset	200	72	0.28	na	na	na	na	3	
7	Shell_2	faceset	200	72	0.28	na	na	na	na	3	
8	Conrod	faceset	72	35	0.33	na	na	na	na	3	
9	end-block	faceset	72	35	0.33	na	na	na	na	3	
10	Crank	faceset	100	40	0.29	na	na	na	na	3	

Table 6.3.1. *Microsoft Access* database or product data model.

In the table above, each of the fields represent the following information:

(1) *ID* – this field defines the identification number of each of the components of the mechanical system.

(2) *Name* – defines the name of each particular component and the name of the data file containing the face set information for that component, unless global is specified. The name global allows for the connection of one or more of the components to the global coordinate system. This enables for a constant relative position to be maintained between two components in terms of the global coordinate system. For the mechanical assembly contained in this database, the relative locations of the crank and the cylinder would remain fixed.

(3) *Type* – defines whether the data contained in each row pertains to a component or the global coordinate system.

(4) *E (GPa)* - defines the elastic modulus of the material.

(5) *G (GPa)* - defines the shear modulus of the material.

(6) *Nu* – defines Poisson's ratio.

(7) *Connected* – this field defines the child components of the parent component. The components of the mechanical system are assembled in a hierarchical manner, as illustrated in Figure 6.3.1. A component may be connected to a parent, and an infinite number of children, or it could just be a stand-alone component. The information in this field takes the form of the following fact, defined by the functor *chd*, and containing eight arguments in the semantics of predicate logic:

chd(ID, X_{tc}, Y_{tc}, Z_{tc}, X_{rc}, Y_{rc}, Z_{rc}, Type)

where: *ID* – represents the ID number of the child component.

X_{tc}, Y_{tc} *and* Z_{tc} – represent the *x, y* and *z* translations respectively of the child component relative to the parent component.

X_{rc}, Y_{rc} *and* Z_{rc} - represent the *x, y* and *z* rotations respectively of the child component relative to the parent component.

Type – represents the constraint type existing between the parent and child. The options are *fixed; translation* or *rotation.*

The advantage of this field is that through the use of predicate logic, the contact surface areas between the parent and child components are automatically calculated and the specified constraint type applied. Therefore the user is not required to manually determine these contact areas, resulting in a more efficient process.

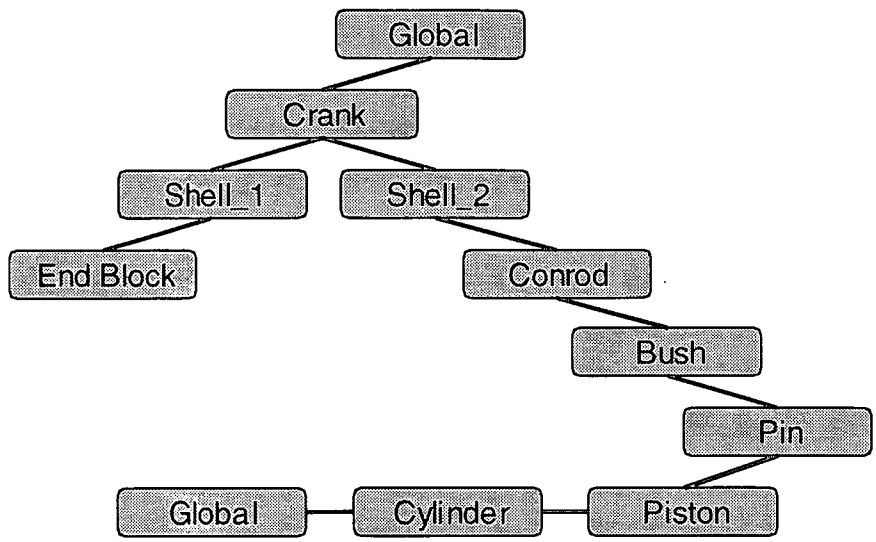


Figure 6.3.1. Hierarchical tree-structure.

(8) *Load* – defines a fact, named by the functor *load*, and containing seven arguments. The fact takes the following form:

load(Type, L_x, L_y, L_z, X, Y, Z)

Where: *Type* – defines a load on a point, curve or surface.
L_x, L_y *and* L_z – define the load magnitude along each of the three axes of a rectangular coordinate system.
X, Y and Z – define the location of the load.

(9) *BC* - defines a fact, named by functor *con*, and containing five arguments. This fact takes the following form:

con($Type_1$, $Type_2$, X, Y, Z)

Where: *Type*$_1$ - defines a constraint on a point, curve or surface.

Type$_2$ – defines the type of constraint to be applied. The options are fixed; translation or rotation.

X, Y and Z – defines the location where the constraint will be applied.

(10) *V-Gauge* – defines the position of a *virtual stress* and *displacement gauge* on the component to be analysed.

(11) *MRF* – defines the mesh refinement factor. The higher the number the greater the mesh density.

(12) *ExtLoad* – if a motion analysis has been performed prior to the finite element analysis, and the reaction loads from this analysis are available, these loads can be input directly into the FE model.

Access supports *SQL (Structured Query Language)*, which allows for proficient communication with external applications. *SQL* is a widespread standard for accessing relational databases. It involves terms like: tables, rows, columns and indexes. The interaction between the *Prolog* application and the *Access* database is possible after a connection has been established. This connection is identified by a connection handle (*DBC_HANDLE*). The interaction is divided into a number of statements. Each statement is identified by a statement handle (*STMT_HANDLE*). Each statement handle can contain an entire field of information.

CHAPTER 7

THE INTEGRATED SYSTEM (PART 2)

7.1 INTRODUCTION

The primary objective of the program is to automatically construct a FE model of the desired component from the database and then analyse this model obtaining the necessary results (stress and displacements at predefined locations), which can then be made available to the engineer or designer. The functionality of the program can be divided into five main sections, or subprograms, each allocated with different responsibilities which, when combined, allow the entire analysis to be performed with minimal user intervention:

(1) Data retrieval from the *Access* database.
(2) Geometrical representation of the primary component.
(3) Contact surface definition
(4) The script generation
(5) Output data retrieval

The program structure is broken down into the five sections mentioned above, and therefore the corresponding rules in each of the sections are numbered separately. In other words, the first section contains rules 1-16, the second contains rules 1-39, etc. In the description of the program contained in the following sections, when reference is made to a particular rule, the following notation will be used to identify the rule in the program code contained in the Appendix:

$$(\text{rule}_i, \text{rule}_{i+1}, \ldots, \text{p X}) \text{ or } (\text{rule}_i - \text{rule}_{i+n} \text{ p X})$$

Where: $rule_i$ – represents the number of the rule in each section contained in the Appendix.
X – represents the page number.

7.2 DATA RETRIEVAL

Through the use of structured query language (SQL), a connection is established with the *Access* database. The connection allows for communication to occur between the server, in this case the program (in *Prolog*), and the client, in this case the database. The interaction between the client and the server is divided into a number of statements. Each statement is represented by a statement handle (STMT_HANDLE) and contains a single row of information from the database. This row of information is further subdivided into the individual fields contained in the database. Due to difficulties experienced representing and manipulating numerical values stored as real numbers, all data, including numerical information, are originally stored as string variables. When calculations are required to be performed, the appropriate numerical fields are converted from string to real variables using standard *Prolog* predicates.

Once communication has been established, the information is stored in a *temporary external database* in a logical format that allows for efficient retrieval, when the information is required. The purpose of the external database (or *facts section* as defined by *Prolog*) is that information in the form of clauses or facts can be added (or subtracted) from the database during the program execution. Therefore, the "*knowledge*" contained in the database is not required to have been predefined and can be explicitly expanded to represent any given model. It must be noted that the external database is temporarily contained within the *Prolog* program, and has no connection with the *Access* database. Subsequent to the program execution, all the information contained in the external database is deleted to prevent unnecessary duplication. Two different types of object are contained within the *Access* database. The first is an object defined by the type *global*. This object merely represents the connection of a component to the global coordinate system. This object, is stored as a fact defined by the functor *gcs* and has two arguments:

gcs(ID, Name)

Where: *ID* – is the ID number for that particular row of information.
Name – is the name of the global coordinate system.

The second type of object is defined by the type *faceset*. This object represents a component contained within the *Access* database and is stored as a fact containing the material properties, information regarding the density of the element mesh and indirectly, the geometry description of the component. The fact is defined by the functor *entity*, and has seven arguments:

entity(ID, Name, faceset(Name), E, G, Nu, Mrf)

Where: *ID* – is the ID number for that particular row of information.
Name – is the name of the component.
faceset(Name) – is a "*compound data object*" representing the type of geometry description. *Compound data objects* allow several pieces of information to be treated as a single item in such a way that they can be easily dissembled when the individual pieces are required separately.
E – represents the modulus of elasticity of the material.
G – represents the shear modulus of the material.
Nu – represents Poisson's ratio for the material.
Mrf – is the mesh refinement factor. The greater the number, the higher the mesh density.

The information regarding the child components, the load and constraints has already been described in section 6.3, and will therefore not be described again. These facts are identical except for the extra *ID* argument in the *load1* and *con1* clauses. These three facts are each originally represented as string variables and not as separate facts. In order to convert them from strings to facts, the following method is used. A text file with a "*.file*" extension is created. A Prolog standard predicate is then used to write the string into the text file. The text file is then opened, and using the standard predicate "*consult*", each individual line in the text file is read and consequently saved in the external database as separate facts. The text file is then deleted. This conversion allows for the information to be more efficiently accessed when it is required. The three facts can be seen below:

chd(ID, X, Y, Z, Rx, Ry, Rz, Type)
load1(ID, Type, Lx, Ly, Lz, X, Y, Z)

con1(ID, $Type_1$, $Type_2$, X, Y, Z)

The next fact to be stored in the external database contains information about the components (both child and parent), which are in direct contact with the primary component (the component to be analysed). Clauses in the form of rules (5,6 p A-5) are used to determine which child components are connected to the primary component. A list, containing the child components (the list can be empty if the primary component has no child components), is formed. Further rules (12-15 p A-5) are then used to determine the parent components of the primary component. The parent components are then added to the original list. The connected fact is defined by the functor *connected*, and has two arguments:

connected(Name, Elist)

Where: *Name* – the name of the primary component.
Elist – is a list containing the components connected to the primary component. The elements of *Elist* take a similar form to the fact defined by the functor *chd*, except that in this case the facts are defined by the functor *e*. If there are no connections, this list will be empty.

In order to obtain analysis information, results such as stresses and displacements, the designer needs to define a location on the component where a *virtual gauge* can be placed. This information is stored as a fact defined by the functor *vg*, and has two arguments:

vg(Name, Coord)

Where: *Name* – is the name of the component on which the *virtual gauge* will be placed.
Coord – represents the *x, y* and *z* coordinates, in a rectangular coordinate system, defining the location of the *virtual gauge*.

Lastly, information regarding the geometrical description of the component is required to be represented in the form of facts. This information is contained in a text file, in the

form of face sets. A description of face sets can be seen in section 6.1. Using the standard predicate *consult*, the information contained in the file is converted into a set of facts, defined by the functor *s*, and having nine arguments:

$$s(X_1, Y_1, Z_1, X_2, Y_2, Z_2, X_3, Y_3, Z_3)$$

Where: X_i, Y_i *and* Z_i - represent the *x*, *y* and *z* coordinates respectively of node i of the face, in a rectangular coordinate system.

At this stage, all the information from the *Access* database has been retrieved and converted into the necessary facts that will allow a FE model to be created and analysed.

7.3 GEOMETRICAL REPRESENTATION

The purpose of this section is to modify the face set data representation into a format where only the boundary curves, of a surface defined on a plane, are specified. It would be a simple process to merely construct the FE model using all the information represented by the face sets. The constructed model would be an exact replica of the face set information, a model composed of a finite number of triangular surfaces defining the original shape. This method of representation would be acceptable, however the final shape would contain redundant curves (curves which do not form the boundary of a surface on a particular plane). The main problem associated with this form of surface representation concerns the meshing (dividing the individual surfaces into a number of finite elements) of these surfaces. Although there are no problems in applying a mesh to a triangular surface, the probability of the mesh containing poorly shaped elements (two examples of which are elements with large aspect ratios or elements which are highly skewed) is significantly higher. Element shapes which are compact and regular usually give the best results [30]. For example, the ideal triangle is equilateral, and the ideal quadrilateral is square, and so on. Therefore to reduce the possibility of the mesh containing poorly shaped elements, wherever possible, a surface on a given plane is defined by the minimum number of curves that would allow the original boundary surface to be exactly reproduced.

Figure 7.3.1 represents this concept. In Figure 7.3.1, the l-shaped bracket is defined by four face sets and nine curves, labelled *c1* to *c9*. The boundary surface can be represented using all nine curves, however, curves *c2* and *c6* are redundant, and therefore their absence does not effect the original boundary surface, which can be efficiently defined by the remaining seven curves.

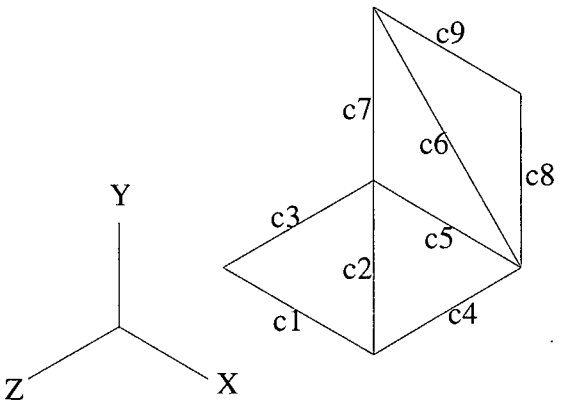


Figure 7.3.1. L-shaped bracket defined by four face sets.

The conversion from the original face set information to a list containing only boundary curves can be divided into the following five sections:

(1) Face set data conversion
(2) Face set plane definition
(3) Contiguous faces on the same plane
(4) Grouping of boundary curves
(5) Surface or solid representation

7.3.1 FACE SET DATA CONVERSION

The face set information, described in the previous section, is initially modified into two new facts that allow for a more convenient representation of the data. The facts, defined by the functors *s1* and *s2*, can be seen below:

s1(Name, $p(X_1, Y_1, Z_1)$, $p(X_2, Y_2, Z_2)$, $p(X_3, Y_3, Z_3)$)
s2(Name, $[C_1, C_2, C_3]$)

The first fact defines the face in terms of its name, and its nodes or vertices. The name argument allows for the program to easily recognise the different faces. The three nodes are each represented by *compound data objects* defining their positions in space. Rules (2,3 p A-5) are used to convert the original face into the new face defined by the functor *s1*. The second fact defines the face in terms of its name, and a list containing three curves joining the three vertices of that face. It must be noted at this point that the functionality of the program relies on the fact that no two faces will occupy the same location in space. Also, no two curves or points may be coincident. Therefore, when the program defines a new curve, it must recognise which curves have already been created and consequently not redefine the same curve. The following logic is used when formulating the facts defined by the functor *s2*:

(1) For the first face, define a new fact recognised by *Name*.
(2) Construct a curve from the 1st point to the 2nd point, from the 2nd point to the 3rd point, and from the 1st point to the 3rd point.
(3) Save the new fact in the database.
(4) For the remainder of the faces, define a new fact recognised by a *NewName*.
(5) Determine if a curve exists from the 1st point to the 2nd point, if the curve exists, use this previously defined curve. Otherwise, if the curve does not exist, construct a curve from the 1st point to the 2nd point.
(6) Repeat (5) for the last two curves.
(7) Save the new fact in the external database.

The corresponding rules (4-8 p A-6) in the language of *Prolog* can be seen in the Appendix. Using this logical structure, the possibility of defining a new face set containing coincident curves is eliminated.

7.3.2 FACE SET PLANE DEFINITION

The purpose of this section is to develop an equation describing the plane of each individual face, allowing for the faces on common planes to be grouped together, thereby forming a boundary surface composed of a number of faces. To define the

equation of a plane, it is first necessary to determine the equation of a vector normal to that plane.

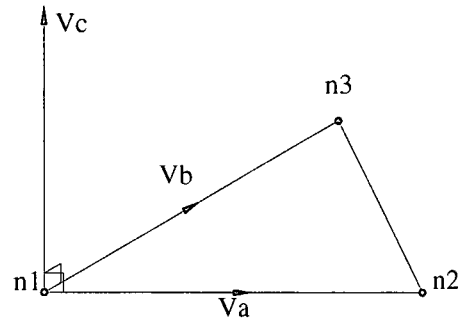


Figure 7.3.2.1 The normal of two vectors of a face set.

Fortunately, this can be done simply by knowing the locations of three points lying on the plane. This information is of course available in the form of facts describing the locations of the three nodes of a face. Consider Figure 7.3.2.1. In this figure, *n1*, *n2* and *n3* represent the three nodes of a given face. *Va* is a vector parallel to the curve constructed between nodes *n1* and *n2*. Likewise, *Vb* is a vector parallel to the curve constructed between nodes *n1* and *n3*. *Vc* is a vector perpendicular to both *Va* and *Vb*, and consequently normal to the plane of the face defined by nodes *n1*, *n2* and *n3*. Using a method known as the *cross product*, the equation of the unknown vector *Vc* can be calculated from the equations of the known vectors *Va* and *Vb* [60]. Assume that *Va* and *Vb* are defined by the following vector equations:

$$\mathbf{Va} = a_1\mathbf{i} + a_2\mathbf{j} + a_3\mathbf{k}$$
$$\mathbf{Vb} = b_1\mathbf{i} + b_2\mathbf{j} + b_3\mathbf{k}$$

Then the cross product of *Va* and *Vb*, denoted *Va×Vb*, can be calculated as follows:

$$\mathrm{Va}\times\mathrm{Vb} = (a_2b_3 - a_3b_2)\mathbf{i} - (a_1b_3 - a_3b_1)\mathbf{j} + (a_1b_2 - a_2b_1)\mathbf{k}$$
$$= \mathrm{A}\mathbf{i} - \mathrm{B}\mathbf{j} + \mathrm{C}\mathbf{k}$$

An equation of the plane passing through (x_o, y_o, z_o) and perpendicular to the nonzero vector $(\boldsymbol{Ai} + \boldsymbol{Bj} + \boldsymbol{Ck})$ defined above, is given by:

$$A(x - x_o) + B(y - y_o) + C(z - z_o) = 0$$

Where: x_o, y_o *and* z_o – represent the coordinates of a point on a plane. This point could be *n1, n2* or *n3* from Figure 7.2.2.1.

Consider the following example. Assume that a face, containing nodes *a*, *b* and *c*, is defined by the following fact:

s1(Name, p(7, 0, 2), p(3, 2, -1), p(0, 5, 2))

Where: a = p(7, 0, 2), b = p(3, 2, -1), c = p(0, 5, 2)

Using the cross product, find the normal to the plane, defined by the face "*abc*", at node *a*. Then determine the equation of this plane. Firstly, two vectors V_{ab} and V_{ac}, having their origin at node *a* and extending to nodes *b* and *c*, are constructed. The vector notation is displayed after the double arrow.

$$V_{ab} = \langle 7\text{-}3, 0\text{-}2, 2\text{-}(-1)\rangle = \langle 4, -2, 3\rangle \quad \Rightarrow \quad 4\mathbf{i} - 2\mathbf{j} + 3\mathbf{k}$$
$$V_{ac} = \langle 7\text{-}0, 0\text{-}5, 2\text{-}2\rangle = \langle 7, -5, 0\rangle \quad \Rightarrow \quad 7\mathbf{i} - 5\mathbf{j}$$

Next, a vector V_n, perpendicular to both V_{ab} and V_{ac} is constructed.

$$V_n = V_{ab} \times V_{ac} = (-2\times0 - 3\times(-5))\mathbf{i} - (4\times0 - 3\times7)\mathbf{j} - (4\times(-5) - (-2)\times7)\mathbf{k}$$
$$= 15\mathbf{i} + 21\mathbf{j} - 6\mathbf{k}$$

Therefore, the equation of the plane normal to V_n and passing through point *(7, 0, 2)* is:

$$15(x - 7) + 21(y - 0) - 6(z - 2) = 0$$
$$\underline{15x + 21y - 6z = 93}$$

Using this same method, an equation describing the plane of each face is developed and stored as a fact, defined by the functor *plane*, having five arguments:

plane(Name, A, B, C, D)

Where: *Name* – represents the name of the face.

A, B and C – are the coefficients of *x, y* and *z*, in the plane equation, respectively.

D – is the summation of the coefficients multiplied by the coordinates of the origin point.

Information describing the plane of each equation is now available to the program, subsequently allowing the orientation of each face to be directly compared and analysed. Rules (9-12 p A-6) were used to develop the equations defining the planes of the faces. The rules are relatively simple and therefore deserve only a brief explanation. Rule (9 p A-6) is responsible for calculating the coefficients. Rule (10 p A-6) investigates the value of the variable defined by *D* in the *plane* fact. The reason for performing this check, is that this value becomes the denominator in a division calculation, and a zero value is therefore not permitted. Depending on the value contained in *D*, either rule (11 p A-6) or rule (12 p A-6) is used to save the fact in the correct format.

It should be noted at this stage that *Prolog* is a programming language primarily for symbolic (non-numeric) computations. When representing real numbers, *Prolog* allocates the variable with only a certain amount of memory, resulting in the occurrence of *rounding off* errors. Consider the following clause:

Assume X belongs to the set of real numbers,

$X = 7 / 3,$

$X \times 3 = 7.$

The above goal would frequently fail (the exact outcome depends on the accuracy of the floating point calculations in use on a particular platform). However, this *rounding off* error or inaccuracy can be overcome by allowing the value to lie within a narrow range. This range has be appropriately termed an *accuracy factor* and is demonstrated below:

Assume X belongs to the set of real numbers,

$$X = 7 / 3,$$
$$X \times 3 < 7 + 0.0001,$$
$$X \times 3 > 7 - 0.0001.$$

The above goal would always prove to be true due to the addition of the *accuracy factor*. The second *equality* operator of the previous goal has been replaced with the "*less than*" and "*greater than*" relational operators. Therefore the *rounded off* value instantiated to *X* is allowed to lie within a specified range. This method of dealing with numerical inaccuracies was used extensively throughout the program.

7.3.3 CONTIGUOUS FACE SETS ON THE SAME PLANE

Once the plane equations have been developed for all the faces, the next step is to identify and group all faces on common planes. Once the faces have been grouped accordingly, these groups are then further examined for individual face contiguity. This concept is illustrated in Figure 7.3.3.1. Initially, recognising only two separate planes, the faces will be grouped into the following two lists or sets:

Set_1 = [s1, s2], Set_2 = [s3, s4, s5, s6, s7, s8]

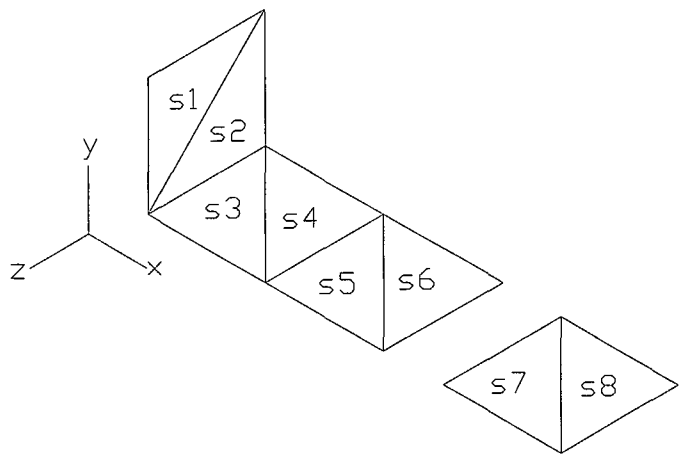


Figure 7.3.3.1 Face set contiguity and common planes.

However, upon further examination, the program recognises that s7 and s8 are not contiguous with the remainder of the set, and therefore a new set is created, resulting in the following sets:

$Set_1 = [s1, s2]$, $Set_2 = [s3, s4, s5, s6]$, $Set_3 = [s7, s8]$

When determining which faces lie on the same plane, the program compares the coefficients (including the variable represented by *D*) of the *plane* facts. Two faces are assumed to be co-planar if any of the following conditions are satisfied:

(1) The planar equations are identical.
(2) The planar equations are exact opposites, for example, *(4x + 9y –3z = 15)* and *(-4x –9y +3z = -15)*.
(3) One equation is an exact multiple of the other.

The logic structure responsible for examining the first two conditions is relatively simple, however, that of the third condition is considerably more complex due to the nature of the planar equations. The difficulty lies in the fact that the coefficients (and the *D* variable) in the *plane* fact could be instantiated with any value, including zero. The logic structure can be described using the following methodology, and the corresponding rules (13-18 p A-6) in the semantics of *Prolog,* can be seen in the Appendix:

(1) For the first *plane* fact contained in the external database, select the first *plane* fact defined by *Name*, and make a new fact called *facelist* containing a list with this *Name*.
(2) Place a marker on the *plane* fact defined by *Name* so that it cannot be used again, and proceed to the (3) with this *facelist* fact.
(3) For the remainder of the *plane* facts, retrieve the values from the *plane* fact defined by *Name* contained in *facelist*.
(4) Select a new *plane* fact defined by *NewName* and retrieve the values. Compare the values from *Name* and *NewName*, and if all the individual values are equal, add *NewName* to the *facelist* list and go back to (3). If all the individual values are opposite, add *NewName* to the *facelist* list and go back to (3). If the values are neither equal or opposite, proceed to (7).
(5) If (7) is true, add *NewName* to the *facelist* list and go back to (3). Otherwise make a new *facelist* fact containing new *plane* fact *NextName* and go back to (3).
(6) If all *plane* facts have been used, then end cycle.

The primary function of the next two rules is to compare the values contained in the *plane* facts, searching for the occurrence of multiples. It must be noted that all the coefficients (and the D variable) must be exact multiples (the multiple for one pair of corresponding values must be the same as the multiples for the rest of the pairs of values) for this rule to be *true*. A ratio, between the corresponding values of two *plane* facts, is established, allowing for the possibility of a common multiplier between the remainder of the coefficients to be investigated. Of course, the instantiation of zero to one or more of the variables in one *plane* fact, relies on the corresponding values of the other *plane* fact to assume the same zero value, if the conclusion of the rule is to be *true*. The logic structure is described by the following statements:

(7) For all values in both sets of *plane* facts, compare the corresponding values from both sets. If both values are zero, then continue. Otherwise check the value of each variable, and if the first or second value is zero, then fail. Otherwise calculate and save the ratio of the two values and proceed to (8).
(8) Establish whether a common multiplier exists for all non-zero pairs. If common multiplier exists, then the conclusion is true. Otherwise the conclusion is false.

At this stage the external database contains information about which particular face sets belong to a certain plane. This information is contained in a fact defined by the functor *facelist*, and has two arguments:

facelist(Name, List)

Where: *Name* – is the name of the *facelist* containing all face sets on a particular plane.
List – is a list containing the names of the face sets.

The next step, as was mentioned previously, is to examine the lists contained in the *facelist* facts, and to determine if all the face sets in each list are contiguous. The following method is utilised:

(1) For each occurrence of the fact *facelist*, duplicate the two arguments of this fact *Name* and *List* into a temporary fact defined by the functor *templist*.

(2) Subtract the first face set from the list of face sets contained in the first *templist* fact. Create a new fact, defined by the functor *curvelist1*, having two arguments, *Name* and an empty list defined by *List*. Place the subtracted face set in the empty list.

(3) Create a fact defined by the functor *lastlist*, having an empty list as its only argument. The purpose of *lastlist,* is to allow the last set of adjacent face sets added to *curvelist1* to be checked in turn for the face sets adjacent to them.

(4) Determine which face sets are adjacent (and in direct contact) to the face set in step (2). Add the adjacent face sets to the list contained in the fact *curvelist1*. Replace the last assertion of the fact *lastlist*, with a new fact, also defined by *lastlist*, containing a list of the adjacent face sets.

(5) Determine which face sets are adjacent to each of the face sets contained in *lastlist.* Delete these face sets from *templist*, and add them to *curvelist1*.

(6) Delete the last assertion of the fact *lastlist*, and replace it with a new fact containing a list of the new adjacent face sets from step (5).

(7) Repeat this process until all the *templist* facts contain empty lists.

The external database has now been expanded with a fact containing a list of contiguous face sets belonging to a common plane:

curvelist1(Name, List)

Rules (19-28 p A-7) show the implementation of the above method in developing the *curvelist1* fact. The following section describes how the boundary curves are extracted from these contiguous face sets.

7.3.4 GROUPING OF BOUNDARY CURVES

The penultimate section concerned with the geometrical representation of components deals with selecting the appropriate boundary curves from the set of curves created from the original faces. A curve will be selected as a boundary curve, if that particular curve is not shared between two faces on the same plane (faces contained in *curvelist1*). It is important to note that a boundary curve can be shared between two faces which do not lie on the same plane, for example, c5 in Figure 7.3.1. The logic structure for this

section of the program is uncomplicated, and essentially involves selecting a particular curve from a certain face set, and then searching for the existence of this same curve in another face contained in the list of *curvelist1*. The logical representation of this section can be described as follows:

(1) From the list in *curvelist1* select a face. From this face select the first *curve*, then the second curve, and then the third curve proceeding to (2) after each selection. Repeat this process until the list in *curvelist1* is empty.
(2) Using the *curve* obtained from (1), select a face that is different from the face selected in(1).
(3) Retrieve the list of curves associated with this face selected in (2). Determine if *curve* is a member of this list of curves. If *curve* is not a member, then go back to (2). Otherwise end the recursive cycle.
(4) If there are no more faces to select, then go to (1) with a list from a different *curvelist1* fact.

In the actual program, three rules (one for each curve of the facet) are used to perform the same function as the last three steps in the above explanation. However, the logic structure of each of these rules is identical. Rules (29-36 p A-8) are responsible for the above mentioned logic. At this stage, the external database contains the final list of boundary curves representing the relevant component. The fact containing the list of boundary curves is defined by the functor *curvelist*, and has two arguments:

curvelist(Name, List)

Where: *Name* – represents the name of the list of boundary curves on a plane.
List – is the list of boundary curves.

A complete geometrical representation of the component is now available to the program. However, at this stage, no information regarding the *type* of model to be created has been defined. The next section describes how the boundary curves defining the geometry of the component are analysed in order to determine the *type* of model to be created.

7.3.5 SURFACE OR SOLID REPRESENTATION

The program is able to represent two types of geometry, namely surfaces and solids. A viable addition to the program would be the ability to represent structures such as frames or trusses using only curves. However due to time constraints, this addition could not be realized. The program has been logically equipped with a feature that allows for the differentiation between components that can only be represented by surfaces, and components that can be constructed using either surfaces or solids. Figure 7.3.5.1 depicts a representation of three components. The first figure, an open box, is represented by five surfaces forming an "*open boundary*". This component can only be represented by a set of surfaces. The next two figures, a closed box and a solid cube, are both represented by six surfaces forming "*closed boundaries*". In the case of *closed boundaries*, the component can be constructed as either a solid or a surface, depending on the information specified by the user. To distinguish between *open* and *closed* boundaries, the program utilises the following strategy.

If all the curves from one boundary surface (curves contained in *curvelist*) are common to another boundary surface, then the component forms a *closed* boundary. If not, the component must be defined as an *open* boundary. Rules (37-39 p A-8) are used to perform the *closed* boundary checking.

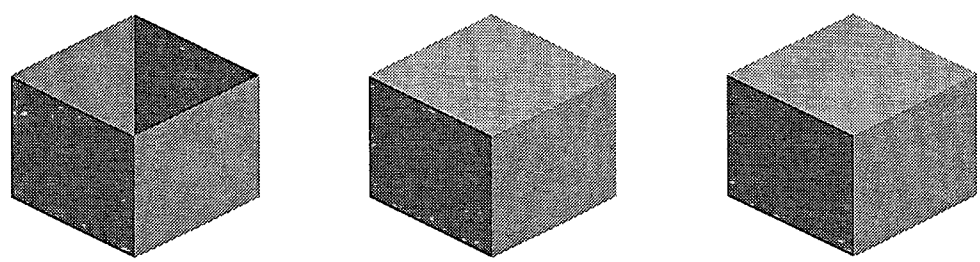


Figure 7.3.5.1. (a) Open box, (b) Closed box, (c) Solid cube.

This section completes the initial geometrical representation of the primary component. At this stage, sufficient information has been placed in the external database to allow for

the creation of a solid or surface model. The following sections describe how the created model is manipulated to allow for the definition and placement of the boundary conditions.

7.4 CONTACT SURFACE DEFINITION

As stated at the end of the previous section, all the necessary information required to create the geometry of the component has been temporarily stored in the external database. The next step in the creation of the FE model, assuming that the model's geometry has been subdivided into a finite number of elements, is to apply the appropriate loads and boundary conditions to that model. This can essentially be performed in one of two ways (or a combination of both), depending on the information contained in the *Access* database, specified by the designer.

(1) The designer can explicitly provide the information concerning the type, and placement location of the loads and boundary conditions. This method is particularly useful when analysing individual components (components which are not part of a mechanical assembly).
(2) Using the hierarchical structure that describes the primary component's parent and child relations, the program automatically calculates the contact surface areas existing between adjacent components. This method increases the efficiency of generating a FE model when analysing components of a mechanical assembly.

This section focuses on the complex task of recognising the contact surface areas between adjacent components. If a designer is manually constructing a FE model of a component whose behaviour or movement is physically constrained by other components, then he / she needs to carefully predetermine the subdivisions of the original boundary surfaces that will accurately represent the boundary conditions imposed on that component. Naturally, if the component contains complex geometry, then this laborious task of subdividing the boundary surfaces, can demand a significant amount of the designers time, thereby directly escalating the costs of the design process. Consider the two components (simple plates each containing three holes) in Figure 7.4.1.

The primary component is the component to be analysed, and therefore its original geometry, as shown in Figure 7.4.1 (a), is required to be subdivided into the corresponding areas of contact and into the remaining free surfaces, as shown in Figure 7.4.1 (d). The secondary component represents the boundary condition imposed by an adjacent component in contact with the primary component. This component is not required to be modelled or analysed, its geometrical representation is only necessary to allow for the subdivision of the primary component. The objective of this section of the program structure is to recognise which surfaces of two adjacent components are in contact, and then efficiently modify the original geometry as required (for example, from (a) to (d) in Figure 7.4.1). It is evident, even from this relatively simple example, that this *subdivision* process, when performed *manually* by the designer, could be a time consuming process, and that any automation in this area, will greatly enhance the efficiency of the design process.

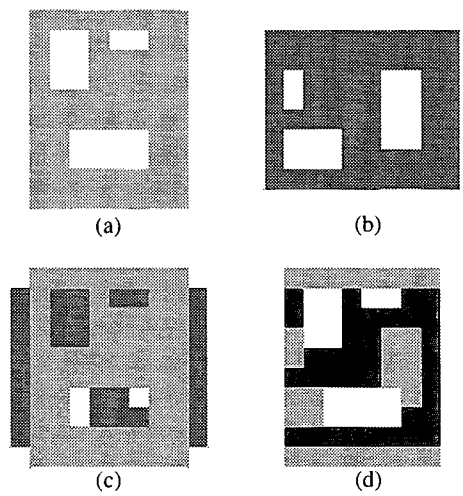


Figure 7.4.1. Representation of the contact areas between adjacent components. (a) Primary component. (b) Secondary Component. (c) The two components in contact. (d) The final contact surface areas (represented by black) and free surfaces (represented by light grey).

As stated before, this is the most complex section of the program structure and is therefore divided into a number of subsections. Consequently, rather than disrupting the flow of the explanation of the program structure, two different strategies, which are used extensively throughout this section, are discussed prior to the start of the discussion on the program structure. This will hopefully allow for a clearer, more

logical explanation of this section. The first, and simpler of the two paradigms, is called the "*Area Method*" and will be discussed in the following section. The second paradigm, has been termed the "*Method of Intersections*", and will be described subsequent to the *area method.*

7.4.1 THE AREA METHOD

The purpose of the *area method* is to determine if a curve, or set of curves, is contained within a boundary surface lying on the same plane. Consequently, a curve, or a set of curves defining a boundary surface, can be said to be in direct contact with the other boundary surface. A boundary surface is a surface defined by a closed set of boundary curves, and possibly containing inner closed boundaries (for example, Figure 7.4.1 (a)). It must be remembered that at the stage when the area method is used, all intersecting curves have been identified and replaced with curve segments connected at the original points of intersection. In other words, no two curves intersect at any location other than the end-points of those curves.

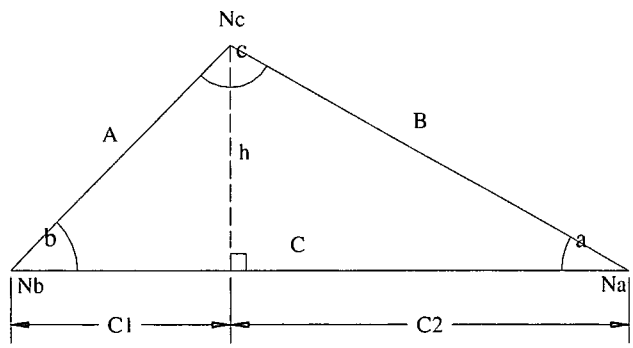


Figure 7.4.1.1. Labelled face set.

Before the *area method* can be explained, an algorithm for calculating the area of a given face set in three-dimensional space needs to be defined. Firstly, consider Figure 7.4.1.1. The program is required to calculate the area of a given face set knowing only the coordinates of the three nodes (*Na*, *Nb* and *Nc*). Therefore, the lengths of curves *A*, *B* and *C* are initially determined using the following equation:

$$L = ((X_{Ni}-X_{Nj})^2 + (Y_{Ni}-Y_{Nj})^2 + (Z_{Ni}-Z_{Nj})^2)^{1/2} \qquad (7.4.1.1)$$

Where: X_{Ni} *and* X_{Nj} – represents the x coordinate of the i^{th} and j^{th} node respectively.

Y_{Ni} *and* Y_{Nj} – represents the y coordinate of the i^{th} and j^{th} node respectively.

Z_{Ni} *and* Z_{Nj} – represents the z coordinate of the i^{th} and j^{th} node respectively.

From Figure 7.4.1.1:

$$C1 = (A^2-h^2)^{1/2} \quad \text{or} \quad h = (A^2-C1^2)^{1/2} \tag{7.4.1.2}$$

$$C2 = (B^2-h^2)^{1/2} \quad \text{or} \quad h = (B^2-C2^2)^{1/2} \tag{7.4.1.3}$$

$$C = C1 + C2 \tag{7.4.1.4}$$

Substitute (7.4.1.3) and (7.4.1.4) into (7.4.1.2),

$$(A^2-C1^2)^{1/2} = (B^2-C2^2)^{1/2}$$

$$A^2-C1^2 = B^2-(C-C1)^2$$

$$A^2-C1^2 = B^2-C^2+2CC1-C1^2$$

$$C1 = \frac{(A2^2 - B2^2 - C2^2)}{2C} \tag{7.4.1.5}$$

From which *h* can be calculated by substitution into either (7.4.1.2) or (7.4.1.3), and finally the area of any face can be determined using the following equation:

$$\text{Area} = \frac{C \times \sqrt{\left(A^2 - \left(\frac{(A2^2 - B2^2 - C2^2)}{2C}\right)^2\right)}}{2} \tag{7.4.1.6}$$

Figure 7.4.1.2 depicts a rectangular plate and hole defined by eight faces. The function of this diagram is to illustrate how a curve segment's relation to a boundary surface can be examined. Stated differently, does a curve segment lie within a closed boundary? For example, using the *area method*, the program is required to determine whether curves *L1* and *L2* are in contact with the boundary surface defined in Figure 7.4.1.2 (a). The approach utilised by the program can be described by the following method:

(1) Determine the midpoint (*mp*) of the specified curve.

(2) Construct a curve from each of the nodes of the face to the midpoint.

(3) Calculate the area of the face set A_{tot}.

(4) Calculate the area of each of the three new triangles (*A1*, *A2* and *A3*).

(5) If $A_{tot} = A1 + A2 + A3$, for any one of the faces defining the boundary surface, then the curve lies within the boundary surface.

(6) All faces of the boundary surface must be checked until a curve is determined to lie within a particular face, or until no more faces exist.

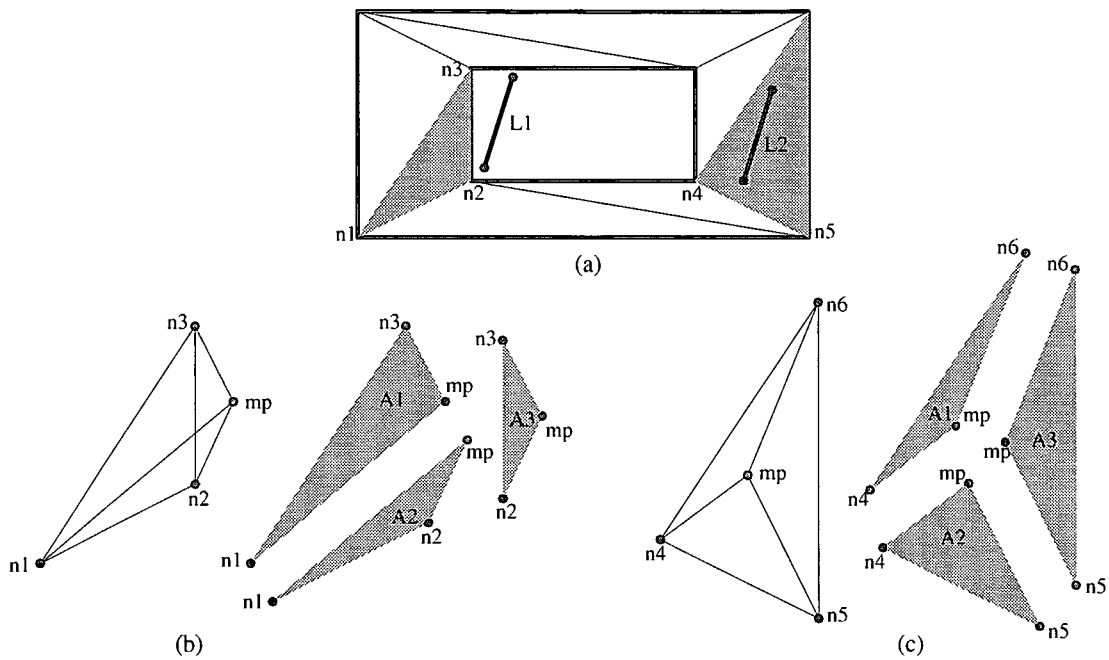


Figure 7.4.1.2. (a) Rectangular plate and two line segments, L1 with no contact, and L2, in contact with the boundary surface. (b) Face set connected to midpoint of L1. (c) Face set connected to midpoint of L2.

The corresponding rules (201-205 p A-21) in the language of *Prolog* can be seen in the Appendix. The outcome of this example is evident from the illustration. Curve *L1* clearly lies outside the boundary surface defined by the faces. When the *area method* is used to determine this, as shown for the highlighted face in Figure 7.4.1.2 (b), the summation of the individual areas will never equal the total area of any one of the faces. However, when checking curve *L2*, for the highlighted face shown in Figure 7.4.1.2 (c), the summation of the individual areas will be exactly equal to the area of that face. Therefore, *L2* is contained within the boundary surface, as shown in the figure.

7.4.2 METHOD OF INTERSECTIONS

The *method of intersections* is used on several occasions throughout the program to perform two main functions, namely to determine if two curves in space intersect, and consequently segment the intersected curves, and to determine whether a set of boundary curves lies inside an *outer* set of boundary curves. The first function is of utmost importance as no two curves representing the modified geometry may intersect at any location other than the end points of those curves. The reason for this is when the existing geometry is modified, to allow for the representation of the contact surface areas between adjacent components, new sets of boundary curves are selected by choosing appropriate curves attached to the end point of each curve. Therefore, all curves must be connected at end points only to allow for this curve selection process.

In Figure 7.4.2.1, the depicted plate is represented by the following sets of *closed boundary curves*:

$$Set_1 = [c1, c2, c3, c4], \quad Set_2 = [c5, c6, c7, c8], \quad Set_3 = [c9, c10, c11, c12]$$

In this example, Set_1 forms an *outer* boundary which contains the other two *inner* boundary sets (Set_2 and Set_3). The method of intersections is used in similar situations to determine which sets of boundary curves form *inner* and *outer* boundaries.

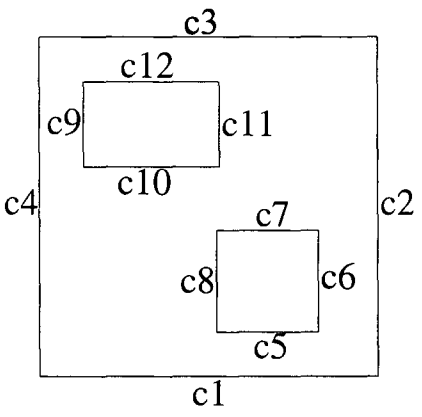


Figure 7.4.2.1 Plate represented by closed boundary curves.

The necessity or importance of this method will become more apparent at a later stage when it is used in the context of the program structure. Initially, the researcher intended

to use "*parametric equations of lines in space*" to query the possibility of two lines intersecting. The parametric equations for a line in space passing through (x_0, y_0, z_0) and parallel to the vector $(V = a_1\mathbf{i} + a_2\mathbf{j} + a_3\mathbf{k})$ can be represented as follows [61] :

$$x = x_0 + a_1 t$$
$$y = y_0 + a_2 t$$
$$z = z_0 + a_3 t$$

It was initially proposed that if two lines in space were each defined by a set of parametric equations, then these equations could be solved simultaneously for the value of *t*. The calculated value of *t* could then be substituted back into the parametric equations and the corresponding *x*, *y*, and *z* values, for the point of intersection, could then be determined. However, problems calculating the value of *t* in certain situations were soon discovered resulting in an alternative method being sought. The major problem was associated with the initial boundary conditions (the coordinates of the origin point). Depending on the location of the origin point, situations arose where the value of *t* could not be solved for a specific value. In other words, the value of *t* could be successfully instantiated with any real number. This uncertainty concerning the value of *t* would have resulted in the formation of complex, and possibly problematic algorithms. Therefore, a simpler more robust, although probably more lengthily, method was developed.

The difficulty inherent in calculating the intersection point of two curves in three-dimensional space was simplified by projecting the curves onto the three orthogonal planes (*XY*, *YZ*, *ZX*) of a rectangular coordinate system. The problem was therefore reduced to calculating the intersection point on the three planes in two-dimensional space. When a curve in space is projected onto a plane, its projected image on that plane will always take the form of one of the following curve types:

(1) *Infinite (I)* – the curve is defined by its location on the horizontal axis.
(2) *Zero (Z)* – the curve is defined by its location on the vertical axis.
(3) *Normal (N)* – the curve is defined by a gradient (other than zero or infinite) and a intercept on the vertical axis.

(4) *Point (P)* – the curve is viewed as a point on that plane defined by it's location on the vertical and horizontal axes.

The purpose of the intersection method is naturally to determine if two curves intersect in space, therefore, when two curves are investigated for the possibility of an intersection occurring on a particular plane, the four curve types mentioned previously can be arranged into the following combinations:

(1) Infinite – Infinite (I-I)
(2) Infinite – Zero (I-Z)
(3) Infinite – Normal (I-N)
(4) Infinite – Point (I-P)
(5) Zero – Zero (Z-Z)
(6) Zero – Normal (Z-N)
(7) Zero – Point (Z-P)
(8) Normal – Normal (N-N)
(9) Normal – Point (N-P)
(10) Point – Point (P-P)

These ten combinations of curves, or *pairs*, define the allowable representations of the two curves on each of the three orthogonal planes. Figure 7.4.2.2 illustrates the projection of two intersecting curves in three-dimensional space onto the three orthogonal planes. Figure 7.4.2.2 (b) depicts the projection of the curves onto the *XY* plane where they are defined as an *Infinite-Normal pair*. Figure 7.4.2.2 (c) shows the projection of the two curves onto the *YZ* plane where they are represented as a *Horizontal-Normal pair*. Lastly, Figure 7.4.2.2 (d) illustrates the projection onto the *ZX* plane where the curves are defined as a *Normal-Point pair*. To determine if these two curves, or any two curves for that matter, intersect in space, it is necessary to initially establish whether the curves intersect on each of the three planes, a prerequisite for intersection in space.

If these conditions or requirements for intersection are realised, it is then necessary to determine the *type of intersection* that exists between the two curves. In other words, the curves need to be segmented, into predefined groups, at the points of intersection. The

importance of this procedure has been repeatedly emphasised due to it's vital function in the program structure. The possible types of intersection are displayed in Figure 7.4.2.3. Conveniently, the types of intersections are divided into two main groups, namely *normal* and *colinear* intersections. It must be noted to avoid unnecessary confusion that the name *normal* does not imply that the curves intersect at right angles to each other, but rather at any angle, other than the curves being parallel.

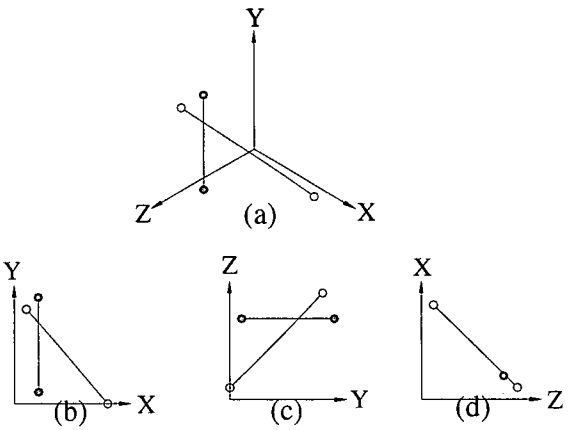


Figure 7.4.2.2. (a) Two curves in a rectangular coordinate system. (b) Projection onto the XY plane. (c) Projection onto the YZ plane. (d) Projection onto the ZX plane.

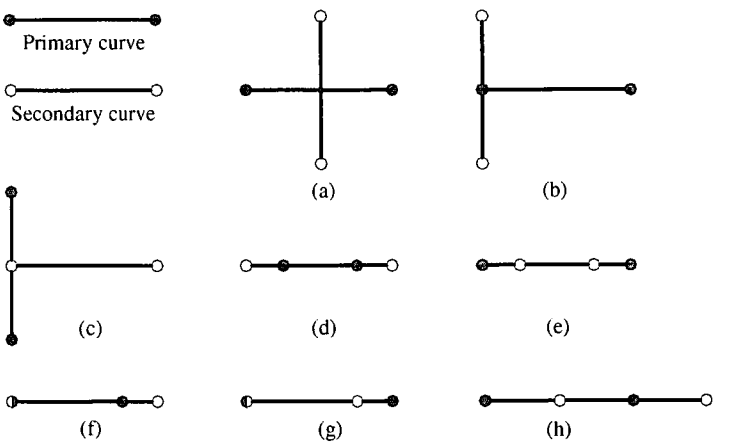


Figure 7.4.2.3. Possible intersection types. (a)-(c) Normal intersections. (d)-(h) Colinear intersections.

Bearing in mind the objective of determining the contact surface areas between the *primary* component (the component to be analysed) and the *secondary* components (the

adjacent components), the only intersections to be analysed are those between primary and secondary components or just between secondary components. In Figure 7.4.2.3, the intersections of *primary curves* (curves belonging to the primary component) and *secondary curves* (curves belonging to the secondary component) are displayed. The intersections in Figure 7.4.2.3 result in the creation of the curves displayed in Table 7.4.2.1. The intersection between any two curves will always be defined by one of the intersection types displayed in the table. Sufficient background information, in the form of rules or conditions governing the curve intersections has been presented, and this should enable the logical reasoning behind the *method of intersections* to be more clearly understood. The method can be summarised as follows:

Figure	Type	Result
(a)	$Normal_1$	4 new curves (2 primary and 2 secondary)
(b)	$Normal_2$	2 new secondary curves
(c)	$Normal_3$	2 new primary curves
(d)	$Colinear_1$	2 new secondary curves
(e)	$Colinear_2$	2 new primary curves
(f)	$Colinear_3$	1 new secondary curve
(g)	$Colinear_4$	1 new primary curve
(h)	$Colinear_5$	3 new curves (1 primary, 1 secondary and 1 shared)

Table 7.4.2.1. The eight possible curve intersection types.

(1) Select two curves and obtain the x, y and z coordinates of their end points, (85 p A-14).

(2) Initially consider projections onto the XY plane, rule (87 p A-14).

(3) Select the first curve.

(4) Obtain the horizontal and vertical coordinate information of both the end points, and from this information, calculate the *change in the horizontal* ($\triangle$h) and *change in vertical* ($\triangle$v), rules (90-93 p A-14).

(5) From $\triangle$h and $\triangle$v obtained above, determine the *curve type* for this particular projection (N, I, Z or P). The curve type is established by determining the gradient ($\triangle$v / $\triangle$h). An infinite gradient results in an *infinite curve*, a zero gradient results in a *zero curve* and any gradient in between results in a *normal curve*. If both end points are coincident, a *point* is created on that plane, rules (94-97 p A-15). The

following facts, defined by the functor *curve1*, are used to represent the four *curve types*:

N – curve1("normal", Integer, M, C)

I – curve1("infinite", Integer, I_h, na)

Z – curve1("zero", Integer, I_v, na)

P – curve1("point", Integer, C_h, C_v)

Where: *Integer* – defines a number that allows two similar curves (curves of the same type) to be identified individually.

M – defines the gradient of the normal curve.

C – defines the intercept on the vertical axis.

I_h *and* I_v – define the intercept on the horizontal and vertical axes respectively.

C_h *and* C_v – define the horizontal and vertical coordinates respectively.

na – defines an argument that is not required.

(6) Repeat steps (4) and (5) for the second curve.

(7) From the *curve1* facts representing the two curves, the program investigates the possibility of these two curves intersecting. At this stage the curves are assumed to have infinite length (unless of course the curve is represented by a point on that plane), and therefore there is a possibility that the intersection occurs outside the actual curve segment. Consequently, this initial result only establishes a probable intersection, but it allows the program to distinguish between a possible intersection and a situation where intersection is impossible. If intersection is deemed impossible for these two curves on this plane, the program terminates any further investigation between these curves, and a new set of curves is selected, rules (98-107 p A-15). Depending on the result obtained, the following four facts, defined by the functor *result*, are used to represent this initial intersection information:

result("intersect", A, B)

result("colinear", A^1, B^1)

result("coincident", A, B)

result("none", "M", "M")

Where: *A* – represents the horizontal coordinate or an unused variable.

B – represents the vertical coordinate or an unused variable.

A^1 *and* B^1 – same as above, except when the two curves are colinear having a gradient between zero and infinite. In these situations, gradient and vertical intercept are represented respectively.

"intersect" – represents the intersection between two non-parallel curves.

"colinear"- represents two curves which are colinear.

"coincident" - represents the projection of two curves as coincident points.

"none" – represents the failure of two curves to intersect.

(8) If the curves in (7) were found to intersect, the next step is to determine whether the intersection point lies on the actual specified curve segment (between the end points of the curve). If the intersection point lies outside the end points of the curve, all further calculations concerning these two curves are terminated, rules (108-109 p A-15). This final planar intersection information is represented by the following fact defined by the functor *status1*:

status1(Type, Integer)

Where: *Type* – represents the type of planar intersection ("intersection", "colinear", "coincident".

Integer – represents an integer between 1 and 3 that allows the program to distinguish between intersections on each of the three orthogonal planes.

(9) Steps (2) to (8) are repeated for the *YZ* plane, rule (88 p A-14).

(10) Steps (2) to (8) are repeated for the *ZX* plane, rule (89 p A-14).

(11) At this stage, depending on the outcome of the previous steps, either the solution procedure for the two curves in question has been abandoned and a new set of curves selected due to the lack of intersection in any one of the planes, or some type of intersection, on each of the planes, has been discovered. The next step is to examine the results and to determine whether the two curves intersect, or are

colinear, in three-dimensional space. This is done by defining the allowable combinations of the types of planar intersections. Table 7.4.2.2 presents the five allowable combinations and the final results of the *intersection method.* Rules (112-116 p A-16). The three planar intersections will always be defined by one of the combinations displayed in the table, resulting in, depending on that particular combination, either an intersection of the curves, or the curves being colinear in space.

Combination	Type	Type	Type	Result
1	"intersect"	"intersect"	"intersect"	Intersection
2	"intersect"	"intersect"	"colinear"	Intersection
3	"intersect"	"colinear"	"colinear"	Intersection
4	"coincident"	"colinear"	"colinear"	Colinear
5	"colinear"	"colinear"	"colinear"	Colinear

Table 7.4.2.2. Intersection method results table.

(12) If the result obtained from the previous step is an intersection, then the program continues with the execution of this step. Otherwise the program proceeds to (13). The function of this step is to calculate the coordinates of the point of intersection in space, from the coordinates of the planar intersections. This procedure is performed using the following logical method represented by rules (124-133 p A-16).

a. Investigate the results from the *XY* plane for a colinear intersection (*result("colinear", A, B)*), where the arguments *A* and *B* represent the gradient and vertical intercept respectively. If such an intersection does not exist, then proceed to (b). For an intersection of two curves to occur in space, only one such planar colinear intersection, where the lines are colinear at any angle between zero and infinity, is permitted. This assumption can be more easily understood by trying to visualize the following scenario. Attempt to picture two curves in space which have a colinear projection on the *XY* plane, with the curves inclined at 45° to the *x* axis (the type of colinear intersection described above). Now, remembering that the curves are required to intersect and not be colinear in space, attempt to visualise a projection of these two curves onto either the *YZ* or *ZX* plane, where the

curves can be viewed as being colinear. If the curves are to intersect in space, this visualisation is impossible. Alternatively, one can easily picture the same scenario where the two curves are colinear in space. Therefore, it can be safely assumed that if the projection onto the *XY* plane depicts a colinear intersection, where the curves are inclined at any angle between zero and infinity, the subsequent projections onto the remaining two planes must depict the two curves intersecting. Consequently, when calculating the coordinates of the intersection point, the y and z values can be obtained directly from the intersections on the *XY*, *YZ* and *ZX* planes. Knowing the y coordinate value, the corresponding x value can be calculated using the gradient (A) and vertical (y) intercept (B) on the *XY* plane, as shown by the following equation:

$$x = \frac{(y - B)}{A} \qquad (7.4.2.1)$$

b. Investigate the results from the *YZ* plane for a colinear intersection (*result("colinear", A, B)*), where the arguments A and B represent the gradient and vertical intercept respectively. If such an intersection does not exist, then proceed to (c). If this type of planar colinear intersection does exist, then obtain the x and z coordinates directly from the *XY* , *YZ* and *ZX* planes. By substituting the z coordinate, the gradient (A) and vertical (z) intercept (B) into the following equation, the corresponding value of y can be calculated.

$$y = \frac{(z - B)}{A} \qquad (7.4.2.2)$$

c. Investigate the results from the *ZX* plane for a colinear intersection (*result("colinear", A, B)*), where the arguments A and B represent the gradient and vertical intercept respectively. If such an intersection does not exist, then proceed to (d). If this type of planar colinear intersection does exist, then obtain the x and y coordinates directly from the *XY*, *YZ* and *ZX* planes. By substituting the z coordinate, the gradient (A) and vertical (x)

intercept (B) into the following equation, the corresponding value of z can be calculated.

$$z = \frac{(x - B)}{A} \qquad (7.4.2.3)$$

d. If no such planar colinear intersections exist, then the x, y and z coordinate values can be obtained directly from the projections onto the XY, YZ and ZX planes.

(13) There are five different forms of colinearity as depicted in Figure 7.4.2.3 (d)-(h). The function of this last step is to divide the curves into the appropriate curve segments. The final curve segments must only contain two end points, naturally, one at either end of the curve segment. The following method is utilised to perform this function, rules (139-148 p A-17).

a. Firstly, the length of each of the curves is calculated. If the length of the first curve is greater than or equal to that of the second curve, then proceed to (b). Otherwise proceed to (c).

b. Initially the program checks for colinearity where the second curve shares a common end point with the first, as illustrated in Figure 7.4.2.3 (f) and (g). If common end points are detected, the curves are segmented accordingly. If no common end points are found, the program attempts to establish whether the second curve lies in between the end points of the first curve, as in Figure 7.4.2.3 (e). The method for determining this is simple. Assume that the first curve (*c1*) has end points *p1* and *p2*, and that the second curve (*c2*) has end points *s1* and *s2*. The program then finds the closest point on *c2* to *p1* (for argument sake, assume that *s1* is closer). Let the distance between *p1* and *s1* be *d1*, and between *p2* and *s2* be *d2*. For *c2* to lie between the end points of *c1*, the following condition must be satisfied:

Length of c1 = d1 + length of c2 + d2

If this condition is proved to be true, the curves are segmented as required. If this condition is proved false, the program then determines whether a single end point of *c2* lies on *c1*, as in figure 7.4.2.3 (h), whereupon, the curves are divided into the appropriate segments. The method used here is similar to the one described above and does therefore not require any further explanation. If the colinear relationship between the two curves cannot be defined by one of the above scenarios, then the program concludes that the two curves are colinear outside the boundaries of their end points, and therefore no curve segmentation is required.

c. This step is virtually identical to (b), except that the order of the two curves being compared is reversed, other than that, the methodology is the same.

To help demonstrate the functionality of the *method of intersections*, the following example has been used, see Figure 7.4.2.4. The illustration consists of two arbitrary shapes defined by their labelled boundary curves and end points. The object of this example is to reveal how the original boundary curves are divided into the final curve segments.

A *Prolog* fact, defined by the functor *c* and containing three arguments, is used in this example to represent each curve and the end points associated with that curve. The first argument is the name of the curve, the second and third arguments are the end points of the curve. The primary and secondary sets of boundary curves, illustrated in Figure 7.4.2.4 (a) and (b), can be defined by the following lists:

Primary set - [c(1, a, b), c(2, b, c), c(3, c, d), c(4, d, a), c(5, e, f), c(6, f, g), c(7, g, h), c(8, h, e)]

Secondary set - [c(9, i, j), c(10, j, k), c(11, k, l), c(12, l, i)]

Each list can contain an indefinite number of elements. For the purpose of this example, each element is represented by the fact defined by the functor *c*. The first step, in solving the curve intersections, is to take the first curve, *c(1, a, b)*, from the primary set and determine if any curves from the secondary set intersect with this curve. As can be seen from the diagram, no curve segments from the secondary set intersect. It should be noted that the line equations representing the curves do not take the location of the

curve segment into account, and therefore curves *c(10, j, k)* and *c(12, l, i)* are initially found to intersect curve *c(1, a, b)*. However, when the end points defining the locations of the curve segments are investigated, the intersections are calculated to occur outside the boundary of the curve segments. Consequently, these curves are determined not to intersect. In a situation like this, where a curve from the primary set is found not to intersect any curve from the secondary set, a *marker* is placed on the primary curve to enforce that it is not used for any further comparisons.

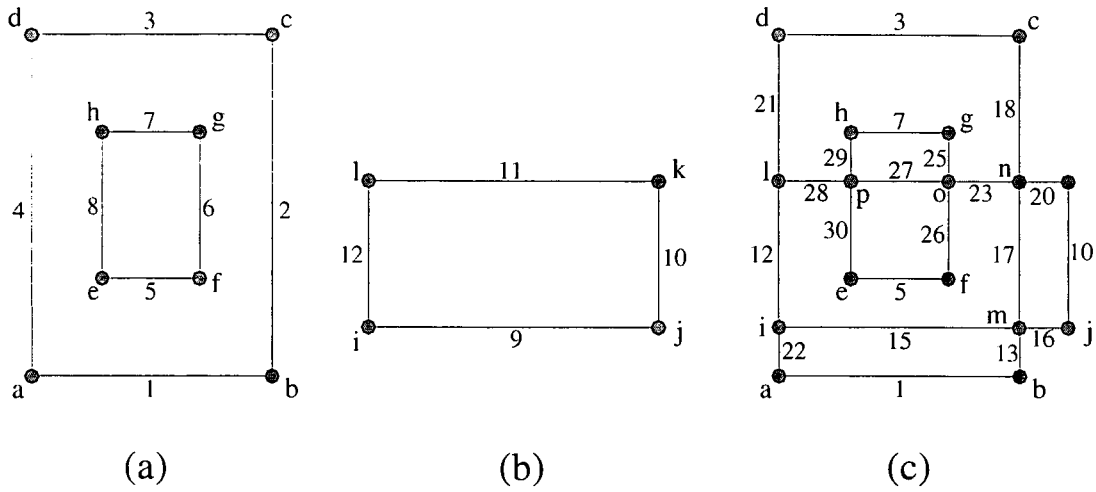


Figure 7.4.2.4. (a) Primary set of boundary curves. (b) Secondary set of boundary curves. (c) Combined boundary curves displaying final line segments.

The next curve, *c(2, b, c)*, is then chosen, and immediately found to intersect with curve *c(9, i, j)*. This intersection results in the creation of two new primary and secondary curves, namely, curves *c(13, b, m)* and *c(14, m, c)*, and curves *c(15, l, m)* and *c(16, m, j)* respectively. The lists containing the curves are then updated by replacing the original curves with the newly created curves. Curve *c(13, b, m)*, the first of the two curves that replaced curve *c(2, b, c)*, is then investigated for the occurrence of an intersection, and when no intersection is found, a *marker* is placed on this curve. Subsequently, curve *c(14, m, c)* of the primary set is selected, and is determined to intersect with *c(11, k, l)*, resulting in the creation of two more primary and secondary curves, namely curves *c(17, m, n)* and *c(18, n, c)*, and curves *c(19, n, l)* and *c(20, n, k)*. These curves are then substituted with the original curves, and then too investigated for intersections. This

process of *checking* for intersections, *creating* new curve segments, and *replacing* the original curves with the newly created curves, is continued until all curves have been allocated a *marker*. When this has been achieved, the end result is a set of curves connected only at their end points. The final primary and secondary sets can be represented by the following lists:

> Primary set – [c(1, a, b), c(13, b, m), c(17, m, n), c(18, n, c), c(3, c, d), c(21, d, l), c(12, l, i), c(22, i, a), c(5, e, f), c(26, f, o), c(25, o, g), c(7, g, h), c(29, h, p), c(30, p, e)].
>
> Secondary set – [c(15, i, m), c(16, m, j), c(10, j, k), c(20, k, n), c(23, n, o), c(27, o, p), c(28, p, l), c(12, l, i)].

This example concludes the explanation of the two paradigms, namely the *method of intersections* and the *area method*, which are both used on several occasions throughout the following sections.

7.4.3 DEFINING THE SECONDARY GEOMETRY

This is the first section, concerned with definition of the contact surfaces, that is responsible for describing the creation of the relevant secondary geometry. Remembering that a FE model is only constructed for the primary component, the sole purpose of the secondary geometry is to identify the contact surfaces existing between these two components. Consequently, depending on the physical relationship inherent between the adjacent components, only a certain portion of the secondary geometry is usually required to be represented. The selection of the appropriate geometry will be discussed after the method utilised for orientating the secondary component relevant to the primary component has been described.

In section 7.2, the *Prolog* facts, used to represent the connectivity information between the primary component and its adjacent components, were introduced. In terms of connectivity, the most significant fact is the one defined by the functor *connected*. This fact contains information about the primary component, as well as a list of all the adjacent components and their orientations to the primary component. Having access to this information, the program can automatically adjust the secondary component's face

set information, and thus, place the component in the required position. The following method is used to orientate the secondary components relative to the primary component, thereby modifying the face set information describing their geometry:

(1) The name of the first secondary component, contained in the first element in the list of the *connected* fact, is obtained. Using this name, the data file, containing the face set information pertaining to this component, is opened. Rule (2 p A-8).

(2) The relative rotations about the three orthogonal axes are retrieved from the *connected* fact. The three angles, initially represented in degrees, are then converted into their radian equivalents. The rotational transformation is performed in three stages. The first stage (3) allows for the rotation about the x axis, the second (4) allows for rotation about the y axis and the third (5) allows for rotation about the z axis. Rule (4 p A-8).

(3) When a point is rotated about a certain axis, only its location relative to the other two axes will change. In other words, if a point located at (6,0,1) is rotated through 90° about the *x axis* for example, its corresponding location after the transformation will be (6,-1,0). Therefore, rotations of a point about the *x axis*, will only result in changes to the *y* and *z* coordinates of that point. Initially the *y* and *z* coordinates of the first vertex of the face are selected, and the angle of the point, on the *YZ* plane relative to the *y axis*, is calculated. The angle is modified, if required, according to its location in one of the four quadrants of the *YZ* plane. The angle of rotation about the *x* axis is then added to this angle, and from this new angle, the modified *y* and *z* coordinates are then calculated. The coordinates of the remaining two vertices of the face are then calculated in the same fashion. This process is then repeated for every face describing the geometry of the secondary component. The new face set information, containing the modified *y* and *z* coordinates, is concurrently stored in the external database, to allow for further modification in the following steps. Rules (5,8,9 p A-8).

(4) The rotation about the *y axis* is performed in an identical manner to that of the *x axis*. The only difference in the explanation is that because the rotation is about the *y axis*, the change in coordinate information will occur to the *x* and *z* coordinates on the *ZX* plane, rules (6,8,9 p A-9).

(5) The rotation about the *z axis* is also performed in an identical manner to that of the *x axis*. The only difference in the explanation is that because the rotation is about the *z*

axis, the change in coordinate information will occur to the x and y coordinates on the XY plane, rules (7,8,9 p A-9).

(6) The next step in the coordinate transformation of the face set information, is the addition of the x, y and z translations to the respective coordinates of the face set vertices. Once this has been completed for all faces, the modification of the secondary face set information is complete. Rule (10 p A-9).

At this stage, the program is now prepared to commence the creation of the secondary geometry, from the modified face set information. The geometry is created in virtually the same method as that of the primary geometry, described in section 7.3. Therefore to avoid unnecessary repetition, the explanation will not be repeated. However, one small difference between the two methods does deserve a brief explanation. As mentioned before, unlike the creation of the primary geometry, where all faces contribute to the final boundary representation, during the creation of the secondary geometry, only those boundary surfaces which are in direct contact with the primary component are required. This condition is realised by initially deleting all the secondary face set information that does not lie on any one of the primary planes (planes which contain primary geometry). Of course, this condition alone does not imply that all the remaining secondary geometry will be in direct contact with the primary component, as it is totally acceptable for two surfaces lying on the same plane to be discrete. However, this condition greatly increases the efficiency of the program by coercing the program to consider a smaller group of faces which are more likely to be in contact with the primary component. Rules (11- 52 p A-9) are responsible for creating the appropriate secondary geometry.

7.4.4 INITIAL GEOMETRY GROUPING

Due to the complexity of certain components, especially three-dimensional components containing geometry on a vast number of planes, the program would be unnecessarily inefficient if it was required to compare geometry from all the planes, when only the geometry on planes which are co-planar are required to be compared. Stated differently, if the program was required to calculate the intersection of hundreds of curves lying on different planes, it would make far greater sense to initially group the curves according to the plane on which they lie, and then investigate for intersections only between

curves belonging to the same group. This is exactly what the program does as soon as the secondary geometry has been created. A fact, defined by the functor *status* and containing three arguments, is created to represent the sets of secondary boundary curves which occupy the same plane as a set of primary boundary curves contained in the fact *curvelist*. The fact takes the following form:

status(Name, List, Plane)

Where: *Name* – represents the name of a *curvelist* fact containing a set of primary boundary curves.

List – represents a list of elements, or possibly an empty set if no secondary geometry occupies the same plane as *Name*. Each element takes the form of a *string variable* representing the name of a fact defined by *scurvelist*, containing a set of secondary curves lying on the same plane as *Name*. This fact defined by *scurvelist* is a secondary component's equivalent to the *curvelist* fact.

Plane – is a compound data object in the form of *pl(A, B, C, D)*. Where *A*, *B*, *C* and *D* represent the coefficients of an equation defining the plane common to a set of boundary curves.

The compound data object *pl(A, B, C, D)* in the above fact allows all secondary curves, co-planar to a set of primary curves, to be grouped into a common list. Rules (53,54 p A-12) are used to perform this grouping function. Once this grouping of primary geometry with co-planar secondary geometry has been accomplished, the program attempts to further group the primary boundary curves contained in the *curvelist* fact. The goal of this further subdivision or grouping, is to discretise the closed boundary curves belonging to the *curvelist* fact. The primary reason for performing this grouping is the ability of the program to distinguish between inner and outer boundaries, as illustrated previously in Figure 7.4.2.1.

The first step in achieving this boundary discretisation is to form subgroups of primary boundary (*pb*) curves from the *curvelist* fact. These *pb* curves are curves which form closed boundaries (the curves join from end point to end point until a closed loop is formed). When all the subgroups of *pb* curves have been identified from each curvelist

fact, they are placed in a list called *PbList*, containing the groups of *pb* curves from a single *curvelist* fact. Consider the representation of Figure 7.4.2.1. This component or shape would be defined by the following *curvelist* fact:

curvelist(Name, [c1, c2, c3, c4, c9, c10, c11, c12, c5, c6, c7, c8])

As mentioned above, the first step is to discretise this list of boundary curves into sets of *pb* curves. Subsequently these separate sets of closed boundary curves, representing the geometry of a component or the geometry on a plane of a particular component, are regrouped for convenience, as demonstrated below:

pb(Name, [c1, c2, c3, c4])
pb(Name, [c5, c6, c7, c8])
pb(Name, [c9, c10, c11, c12])
pblist(Name, [c1, c2, c3, c4], [c5, c6, c7, c8], [c9, c10, c11, c12])

To perform these grouping functions, the following logical reasoning is employed by the program.

(1) The first *status* fact, *Name*, is selected, and the list contained within this fact is examined for the presence of secondary curves. If no secondary curves exist, the boundary curves belonging to the *curvelist* fact, also represented by *Name*, are determined to represent a *free surface*, as there is no possibility for inter-component contact on this surface. If the list is not empty, an empty *pblist* fact is created, represented by *Name*, and the boundary curves belonging to the *curvelist* fact, also represented by *Name*, are sent to (2), rule (55 p A-12).

(2) The program removes the first curve from this set of boundary curves, and retrieves the information regarding the end points of this curve. For reasons that will become evident at a later stage, the program starts to investigate the end point information for the x, y or z coordinate of the largest value. All end points are examined, and the largest value obtained is stored in a fact defined by the functor *big*, rule (67 p A-12). The reason for selecting the two end points from the first curve, is to enable a starting and ending point of a set of closed boundary curves to be defined. These end points are then sent to (3). Rule (57 p A-12).

(3) The first end point represents the starting and ending point of the closed boundary curves. This point is fixed and is used to check when a closed loop has been defined. The second end point defines the start of each new curve in the closed boundary. This end point, originally representing the end of the first curve, then represents the start of the next curve attached to this curve. In other words, the program attempts to find a another curve that has this end point as one of its end points. The remaining end point from this curve is then compared to the original end point. If the end points match, a closed loop has been defined. If no match is detected, the program then seeks to find a new curve attached to this end point. Therefore, by following this methodology, a continuous set of curves, attached form end point to end point, can be detected, and consequently, a set of closed boundary curves can be defined. Rules (58-60 p A-12).

Once the primary boundary curves have been grouped into discontinuous sets of closed boundary curves, the program then follows the same reasoning and performs the same functions on the secondary boundary (*sb*) curves, rules (61-66 p A-12). The only difference here, are the names used to represent the facts, as shown below:

curvelist(Name, List)	$\rightarrow$	scurvelist($Name_1$, $List_1$)
pb(Name, List)	$\rightarrow$	sb($Name_1$, $List_1$)
pblist(Name, List)	$\rightarrow$	sblist($Name_1$, $List_1$)

Before the program can begin defining the inner and outer boundary curves, one more fact needs to be created. The intention of this fact is to enable a curve to be constructed from a specified location to an end point belonging to a *pb* or *sb* curve. The reason for constructing this curve will be explained shortly, but for the moment, the method used will be sufficient. The curve is constructed, like any other curve, between two end points. The one end point is arbitrarily selected from a *pb* or *sb* curve. The other end point has been appropriately termed the *planar reference point* (*prp*). The *prp* is a point lying on the same plane as a set of boundary curves, but it is created at a location that is far beyond the boundaries of these curves. The reason for this relatively distant location is to ensure that the constructed curve will always have one end point outside the outermost boundary of the component. Previously, a fact defined by *big* was created.

The reason underlying the creation of this fact should now be evident. The argument contained in this fact is the largest coordinate value obtained from the end points of the boundary curves. The *prp* is created at a location that was arbitrarily chosen to be five times greater than the value contained in the fact *big*. This ensures that the *prp* is always outside the boundaries of the component's geometry. Rules (68-71 p13).

The external database should now contain all the necessary information to begin defining the inner and outer boundary curves. The program initially examines the curves belonging to *pblist*. The objective is to determine which *pb* curves form the outer and inner boundaries. From this point onwards, *primary outer boundary* curves will be referred to as *pob's*, and the inner boundary curves as *pib's*. A simple method, known as the *"Counting-Intersection Method"*, was devised that allows the program to distinguish between *pob's* and *pib's*. The method relies on a curve being constructed from the *prp* to an end point of a *pb* curve. Because at this stage the external database contains no information about inner or outer boundary curves, an arbitrary end point is selected on any one of the *pb* curves contained in *pblist*. Consider Figure 7.4.4.1 on the following page. To allow for a clearer representation of the two shapes, the *prp* has not been located in its correct position, but instead it has been moved closer to the outer boundaries of the two shapes. The functionality of the *counting-intersection method* is based on the following two conditions:

(1) If the number of intersections, for one *pb* set between the end points of the new constructed curve, is *odd*, then the primary boundary to which the constructed curve is connected lies inside the *pb* curves which are intersected by the constructed curve. In other words, in Figure 7.4.4.1 (a), the constructed curve between the *prp* and the end point contained in *pb2*, intersects *pb3* at three separate locations between the end points of the constructed curve, and therefore, *pb2* is a *pib* and *pb3* a *pob*. It can be deduced from this assumption, that because *pb3* is the outer boundary, and because only one outer boundary is permissible, that *pb1* must therefore also be a *pib*.

(2) If the number of intersections, for one *pb* set between the end points of the new constructed curve, is *even*, then the primary boundary to which the constructed curve is connected lies outside the *pb* curves which are intersected by the constructed curve. To reiterate, in Figure 7.4.4.1 (b), the constructed curve between

the *prp* and the end point of *pb1*, intersects *pb3* on four separate occasions between the end points of the constructed curve, and therefore, *pb1* lies outside *pb3*. However, as depicted by the illustration, this fact alone cannot be used to deduce that because *pb1* lies outside *pb3*, that *pb1* is the outer boundary. It should be noted though that (b) is a hypothetical example, and that this situation, where *pb1* lies outside *pb3*, can never occur. In reality, *pb1* and *pb3* would belong to different *curvelist* facts and would therefore never be compared.

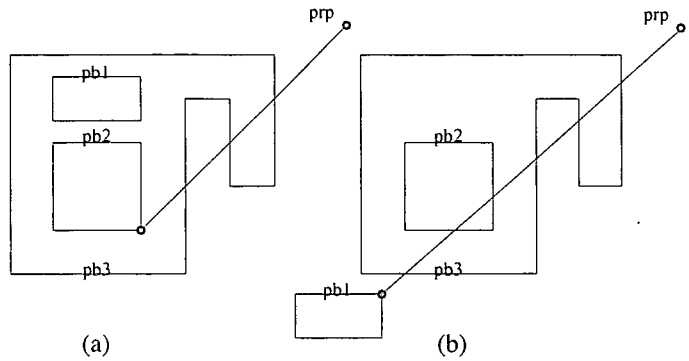


Figure 7.4.4.1. (a) Shows an inner boundary *pb2* within an outer boundary *pb3*. (b) Shows a boundary *pb1* outside the outer boundary *pb3*.

It is important to remember that if a *prp* is created such that the curve, subtending this point and the end point of the inner boundary curve, intersects one or more of the outer boundary curves at their end points, then a new *prp* is created at a location offset from the original. This is performed simply by multiplying the original *x, y* and *z* coordinates by an adjustment factor. In order to perform the boundary checking functions described above, the following logical reasoning was utilised:

(1) The name and list are selected from the first *pblist* fact. The list is examined for the number of elements that it contains. If the list only contains one element (one *pb* curve set), then obviously, there are no inner boundaries and this *pb* set automatically becomes the *pob*, as defined by the fact:

pob(Name, List)

If the list contains more than one element, the program proceeds to (2) with the list and name from this *pblist* fact. Rule (72 p A-13).

(2) The program selects the first curve from the list obtained from (1). A curve is then constructed from an end point of this curve to the *prp*, and then stored in the external database. The program subsequently proceeds to (3) with the name from the *pblist* fact, the remainder of the list (excluding the first curve), and the name of the first curve. Rule (73 p A-13).

(3) The main function of this step is to check the database for certain asserted information. It is important to remember that most of these functions use recursion (where the same function is called repeatedly, creating a program that "loops" itself until certain conditions are satisfied), resulting in the same step being used repeatedly, until some condition is realised. Therefore information that is asserted in subsequent steps can be detected when the cycle is repeated. For example, if the fact *end* (asserted in one of the following steps) is detected, the program is prompted to restart the boundary checking process with a new *pblist*. If the fact *newprp* is detected, the program creates a new *prp* and restarts the program at step (1). If neither *end* nor *newprp* are detected, the program proceeds to (4) with the first set of *pb* curves selected from the remainder of the list from the *pblist* fact. Rule (74,75 p A-13).

(4) Using the *method of intersections*, the program searches through the set of *pb* curves attempting to find all the possible intersections. When all the curves in this *pb* set have been checked for intersections, the program proceeds to (5). Rule (77 p A-13).

(5) If there were an *odd* number of intersections in the previous step, the *pb* set containing the curves that were intersected is defined as a *pob*. The fact *end* is asserted, and the remaining *pb* sets in the list from the fact *pblist* are defined as *pib's*. This fact takes the following form (the second argument, $Name_2$, allows for multiple *pib's* to be defined):

pib($Name_1$, $Name_2$, List)

If there were an even number of intersections, the *pb* set containing the curves that were intersected is defined as a *pib*. The program then goes back to (3), where the remainder of the *pb* sets are examined. Rule (76 p A-13).

This boundary checking process continues until one of the *pb* sets is proved to be a *pob*. Alternatively, if this fails, the process is continued until all *pib's* have been identified,

and the remaining *pb* set is then defined as the *pob*. In a similar manner to the above method, the secondary geometry is grouped into inner and outer boundaries. The methodology is almost identical to that of the primary set and will therefore not be repeated. However, there is a small difference in the facts representing the inner and outer boundaries. The two facts are modified to allow for an extra argument representing the group of the secondary geometry. This argument allows for the concurrent representation of multiple secondary components attached to the primary component. Rules (78-84 p A-14). These secondary facts are represented as follows:

sob(Name, Group, List)

sib($Name_1$, Group, $Name_2$, $List_2$)

7.4.5 BOUNDARY INTERSECTIONS

The *method of intersections* has already been extensively discussed and does therefore not require any further reiteration. As mentioned previously, the objective of this section is to determine all the possible intersections between primary and secondary curves, and also between the secondary curves themselves. Once the intersection point between two intersecting curves has been calculated, the program is responsible for "breaking" or "segmenting" the original curves at the intersection point, and then creating the new curve segments. The appropriate facts containing the original curves are then updated accordingly. Rules (151-170 p A-18) are used for checking primary-secondary intersections, while rules (183-196 p A-20) perform the secondary-secondary intersection checking. After the completion of these functions, all the relevant geometry has been segmented accordingly and the lists containing the boundary curves, describing the component's geometry, contain only curves that intersect or connect at the end points of those curves.

7.4.6 CURVE CLEANUP AND LABELLING

Bearing in mind that the primary concern of this entire section is to create a FE model of the primary component, and that the secondary geometry's only function is to

provide the location of the contact surface on the primary component, it is understandable that situations will arise, where secondary curves are located outside the boundary of the primary component. Clearly theses curves being externally located, cannot be included in the primary geometry without changing the original intended shape of the component. Therefore, these external curves need to be removed from appropriate secondary sets. This concept is illustrated in Figure 7.4.6.1. (a) and (b) represent the secondary and primary geometry respectively. (c) represents the combined geometry and the newly created curve segments as a result of the primary-secondary curve intersections. (d) depicts the final curve segments after the external secondary curves have been removed. The *area method*, described in section 7.4.1, is used to determine whether a secondary curve lies within the boundary of the primary component. Rules (197-204 p A-20) provide the necessary logic behind this curve deletion process.

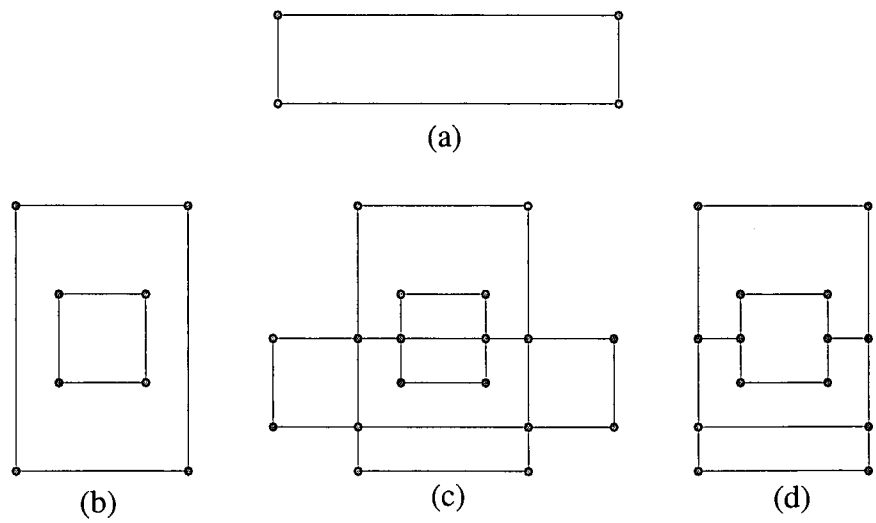


Figure 7.4.6.1. (a) Secondary geometry. (b) Primary geometry. (c) Combined geometry. (d) Final curves after deletion.

The next step can be suitably referred to as curve labelling. This is an important procedure as it allows the program to correctly select the appropriate curve when defining the *free surfaces* and *contact surfaces* of the primary component. Basically, a *free surface* is a surface which is not in direct contact (when two surfaces lie one on top of the other) with another surface, and a *contact surface* is one which is in direct contact with another surface. The curves are labelled according to their relation with the secondary surfaces on a specific plane. Four different types of boundary curves exist:

(1) *No contact (nc)* – this is a primary curve which has no physical connection to the secondary surface. The curve is defined by the data object *nc*.

(2) *Boundary contact (bc)* – this a secondary boundary curve. The curve is defined by the data object *bc(Name, Group)*.

(3) *Full contact (fc)* – this is a primary curve in direct contact with the secondary surface, although it may not lie on the boundary of the secondary surface. The curve is defined by the data object *fc(Name, Group)*.

(4) *Shared (shared)* – this curve forms part of the boundary of both the primary and secondary surfaces, in other words, it is shared between a primary and secondary surface. The curve is defined by the data object *shared*.

The data objects in the above definition, are *Prolog's* definition of a user defined variable. The *Name* and *Group* arguments enable a specific set of boundary curves on a particular secondary component to be represented. The following fact is used to represent the different curve types:

curvestatus(Curve, Type, Name)

Where: *Curve* – represents the name of the specified curve.
Type – represents the type of boundary curve (*nc*, *bc*, *fc* or *shared*).
Name – represents the name of the primary surface to which the curve type belongs. This argument allows a curve, belonging to more than one plane or boundary surface, to have different curve types (if required) on each plane.

The four curve types listed above are determined using the following reasoning or selection strategy:

(1) *Nc curve* – all *nc* curves are detected using a two step process. The first step is simple in it's execution and relies on selecting boundary curves from the *curvelist* facts that have no planar association with the secondary geometry. Consequently, due to this lack of association, all the boundary curves contained in these *curvelist* facts are *nc* curves. This step is implemented by the program by selecting all *status* facts that have empty lists (remember that the list, belonging to the *status* fact,

contains the secondary boundary surfaces connected to a particular *curvelist* fact). Once the required *status* facts have been selected, the corresponding curves in the appropriate *curvelist* facts are defined accordingly. Rules (212-214 p A-21). The second step is initiated by selecting *status* facts that do not contain empty lists. The *curvelist* facts associated with these *status* facts are then chosen and their the boundary curves are selected sequentially. Provided that the curves have not been previously defined as either *shared* or *fc* curves, the program will then determine the location, using the *area method*, of these curves relative to the secondary surfaces contained in the *status* fact lists. If the curves are determined to lie outside the boundaries of the secondary surfaces, then they are defined as *nc* curves. Rules (224-231 p A-21).

(2) *Bc curve* – any curve that belongs to a *scurvelist* fact (all secondary geometry), and has not been previously defined as a *shared* curve, is asserted as a *bc* curve. Rules (220-223 p A-21).

(3) *Fc curve* - *status* facts that do not contain empty lists are selected. The *curvelist* facts associated with these *status* facts are then chosen and their boundary curves are selected sequentially. Provided that the curves have not been previously defined as either *shared* or *fc* curves, the program will then determine the location, using the *area method*, of these curves relative to the secondary surfaces contained in the *status* fact lists. If the curves are determined to lie within the boundaries of the secondary surfaces, then they are defined as *fc* curves. Rules (224-231 p A-21).

(4) *Shared curve* – shared curves are selected in a two-step process. The first step involves selecting boundary curves that are common to both primary and secondary sets. These curves being common to both sets, are then defined as shared curves. Rules (212-218 p A-21). The second step checks for the occurrence of secondary shared curves (curves shared between secondary sets). This check is implemented by initially selecting a secondary curve set (an *scurvelist* fact), and then checking the remainder of the *scurvelist* facts for common curves contained in these facts. Once all of the *scurvelist* facts have been checked against the initial *scurvelist* fact, the initial fact is removed from the selection process and the remainder of the facts are investigated. Rules (232-239 p A-22).

After the completion of the curve removal and curve labelling processes, the program has reached a level where the external database contains all the necessary information to

allow for the primary component's surface geometry to be subdivided or segmented into the appropriate areas of contact and areas which are free. This segmentation process is explained in the following two sections responsible for defining the ultimate geometry representation of the primary component.

7.4.7 DEFINITION OF THE FREE SURFACES

A free surface, as mentioned previously, is essentially a surface, belonging to the primary component that does not lie in direct contact with a secondary surface. It is therefore a surface free from the constraints imposed by any secondary components. When the program is defining the free surfaces, and contact surfaces for that matter, it follows a predetermined *curve selection strategy* that allows the appropriate curves to be selected in the correct order. As each new curve is selected, it is deleted from its appropriate set, and the curve selection process continues until all the curves have been deleted and the new surfaces have been defined. Due to the duplicitous nature of the secondary geometry (secondary curves always belong to both free and contact surfaces), a duplicate of each of the secondary curves is generated and two new facts are created. The first fact is defined by the functor *tempsob*, and contains two arguments. The second fact, defined by the functor *tempsib* and also containing two arguments, is the equivalent representation for the secondary inner boundaries. All inner boundaries are grouped into the one *tempsib* fact. Rules (240-247 p A-22) allow the creation of these two facts.

tempsob(Name, [c_1, c_1, c_2, c_2, ..., c_n, c_n])
tempsib(Name, [c_1, c_1, c_2, c_2, ..., c_n, c_n])

Where: *Name* – represents the name of the temporary outer boundary set.
[c_1, c_1, c_2, c_2, ..., c_n, c_n] – represents the list of outer boundary curves and their duplicates.

The first step in creating the free surfaces is to begin defining the outer boundaries of these free surfaces. The following method is implemented:

(1) Select a *status* fact that contains at least one element (at least one secondary geometry set). Retrieve the name argument (*Name*) from this fact, and using this name, select the corresponding fact defined by *pob*. From this fact obtain the list (*List*) of primary outer boundary curves and send this list along with *Name* to (2). Rules (240,241 p A-22).

(2) Select the first curve (*Curve*) from *List*, and then determine if this curve is a *nc* curve. If it is an *nc* curve, then check if any free surfaces have already been defined. Following a standard naming convention consisting of a letter-number combination, for example "L1", any number of surfaces can be defined simply by adding a unit value to the number part of the previous name. The program selects the specified name for this free surface (*Free*) from this naming convention. The selected curve is then deleted from the appropriate sets. A temporary duplicate curve, having the same end points, is then created from *Curve* and asserted into the database. The fact is defined by the functor *templine*, and contains three arguments as follows:

templine(Curve, P1, P2)

The *templine* fact is essentially used as a "place marker" containing information about the starting end point *P1*, and the current end point *P2*. Stated differently, the program attempts to find another curve having an end point *P2*. If such a curve is determined to exist, the fact *templine(Curve, P1, P2)* is deleted from the external database, and replaced with the fact *templine(Curve, P1, P3)*, where *P3* is the second end point of that curve. Another fact, defining a new free surface, is then created. The list argument (*FList*) in the fact initially only contains a single element, namely *Curve*. This list is expanded during the subsequent steps. The fact is defined by the functor *freesurf*, and contains three arguments as follows:

freesurf(Name, Free, [Curvel[]])

From here, the program proceeds to (4). If the curve was not an *nc* curve, the next curve in *List* is selected and (2) is repeated. This process will be repeated until all curves have been exploited, whereupon, the program proceeds to (3). Rules (248,249 p A-22).

(3) This step is virtually identical to step (2) except for two small differences. Firstly, this step initially searches for a shared curve as apposed to a nc curve. Secondly, once all the curves have been exploited, the program proceeds to (6) as apposed to (3) in the above step. Rules (250,251 p A-22).

(4) In this step the *curve selection strategy* for free surfaces is defined. This selection strategy specifies the allowable curves, in a predetermined order, that the program follows when selecting curves defining the boundaries of the free surfaces. The curves are always selected in the following order: *nc, bc, shared.* When the program implements this step, it initially specifies the *nc* curve type and sends this information to (5). If however, (5) fails with this curve type, the program will then select the *bc* curve type and reattempt step (5). If (5) fails once again, the *shared* curve type will then be chosen and subsequently, step (5) will be repeated. Rule (252 p A-22).

(5) This step is responsible for selecting the appropriate curves according to the curve selection strategy of (4). From *templine(Curve, P1, P2)*, the program retrieves information about the starting end point *P1* and the current end point *P2*. The program then investigates the database for a new curve, namely *NCurve*, having *P2* as one of its end points. If such a curve is found, and it is not already a member of the list contained in the *freesurf* fact, then the other end point *P3* of this curve, is compared to *P1*. If *P3* is equal to *P1*, then a closed boundary containing a free surface has been defined. If the end points are not equal, the *templine* fact is replaced by a new templine fact having the end point *P3* as its third argument (the current end point). The program then goes back to (4). Rules (253-256 p A-22).

At this stage in the definition of the free surfaces, all the outer boundaries, originating from the *pob* set, containing these free surfaces have been defined. The next step is to determine if any inner boundaries are contained within these newly defined free surfaces.

(6) The program searches for a newly created *freesurf* fact that has not yet been checked for the presence of inner boundaries. Using the *Name* argument from this fact, the program retrieves the *status* fact having the same name, and also containing the list of secondary surfaces, namely *SList*, lying on the same plane. This information is then sent to (7). Rule (262 p A-23).

(7) The first element of the list *SList* is selected, and from this information, the list of boundary curves, namely *BCList*, belonging to a *sob* fact is selected. The first curve, Curve1, of *BCList* is selected and the new information is sent to (8). Rules (263,264 p A-23).

(8) A curve is constructed from the primary reference point (*prp*) to the midpoint of *Curve1*, and using the *counting-intersection method*, the program determines whether *Curve1*, and therefore the *sob* set containing *Curve1*, lies within the outer boundary of the free surface. If the sob set is determined to lie within the outer boundary, then this set of curves is added to the list *FList*, of the *freesurf* fact. If the *sob* set is not contained within the outer boundary, the program then investigates the remainder of the *sob* sets. When all *sob* sets have been attempted, the program then proceeds to (9). Rules (266,267 p A-23).

(9) Steps (7) and (8) are repeated, except this time, the goal is to search for the presence of *pib* sets within the free surface outer boundaries. Rules (268-275 p A-24).

The program would have now successfully defined all the free surfaces, including their inner boundaries that originated from the *pob* sets. Although this type of free surface usually constitutes the majority, there is still a second type of free surface, one that originates from the sib set, that needs to be defined. It is easy to picture this type of free surface. Remember that the outer boundaries of surface define the outer edges or limits of that surface, and that the inner boundaries define the inner limits or wholes in that surface. Therefore, any surface, belonging to the primary component that lies within the boundaries of the *sib* curves, will consequently be a free surface.

(10) From the *status* fact, a primary surface that is connected to secondary surfaces (a status fact containing a non-empty list), is selected. The name and group information belonging to the first element of this group is retrieved. Using this information, the program selects the corresponding *sib* fact, and sends the list (*SibList*), contained in this fact, to (11). Rules (278-281 p A-24).

(11) The program then attempts to determine whether any of boundary curves contained in *SibList*, belong to any of the newly defined free surfaces. This situation, where curves belonging to a *sib* set already form part of the newly created free surfaces, would occur if the secondary component was oriented in such a way that a *sib* set intersected with any of the primary boundary curves. If the curves from

SibList are determined not to belong to any free surfaces, then the program proceeds to the following step, otherwise, a new *sib* set is selected and this step is repeated. Rules (282,283 p A-24).

(12) This step defines the outer boundaries of these new free surfaces according to the curve selection strategy defined in (4). Rule (285 p A-24).

At this stage in the program execution the outer boundaries of the new free surfaces have been defined. The final procedure in the definition of the free surfaces is to determine whether any inner boundaries lie within these new outer boundaries. Due to the fact that the latter free surfaces were created from *sib* curves, the only allowable possibility for the existence of inner boundaries within these free surfaces restricts the type of boundary curves to those contained in the *pib* set.

(13) The program attempts to select a *pib* fact where the curves belonging the list *PList*, contained in this fact, have not already been defined as part of a free surface. It then sends the appropriate information to (14). Rules (286-290 p A-24).

(14) The first curve of *PList* is selected, and using the *counting-intersection method*, the programs determines whether this curve lies within the outer boundaries of the new free surfaces, and depending on the outcome of this investigation, the program either adds the *pib* curves to the free surfaces, or selects another *pib* set. This process is continued until all *pib* sets have been attempted. Rules (291-293 p A-24).

The final result of the above methodology is the definition of all the possible free surfaces. The final step, with regard to the geometry creation and contact surface definition, is to define the areas of contact on the primary component.

7.4.8 DEFINITION OF THE CONTACT SURFACES

Contact surfaces as the name suggests are the surfaces of the primary component which are in direct contact with surfaces belonging to the secondary components. These are the surfaces to which the boundary conditions, inherent in the type of connection existing between the adjacent components, are applied. In a similar fashion to the creation of the free surfaces, the contact surfaces are defined using a modified *curve selection strategy*.

As before, the creation of the contact surfaces begins with the definition of the outer boundaries originating from the *pob* set.

(1) The execution commences by selecting a *status* fact containing a non-empty list, namely *SList*, of secondary boundary curves. A *pob* fact containing the same name argument as the *status* fact is then selected, and the information represented by these two facts is sent to (2). Rule (294 p A-25).

(2) From the list of boundary curves contained in the pob fact, the first curve, *Curve*, is selected. Using the *curvestatus* fact, the program determines whether *Curve* is an *fc* curve. If Curve proves not to be an *fc* curve, then the next curve from the list is selected and step (2) is repeated until either an *fc* curve is discovered, or until the list is empty, whereupon the program proceeds to (3). If an *fc* curve is discovered, this curve is then deleted from the appropriate sets and a fact defined by the functor *consurf* is created.

consurf($Name_1$, Group, $Name_2$, CList)

CList originally contains only one element, namely *Curve*, as follows: *[Curve|[]]*. The remainder of the curves defining this contact surface will be added in the subsequent steps. A *templine* fact is then created containing the end points of *Curve*. Rules (295,296 p A-25).

(3) This step is virtually the same as (2) except that a list of *pob* curves is searched for the occurrence of *shared* curves instead of *fc* curves. Also instead of proceeding to (3), the program goes directly to step (4). Rules (297,298 p A-25).

(4) This step contains the *curve selection strategy* for contact surfaces. When defining contact surfaces, the program always selects curves in the following order: *fc, bc, shared*. Rule (302 p A-25).

(5) This step is responsible for selecting the appropriate curves according to the curve selection strategy of (4). From *templine(Curve, P1, P2)*, the program retrieves information about the starting end point *P1* and the current end point *P2*. The program then investigates the database for a new curve, namely *NCurve*, having *P2* as one of its end points. If such a curve is found, and it is not already a member of the list contained in the *consurf* fact, then the other end point *P3* of this curve, is compared to *P1*. If *P3* is equal to *P1*, then a closed boundary containing a free

surface has been defined. If the end points are not equal, the *templine* fact is replaced by a new *templine* fact having the end point *P3* as its third argument (the current end point). The program then goes back to (4). Rules (303-306 p A-25).

(6) Upon completion of checking for *fc* and *shared* curves from the pob set, the program then attempts to find a *bc* curve from the *sob* set. To avoid unnecessary repetition, this step will not be described in detail since it follows the same procedure as (3), except in this case the program is searching for a *bc* curve from the list of *sob* curves. If no *bc* curves are available, the program then attempts to find a *shared* curve from the *sob* curves. Rules (307-310 p A-25).

At this stage all the outer boundaries defining the contact surfaces have been defined and the final task is to once again investigate these newly defined surfaces for the presence of inner boundaries. The search for inner boundaries is appropriately restricted to the *pib* and *sib* sets, as these are the only boundary types that can be contained within the outer boundaries of the contact surfaces.

(7) A non-empty list of secondary surfaces from the *status* fact is selected. From the first element of this list, the name and group information is retrieved, and the *consurf* fact having the same name and group arguments is selected. Rules (313-316 p A-25).

(8) A *pib* set, lying on the same plane as the above contact surface, is then selected. The list of inner boundaries contained in this *pib* set is then sent to (9). Rule (317 p A-26).

(9) From this set of inner boundaries the program attempts to find a *fc* curve using the *curvestatus* fact. If such a curve is discovered, the curve is added to the *consurf* fact and deleted from the appropriate sets. The program then goes back to (4) to define the remainder of the inner boundaries. If no such curve is found, the program will then recheck the same list for the occurrence of a *shared* curve. Depending on the outcome of the search, the program will either go back to (4) to define the rest of the inner boundaries, or proceed to (10) and attempt to find a *sib* set within the contact surfaces. Rules (318-321 p A-26).

(10) The first *sib* fact is selected and the first curve from the list contained in this fact is retrieved. Using the *curvestatus* fact, the program checks whether this curve is a *bc* curve. If not, the next curve in the list is selected. This process is continued until

a *bc* curve is detected, or until the list is empty. If a *bc* curve is discovered, the program then goes back to (4) to define the remainder of the inner boundary curves. Rules (322-323 p A-26).

On the completion of this last step, the final geometry describing the primary component has been defined. Using only the information regarding the physical relations between adjacent components and the face set information defining the component's shape, the program is able to successfully identify or recognise the surfaces between the primary and secondary components that are in contact. Subsequently, through the utilization of the appropriate facts containing information about the type of boundary condition existing between the adjacent components, the program applies the necessary boundary conditions to the appropriate contact surfaces. The application of the required loads and boundary conditions are described in sections 7.5.3 and 7.5.4.

7.4.9 TWO-DIMENSIONAL PLATE EXAMPLE

In an attempt to clarify any uncertainties that may exist regarding the recognition of contact surfaces, the following example has been included. To avoid the unnecessary confusion that may be associated with a three-dimensional model, a relatively simple plate model, first illustrated in Figure 7.4.1, has been used. Although the model is contained in one plane only, therefore allowing a more lucid representation, the same principles apply to more complex three-dimensional geometry.

In an attempt to provide a succinct example, the initial stages of geometry definition for both the primary and secondary components have been omitted and the example commences from a point where the boundary representation has been determined. For the duration of this example, curves will be represented by integers. Figure 7.4.9.1 shows a labelled boundary curve representation of the primary and secondary components. The graphical representation of the two plates illustrated in Figure 7.4.9.1 is stored in the external database as two separate facts, namely *curvelist* and *scurvelist*, representing the primary and secondary components respectively.

curvelist("s1", [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16])

scurvelist("s2", [17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32])

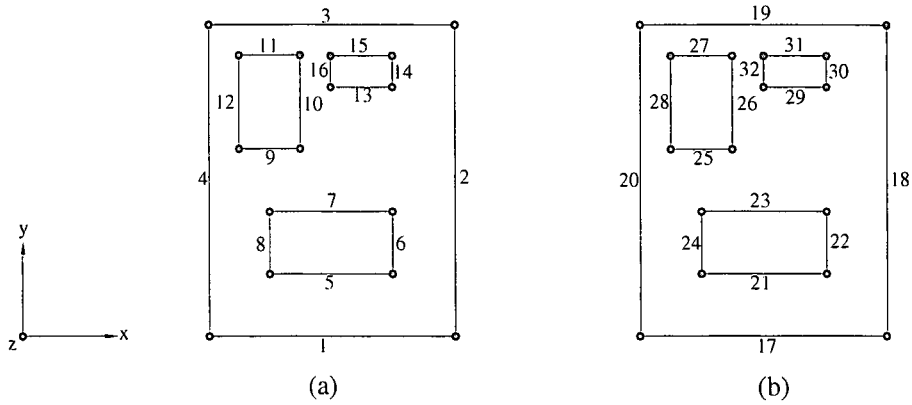


Figure 7.4.9.1. (a) Primary component. (b) Secondary component.

The next step is to orientate the secondary geometry relative to the primary component according to the information specified in the database. For the purpose of this example, it will be assumed that the secondary component will be orientated as depicted in Figure 7.4.1. The following fact defined by the functor *connected* contains the relevant translational and rotational information. It also includes the type of constraint existing between the primary and secondary component, which was arbitrarily assumed to be *fixed.*

connected(1, 4.5, 0.5, 0, 0, 0, 90, fixed)

Once the secondary component has been correctly orientated, the program then needs to group the secondary boundary surfaces according to their relation to the primary boundary surfaces. Because only one primary and secondary boundary surface exists, and both these surfaces are co-planar, the following fact, defined by the functor *status*, is created with only a single element in its list.

status("s1", [pair("s2", "plate1")|[]], pl(0, 0, 1, 0))

The program then begins the process of grouping the primary and secondary boundary curves into sets of closed boundary curves. This information is contained in the facts

pblist and *sblist*, representing the primary boundaries and secondary boundaries respectively.

pblist("s1", [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13, 14, 15, 16]])
sblist("s2", [[17, 18, 19, 20], [21, 22, 23, 24], [25, 26, 27, 28], [29, 30, 31, 32]])

From these sets of closed boundaries, the program determines which are the outer and inner boundaries, and subsequently creates the following facts to store this information:

pob("s1", [1, 2, 3, 4])
pib("s1", "p1", [5, 6, 7, 8])
pib("s1", "p2", [9, 10, 11, 12])
pib("s1", "p3", [13, 14, 15, 16])
And
sob("s2", "plate1", [17, 18, 19, 20])
sib("s2", "plate1", "s1", [21, 22, 23, 24])
sib("s2", "plate1", "s2", [25, 26, 27, 28])
sib("s2", "plate1", "s3", [29, 30, 31, 32])

The next step is to determine the intersections between the primary and secondary curves, and consequently define the new curve segments originating from the points of intersection. Figure 7.4.9.2 shows the intersections and the updated curve segments. The following statements define the original curves and the corresponding curve segments, replacing the original curves, which result from the intersections:

Original curve (2) - replacement curves (33, 34, 35)
Original curve (4) - replacement curves (36, 37, 27, 38, 31, 39, 40)
Original curve (5) - replacement curves (41, 42)
Original curve (6) - replacement curves (43, 44)
Original curve (7) - replacement curves (45, 46, 47)
Original curve (12) - replacement curves (48, 49)
Original curve (18) - replacement curves (50, 51, 11, 52, 15, 53, 54)
Original curve (20) - replacement curves (55, 56, 57)
Original curve (23) - replacement curves (58, 59)

Original curve (24) - replacement curves (60,61)

Original curve (26) - replacement curves (63, 47)

Original curve (28) - replacement curves (62, 41)

Original curve (29) - replacement curves (49, 64)

Once the primary-secondary curve intersections have been determined, the program then establishes whether any secondary-secondary curve intersections exist. However, because in this example there is only one secondary component, this step is omitted. The original outer and inner boundary facts are then updated by removing the original curve, and replacing this curve by the recently created curve segments, as shown by the following facts:

pob("s1", [1, 33, 34, 35, 3, 36, 37, 27, 38, 31, 39, 40])

pib("s1", "p1", [41, 42, 43, 44, 45, 46, 47, 8])

pib("s1", "p2", [9, 10, 11, 48, 49])

pib("s1", "p3", [13, 14, 15, 16])

And

sob("s2", "plate1", [17, 50, 51, 11, 52, 15, 53, 54, 19, 55, 56, 57])

sib("s2", "plate1", "s1", [21, 22, 58, 59, 60, 61])

sib("s2", "plate1", "s2", [25, 63, 47, 27, 62, 41])

sib("s2", "plate1", "s3", [49, 64, 30, 31, 32])

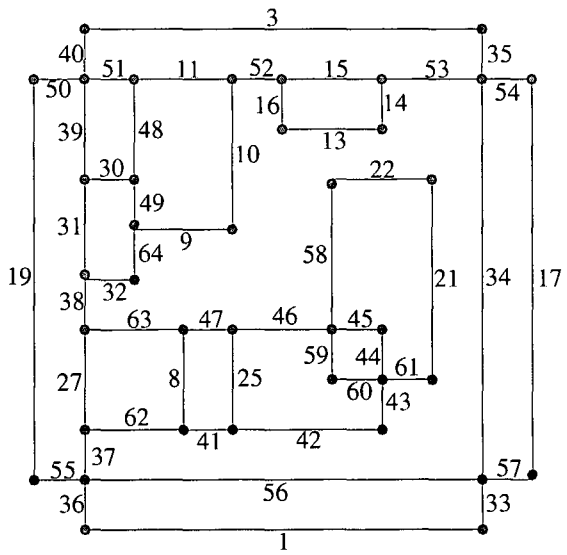


Figure 7.4.9.2. Primary and secondary curve intersections.

The secondary curves which do not lie within the primary boundary surface, and the secondary curves which are common with the primary sets are then removed from the appropriate secondary sets, resulting in the substitution of following facts:

sob("s2", "plate1", [51, 52, 53, 56])
sib("s2", "plate1", "s1", [21, 22, 58, 61])
sib("s2", "plate1", "s2", [63, 62])
sib("s2", "plate1", "s3", [64, 30, 32])

The remaining curves are then labelled according to their relation to the secondary surface. The following facts are used to represent this:

curvestatus(1, nc, "s1")
curvestatus(3, nc, "s1")
curvestatus(8, nc, "s1")
curvestatus(9, fc("s2", "plate1"), "s1")
curvestatus(43, fc("s2", "plate1"), "s1")
curvestatus(44, nc, "s1")
curvestatus(45, nc, "s1")
curvestatus(46, fc("s2", "plate1"), "s1")
curvestatus(47, shared, "s1")
curvestatus(48, fc("s2", "plate1"), "s1")
curvestatus(49, shared, "s1")
curvestatus(51, bc("s2", "plate1"), "s1")
curvestatus(52, bc("s2", "plate1"), "s1")
curvestatus(53, bc("s2", "plate1"), "s1")
curvestatus(56, bc("s2", "plate1"), "s1")
curvestatus(58, bc("s2", "plate1"), "s1")
curvestatus(61, bc("s2", "plate1"), "s1")
curvestatus(10, fc("s2", "plate1"), "s1")
curvestatus(11, shared, "s1")
curvestatus(13, fc("s2", "plate1"), "s1")
curvestatus(14, fc("s2", "plate1"), "s1")
curvestatus(15, shared, "s1")

curvestatus(16, fc("s2", "plate1"), "s1")
curvestatus(21, bc("s2", "plate1"), "s1")
curvestatus(22, bc("s2", "plate1"), "s1")
curvestatus(27, shared, "s1")
curvestatus(30, bc("s2", "plate1"), "s1")
curvestatus(31, shared, "s1")
curvestatus(32, bc("s2", "plate1"), "s1")
curvestatus(33, nc, "s1")
curvestatus(34, fc("s2", "plate1"), "s1")
curvestatus(35, nc, "s1")
curvestatus(36, nc, "s1")
curvestatus(37, fc("s2", "plate1"), "s1")
curvestatus(38, fc("s2", "plate1"), "s1")
curvestatus(39, fc("s2", "plate1"), "s1")
curvestatus(40, nc, "s1")
curvestatus(41, shared, "s1")
curvestatus(42, fc("s2", "plate1"), "s1")
curvestatus(62, bc("s2", "plate1"), "s1")
curvestatus(63, bc("s2", "plate1"), "s1")
curvestatus(64, bc("s2", "plate1"), "s1")

The last grouping procedure to be carried out before the definition of the free and contact surfaces involves creating two new facts containing pairs or duplicates of secondary curves, as demonstrated by the following facts:

tempsob("s2", [51, 51, 52, 52, 53, 53, 56, 56])
tempsib("s2", [21, 21, 22, 22, 58, 58, 61, 61, 63, 63, 62, 62, 64, 64, 30, 30, 32, 32])

These two facts along with the *pob* and *pib* facts are the final facts used to create the free and contact surfaces. As mentioned previously, the curves are selected according to the appropriate *curve selection strategy*. As each curve is selected from one of the above facts, it is then removed from this fact, and the curve selection process continues until all curves have been removed (until all facts contain empty lists). Figure 7.4.9.3

illustrates the final curves used to discretise the original primary surface. The program initiates the representation of the new boundary surfaces by initially defining the areas which are free. The first *nc* curve, *curve 1*, belonging to the *pob* fact is selected. Following the *curve selection strategy* for free surfaces (*nc, bc, shared*), the program attempts to find a suitable curve attached to one of the end points of *curve 1*.

For the purpose of this example, assume that the program proceeds from the end point on the right hand side of the curve. The search results in *curve 33* being selected. The program then attempts to find the curves attached to the opposite end point of *curve 33*. In this case two curves, namely *curve 34* and *curve 56*, are detected. However *curve 34*, being an *fc* curve, is immediately removed from the selection leaving only *curve 56*, a *bc* curve, which is therefore appropriately selected. Once again the program arrives at a juncture where a choice needs to be made between two curves, *curve 36* and *curve 37*, attached to the opposite end point of *curve 56*. *Curve 36*, being an *nc* curve, is selected as apposed to *curve 37*, which is an *fc* curve and therefore cannot be selected in the definition of free surfaces. Once *curve 36* is selected, a closed boundary is formed defining the first free surface. The following fact, defining this free surface, is added to the external database:

freesurf("s1", "f1", [1, 33, 56, 36])

The program will then attempt to define other free surfaces by searching for another *nc* curve belonging to the *pob* fact. The result of this searching will be the selection of *curve 3*, the next *nc* curve in the *pob* fact. *Curve 35* will then be selected, and once again the program will arrive at a juncture where a choice needs to be made between *curve 53* and *curve 34*. However *curve 34*, being an *fc* curve, is removed from the selection leaving only *curve 53* to be selected. The remaining curves, defining this free surface, are then selected as follows: *curve 15, curve 52, curve 11, curve 51* and *curve 40*. Once *curve 40* is selected, a closed boundary is formed defining the next free surface.

freesurf("s1", "f2", [3, 35, 53, 15, 52, 11, 51, 40])

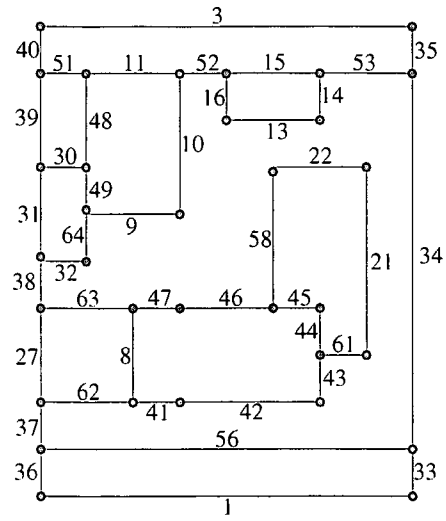


Figure 7.4.9.3. The final curves for the selection process.

At this stage there are no more *nc* curves belonging to the *pob* fact, and therefore a search commences for a *shared* curve in the pob fact. *Curve 27* is the first *shared* curve to be selected. The program then selects remaining curves in this order: *curve 62, curve 8* and finally *curve 63*, which subsequently forms the closed boundary defining the free surface represented by the following fact:

freesurf("s1", "f3", [27, 62, 8, 63])

The program then attempts to select another *shared* curve from the *pob* fact, resulting in the selection of *curve 31*. The remaining curves, defining this free surface, are selected in the following order: *curve 32, curve 64, curve 49* and finally *curve 30* forming the closed boundary, represented by the following fact:

freesurf("s1", "f4", [31, 32, 64, 49, 30])

At this stage, all *shared* and *nc* curves belonging to the *pob* fact have been selected. The next step is to investigate these newly created free surfaces for the presence of inner boundary curves. From Figure 7.4.9.3, it is evident that no inner boundaries exist inside the free surfaces, and therefore the program proceeds to the following step involving the definition of free surfaces inside the secondary inner boundaries. The program automatically detects that two of the three *sibs* have already been used in creation of

free surfaces, and therefore these *sibs* are excluded from the selection to avoid overlapping or multiple surfaces.

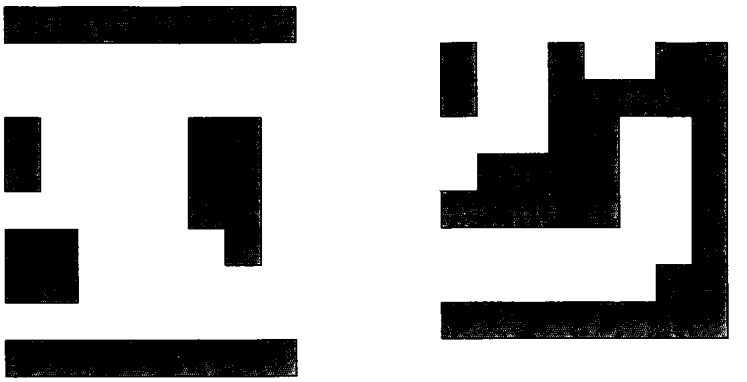
The first *bc* curve, *curve 21*, from the remaining *sib* fact is selected. Once again, using the *curve selection strategy*, the remaining curves, defining this free surface, are selected in the following order: *curve 22, curve 58, curve 45, curve 44* and *curve 61*. Once *curve 61* has been selected, a closed boundary forming the final free surface is defined. The following fact represents this surface:

freesurf("s1", "f5", [21, 22, 58, 45, 44, 61])

After the definition of this last free surface, the program searches for the occurrence of inner boundaries contained within the new outer boundary. However, it is clearly visible from the figure that no inner boundaries are present. Figure 7.4.9.4 (a) shows the graphical representation of the free surfaces. The next step is to begin the creation of the contact surfaces by attempting to select an *fc* curve from the *pob* set. The first *fc* curve to be selected is *curve 37*. The remainder of the curves are selected according to a slightly modified *curve selection strategy* (*fc, bc, shared*). Following this strategy, the next curve to be selected is *curve 56*, since the other curve, *curve 36*, is an *nc* curve and therefore cannot be selected when defining contact surfaces. The remaining curves defining this contact surface are then selected according to the order specified by the *curve selection strategy* for contact surfaces.

The following curves are selected: *curve 34, curve 53, curve 14, curve 13, curve 16, curve 52, curve 10, curve 9, curve 64, curve 32, curve 38, curve 63, curve 47, curve 46, curve 58, curve 22, curve 21, curve 61, curve 43, curve 42, curve 41* and finally *curve 62*. The first contact surface is defined when *curve 62* is selected forming a closed boundary. The following fact represents this contact surface:

consurf("s2", "c1", [37, 56, 34, 53, 14, 13, 16, 52, 10, 9, 64, 32, 38, 63, 47, 46, 58, 22, 21, 61, 43, 42, 41, 62])



(a) (b)

Figure 7.4.9.4. (a) The five created free surfaces. (b) The corresponding contact surfaces.

The program then searches for the next *fc* curve in the *pob* set and consequently selects *curve 39. Curve 30, curve 48* and *curve 51* are then selected according the order specified by the *curve selection* strategy. Once *curve 51* has been selected, a closed boundary is formed defining the contact surface, as represented by the following fact:

consurf("s2", "c2", [39, 30, 48, 51])

The next step is to investigate these new contact surfaces for the presence of inner boundaries, which according to the figure, clearly do not exist. At this stage all the specified curves have been selected and removed from the appropriate facts, and therefore the definition of the free and contact surfaces is complete. The contact surfaces can be seen in Figure 7.4.9.4 (b).

7.5 THE SCRIPT GENERATION

This section of the program has the very important function converting the relevant information contained in *Prolog's* temporary external database into a script (text file) which can be directly interpreted by MSCN4W, a *Visual Basic* type programming language used to perform direct manipulations on the MSCN4W *database*. Once this script has been interpreted, the specified FE model is automatically generated and

subsequently analysed. Section 6.2 was devoted to describing the physical aspects of the script, and therefore a description of the structure will not be repeated. However, there are other important aspects or functions of this section that need to be appropriately discussed. It should be noted that the program code in this section, and the rest of the sections for that matter, was designed to allow for the representation of solid and surface geometry having no restrictions on their shape or size. In other words all geometry, with the exception of models composed only of curves, regardless of their shape, size and complexity can be conveniently represented by the program. In terms of the physical layout, this section has been divided into the following subsections:

(1) Coordinate system Creation.
(2) Mesh Generation.
(3) Load Application.
(5) Constraint Application.

7.5.1 COORDINATE SYSTEM CREATION

The purpose of this section is to allow for the creation of the necessary coordinate systems resulting from the application of rotational and translational boundary conditions to the FE model. (Constraints are ultimately applied to the nodes of a FE model. If an unconstrained node could be isolated and removed from the model, this node having no physical limitations on its movement would be allowed to move freely in all three dimensions of space. These allowable movements are referred to as the *degrees of freedom* of the node. When considering three-dimensional models, all nodes are assumed to have six *degrees of freedom*. Stated differently, the nodes are allowed to translate and rotate about each of the three specified axes. The necessary constraints are applied to the model by suppressing the corresponding *degrees of freedom* associated with that constraint.) In the case of a translational constraint, the default *rectangular coordinate system* could be used to define the constraint, provided the axis of translation coincided with one of the three orthogonal axes. However, when considering a rotational constraint applied to a model, in order to allow for the necessary rotation about the specified axis, a *cylindrical coordinate system* needs to be created with its z-

axis parallel and colinear to the axis of rotation. Therefore to standardise a procedure for creating coordinate systems, whenever a translational or rotational constraint is applied to a model, irrespective of the orientation of that axis of translation or rotation, a new coordinate system is created. It should be noted that the application of a fixed constraint to a FE model does not require the creation of a new coordinate system as the constraint would still be fixed irrespective of the orientation of that coordinate system.

One of the problems associated with the creation of the specified coordinate systems was the fact that the scripting language did not contain the inbuilt functions that would allow the creation of non-rectangular coordinate systems. Therefore an alternate method needed to be devised that would allow this problem to be bypassed. This method involved the use of MSCN4W *Program Files*, which are essentially scripts used to execute MSCN4W commands to automatically create or modify models. The logical reasoning behind the creation of the appropriate coordinate systems was programmed according to the following methodology:

(1) The program retrieves the list of components, adjacent to the primary component, from the *connected* fact. If the list is empty, a blank file is automatically created. Otherwise the first element of the list is examined for the presence of a rotational or translational constraint type. If either one of these constraint types is detected, the program will proceed to (2), otherwise the next element in the list is investigated. Rules (1,2 p A-26).

(2) Using the value obtained from the *big* fact, the program arbitrarily multiplies this number by 3, 3.2 and 3.5 to obtain the *x, y* and *z* coordinates, of a point in space, respectively. The purpose of this point is to provide the location of the third point required to orientate the coordinate system as specified. The exact location of this point is not important, and hence the arbitrary multiplication of the value above. The reason for this is that the first two points required to specify the orientation of the axis of translation or rotation are contained within the *compound data object*, *Type*, which is the last argument in the *connected* fact. *Type* can assume one of the following forms:

fixed

trans(X_1, Y_1, Z_1, X_2, Y_2, Z_2)

rot(X_1, Y_1, Z_1, X_2, Y_2, Z_2)

Where: *X_1, Y_1 and Z_1* – refer to the *x, y* and *z* coordinates of the first point on a line defining the axis of rotation or translation.

X_2, Y_2 and Z_2 - refer to the *x, y* and *z* coordinates of the second point on a line defining the axis of rotation or translation.

The coordinate information specified inside the above compound data objects is used to orientate, say for example the *z-axis*, of the *rectangular* or *cylindrical coordinate system.* This axis then serves as either the axis of translation using the rectangular coordinate system or the axis of rotation when using a cylindrical coordinate system. The third point is only needed to specify the direction of one of the two remaining orthogonal axes, and therefore its exact location is unimportant as the program is only concerned with specifying the axis along which translation or rotation will occur. Rule (3 p A-26).

(3) The next step involves identifying the type of constraint represented by the argument *Type*, and then creating the required coordinate system. As mentioned before, if *Type* is fixed, no coordinate system is created, if *Type* is rotational, a new *cylindrical coordinate system* is created, and if *Type* is translational, a new *rectangular coordinate system* is created. Rules (4-9 p 26).

Once the required coordinate systems have been created and labelled appropriately, the program then opens a text file and writes the necessary information to this file in the specified format. This file contains information about the type, number and physical location of the required coordinate systems. The following insert is a typical example of the file format used for creating coordinate systems. In this particular example, the first three lines are responsible for creating a *cylindrical coordinate system* and the following three lines for creating a *rectangular coordinate system.*

```
$ Model Coord Sys
{My}<A-Z><@14002><PUSH><OK><A-X>1<A-Y>0<A-Z>0<OK><A-
X>2<A-Y>0<A-Z>0<OK><A-X>8<A-Y>9<A-Z>10<OK><Esc>
$ Model Coord Sys
```

{My}<A-Z><OK><A-X>0<A-Y>0<A-Z>0<OK><A-X>1<A-Y>1<A-Z>1<OK><A-X>8<A-Y>9<A-Z>10<OK><Esc>

When the program is ready to initiate the creation and analysis of the primary component, it sends out a command to the operating system to open an external program, namely MSCN4W. Upon opening, a *Program File,* similar to the one above, automatically executes the necessary commands to create the required coordinate systems. All these actions are completed prior to the opening of the script containing the model and analysis information. Therefore, although the coordinate systems were created separately and by a different means, they can still be referred to directly by the script.

7.5.2 MESH GENERATION

For the purpose of the research performed for this dissertation, it was initially decided that the powerful automatic meshing facility, provided by MSCN4W, would be fully exploited when developing the FE models. With regards to the automatic meshing of the primary component, only two forms of information, concerning the element size and type, need to be supplied to the MSCN4W database. The type of element is automatically determined by the program after establishing whether the component should be represented by a surface or solid model (permissible element types are discussed in section 4.5.1). The second form of information required by MSCN4W, regarding the automatic meshing of the component, concerns the size of the elements or the corresponding mesh density. In this case, the user has direct control of the final element density by specifying a *mesh refinement factor* (*mrf*), which is used as follows. The program initially calculates the average length of all the boundary curves defining the component. Once this value is determined, it is then divided by twice the value of the *mrf*. This value is then used as the average element size when meshing the model. The reason the *mrf* is multiplied by a factor of two is that through experimentation, it was discovered that this value resulted in a suitable range of element sizes when the *mrf* was varied between one and five. Rules (58-62 p A-30) are used to define the necessary meshing information.

7.5.3 LOAD APPLICATION

In order for an integrated system to accurately represent the varied conditions encountered in reality, the designer should have the ability to define any type of load that he / she assumes necessary. Therefore a general approach is required to be adopted for the specification of loads. This generality is achieved through the implementation of the following fact:

load2(ID, Num, Type, L_x, L_y, L_z, X, Y, Z)

This fact is essentially the same as the *load* fact described in section 6.3, except for the *ID* and *Num* arguments. *ID* is responsible for defining the primary component one which the specified load acts and *Num* defines the particular load number acting on the component represented by *ID*. This allows individual loads to be identified when more than one load is acting on a certain component. As described previously, the argument *Type* defines on which entity (point, line or surface) the load acts. The arguments capitalised by '*L*', represent the load magnitude along the three orthogonal axes. The final three arguments represent the *x, y* and *z* coordinates of a specific location used to identify the required entity (point, line or surface). Once the entity has been identified, it is on this entity that the load will be applied.

When *Prolog* is writing the script for the primary component, in other words during the program execution, it has no prior knowledge of the node numbering or locations, as the model has not yet been created. Therefore there is no direct manner in which the appropriate entity can be identified and subsequently have the corresponding loads applied prior to the script execution in MSCN4W. Consequently an alternate method of load application needed to be developed. The method works as follows:

(1) The program retrieves the coordinates of the location where the load will be applied from the *load2* fact. This information is then written in the script in the form of the following variables:

coordl2.x – representing the x coordinate

coordl2.y – representing the y coordinate

coordl2.z – representing the z coordinate

(2) The program then writes the following "looping" structure which performs the task of identifying the closest node to the location specified in (1):

```
nid = 0
dist = 1000

For k = 1 To num
     pos = esp_CoordOnNode(k, coord)
     dx = coordl2.x - coord.x
     dy = coordl2.y - coord.y
     dz = coordl2.z - coord.z
     d = Sqr(dx*dx + dy*dy + dz*dz)
     If d < dist Then
          dist = d
          nid = k
     End If
Next k
```

Where: *nid* – represents the node ID number, starting from node zero.

dist – is the shortest distance between two locations.

num – is the number of nodes in the FE model.

esp_CoordOnNode – is a built-in function, inherent in the scripting language, used to identify the location of a particular node.

d – represents the distance between the specified location and a node contained in the model.

Once this loop has been executed for all the nodes in the FE model, the shortest distance between the node and the specified location would have been determined and subsequently stored in the *dist* variable. The corresponding node associated with this shortest distance is then stored in the variable *nid.*

(3) The next step is to apply the specified load to the required entity. This is performed as follows for the three different entities:

Point load – the load is applied directly to the node labelled *nid.*

Curve load – the curve to which the node labelled *nid* belongs is identified and the load is then applied to this curve.

Surface load – the surface to which the node labelled *nid* belongs is identified and the specified load is the applied to this surface.

By implementing the above methodology, the designer can specify any load, or combinations of loads, that may be required. Rules (63-67 p A-30) allow for the implementation of this methodology. Provisions have been made to the program, to allow for the inclusion of a useful integrated-function at a later stage. If a component to be analysed forms part of a working mechanical assembly and the designer / engineer requires the stress resulting from normal operating conditions, a useful feature would be the ability to incorporate the results obtained by initially performing a *motion analysis* and subsequently applying these resultant loads to the FE model.

7.5.4 CONSTRAINT APPLICATION

The required constraints or boundary conditions are applied to the FE model in one of three ways:

(1) The designer specifies the location and entity type (point, curve or surface) to which the constraint will be applied. The constraint can be fixed, rotational or translational. This information is contained in the fact defined by the functor *con2*.

con2(ID, Num, $Type_1$, $Type_2$, X, Y, Z)

The fact *con2* is similar to the fact *con*, defined in section 6.3, except that it includes two extra arguments, namely *ID* and *Num*. *ID* represents the primary component to which the constraint applies and *Num* represents the identification number of this constraint being applied to the FE model. As mentioned in section 6.3, $Type_1$ defines the entity (point, curve or surface) to which the constraint applies, and $Type_2$ defines the type (fixed, rotation or translation) of constraint to be applied. The last three arguments represent the *x, y* and *z* coordinates of a specific location used to identify the required entity (point, line or surface). The methodology used to identify the appropriate entity is identical to that used for the *load2* fact described in the previous section. Rules (74-80, p A-32).

(2) The designer specifies the type of constraint and the program automatically determines, using the contact surface recognition capability, the areas of the model to which the constraint is applied. The following fact, created in the contact surface recognition section, defines a surfaces to which the constraint applies:

consurf($Name_1$, Group, $Name_2$, List)

Where: *$Name_1$* – defines the name of one of the original secondary surfaces to which the new segmented surface belongs.
Group – defines the name of the component to which the surface belongs.
$Name_2$ – represents the name of a new contact surface created from the segmentation of a primary surface.
List – contains a list of boundary curves defining the contact surface.

Using the information from the above fact, the program has direct access to the boundary curves defining the contact surface where the constraint is applied. From the above fact, the program retrieves the *Group* argument, and using this argument obtains the type of constraint from the following fact:

group(Group, Type)

Once the constraint type has been identified, the program then associates this particular constraint with the relevant coordinate system created at the start of this section (with the exception of a fixed constraint which utilises the default *rectangular coordinate system*). Rules (68-73 p A-31).

(3) This method is a combination of the above two methods. Rules(68-80 p A-31).

The application of the constraints to the FE model completes the generation of the script. Once the script has been completed, the program sends out a command to the operating system to open MSCN4W. Several difficulties were encountered at this stage due to the program structure of the analysis code. The MSCN4W directory contains an executable application named *FEMAP*, which when opened is almost visually identical to MSCN4W. One of the differences between the two is the ability of *FEMAP* to

automatically open the required script file. This allows the program to automatically generate the FE model without any user intervention. However, once the model has been created, an input file needs to be produced containing the relevant information required to analysis the model. The problem is associated with the format of this input file which does subsequently not allow for the generation of the output file, and therefore the results cannot be obtained. This problem was bypassed by using MSCN4W to create the input file, unfortunately this bypass resulted in the need for user intervention. As apposed to having the model automatically constructed, the user is now required to open the necessary script file manually, although this only requires a couple buttons to be pressed. Once the script has been opened, the remainder of the operations are performed automatically.

7.6 OUTPUT DATA RETRIEVAL

The objective of this section, as the heading suggests, is to obtain the results from a FE analysis. The type of output data that one can obtain from the analysis is extremely varied ranging from translations and rotations to principle and max shear stresses. However, for the purpose of the work presented here, the actual choice of output data is immaterial as the process of changing from one data type to the next is relatively simple. Ultimately though, it was decided that the program would select the resultant displacement and the *Von Mises* stress from a specified location. (The exact location is specified by the designer and is stored in fact defined by the functor *vg*, an acronym for *virtual gauge*, containing the *x, y* and *z* coordinates of the location of the gauge.) The *Von Mises* stress is basically an equivalent or average stress and is defined by the following equation [30] :

$$\sigma e = \frac{1}{\sqrt{2}}\left[(\sigma 1 - \sigma 2)^2 + (\sigma 2 - \sigma 3)^2 + (\sigma 3 - \sigma 1)^2\right]^{1/2} \qquad (7.6.1)$$

Where: σe - defines the Von Mises stress.

$\sigma 1$, $\sigma 2$ *and* $\sigma 3$ - represent the three principal stresses, where $\sigma 1$ is algebraically the largest and $\sigma 3$ the smallest.

As mentioned at the end of the previous section, before an analysis can be performed on the FE model, MSCN4W is required to create an input file, also referred to as a *DAT* file because of the *.DAT* extension used when naming the file. The *DAT* file contains the following information:

(1) The type of analysis (*linear static, non-linear, buckling* etc) to be performed.
(2) The models geometry in terms of the node locations.
(3) A collection of finite elements and their corresponding nodes.
(4) The applied loads.
(5) The applied constraints.
(6) Requests for the type of output quantities to be calculated.

The program initially attempts to calculate the resultant displacement at the location specified by the *virtual gauge*. In a FE model, the results for displacements and stresses are obtained directly from the nodes of that model. As stated before, the information about the nodes and their locations is defined in the *DAT* file. Because the placement of the virtual gauge is not restricted to coincide with the node locations, the program employs two different methods for identifying the node, or nodes, associated with the *virtual gauge*:

(1) If the *virtual gauge* lies within a predetermined allowable distance from a node, for argument sake one millimetre, then the program will obtain the identification number of that node from the DAT file.
(2) If the gauge lies outside the predetermined allowable distance, the program then identifies the four closest nodes and calculates the distance between the gauge and the node for each of the four nodes.

Once the analysis has been performed using the information from the *DAT* file, an output file, also referred to as a *F06* file because of the *.F06* extension on the filename, is created. The *F06* file contains information regarding the displacements, resultant forces and various stresses on each of the nodes of the FE model. In case (1) above, to obtain the displacement of the specified node, the program simply uses *string parsing techniques* to identify the node in the *F06* file, and then retrieve the required information concerning the displacement. Case (2) is slightly more complicated and

requires the program to obtain the displacements for the four specified nodes. Using this information and the distances calculated between the gauge and the nodes, the program interpolates an average displacement for the *virtual gauge*. Rules (4-24 p A-33).

The next step is to calculate the *Von Mises* stress for the displacement gauge. The methodology used can be summed up as follows:

(1) If the *virtual gauge* is coincident with a node, then the program will identify all the adjacent elements attached to this node. Once the necessary elements have been detected and their identification numbers obtained from the *DAT* file, the program proceeds to the *F06* file. Here it searches for the elements, using the identification numbers obtained, and retrieves the value of the *Von Mises* stress for each of the elements. Subsequently, the highest stress value between these elements is determined and saved in the temporary external database. This is the value that is quoted as the *Von Mises* stress at the virtual *gauge*.

(2) If the *virtual gauge* does not lie upon a node, then the program initially determines the two closest nodes to the gauge. Once these nodes have been identified, the two elements attached to these nodes are detected and consequently the distance between the centroid of each element and the virtual gauge is calculated. Finally the stress at the gauge is quoted as being the stress at the element whose centroid is closer to the gauge. This stress is then saved in the temporary external database and quoted as the *Von Mises* stress at the *virtual gauge*.

Rules (25-76 p A-34) are responsible for obtaining the element identification numbers and eventually for determining the *Von Mises* stress on the appropriate element. This section concludes the description of the program structure and the methodology used to initially create the required FE model and finally to obtain the results of the analysis due to the application of the appropriate loads and boundary conditions. Once this process has been completed, the results can be made immediately available to the designer or engineer, or they can be stored in the *Access* database for use at a later stage. One of the most significant advantages of this design methodology is that the entire conceptual design process, from model generation to FE analysis, can be performed in a fraction of the time taken to perform the same task manually.

CHAPTER 8

APPLICATION EXAMPLE

The following example demonstrates the implementation of the integrated system to one of the most commonly encountered and used mechanical assemblies - a piston-crank mechanism. For the purpose of this example, the mechanism has been simplified by omitting some of the "less important" components. Also, to limit the size of the face set data files (defined in Appendix A1), all cylindrical shapes have been replaced by octagonal cylinders as illustrated in Figure 8.2. It should be noted that the components illustrated in Figure 8.2 are not represented to scale, but rather at the maximum size permitted by the specified picture box for each of the individual components. The components are depicted as they would appear in MSCN4W after the meshing process has been performed. A complete representation of the mechanical assembly (depicting the assembled mechanism excluding the cylinder and bush) can be seen illustrated in Figure 8.3. The hierarchical tree-structure showing the inter-component relationships can be seen in Figure 8.1. It is important to remember that the information defined by this hierarchical structure determines the allowable movement, of each of the components, with respect to the specific parent-child relations of that component.

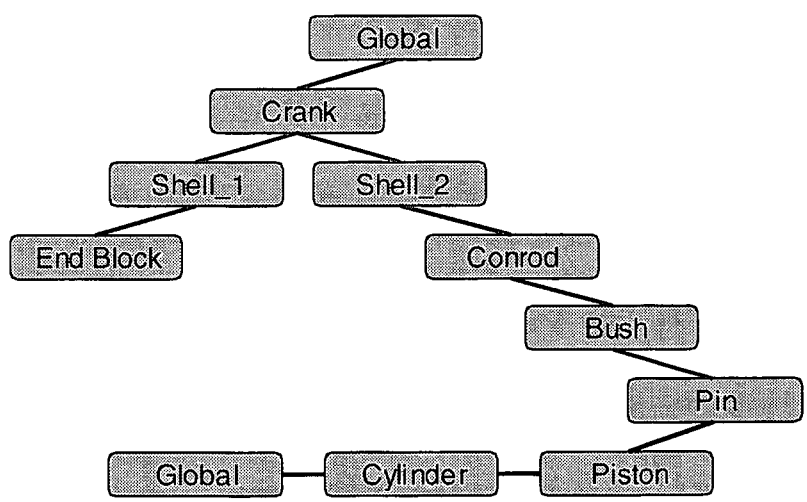
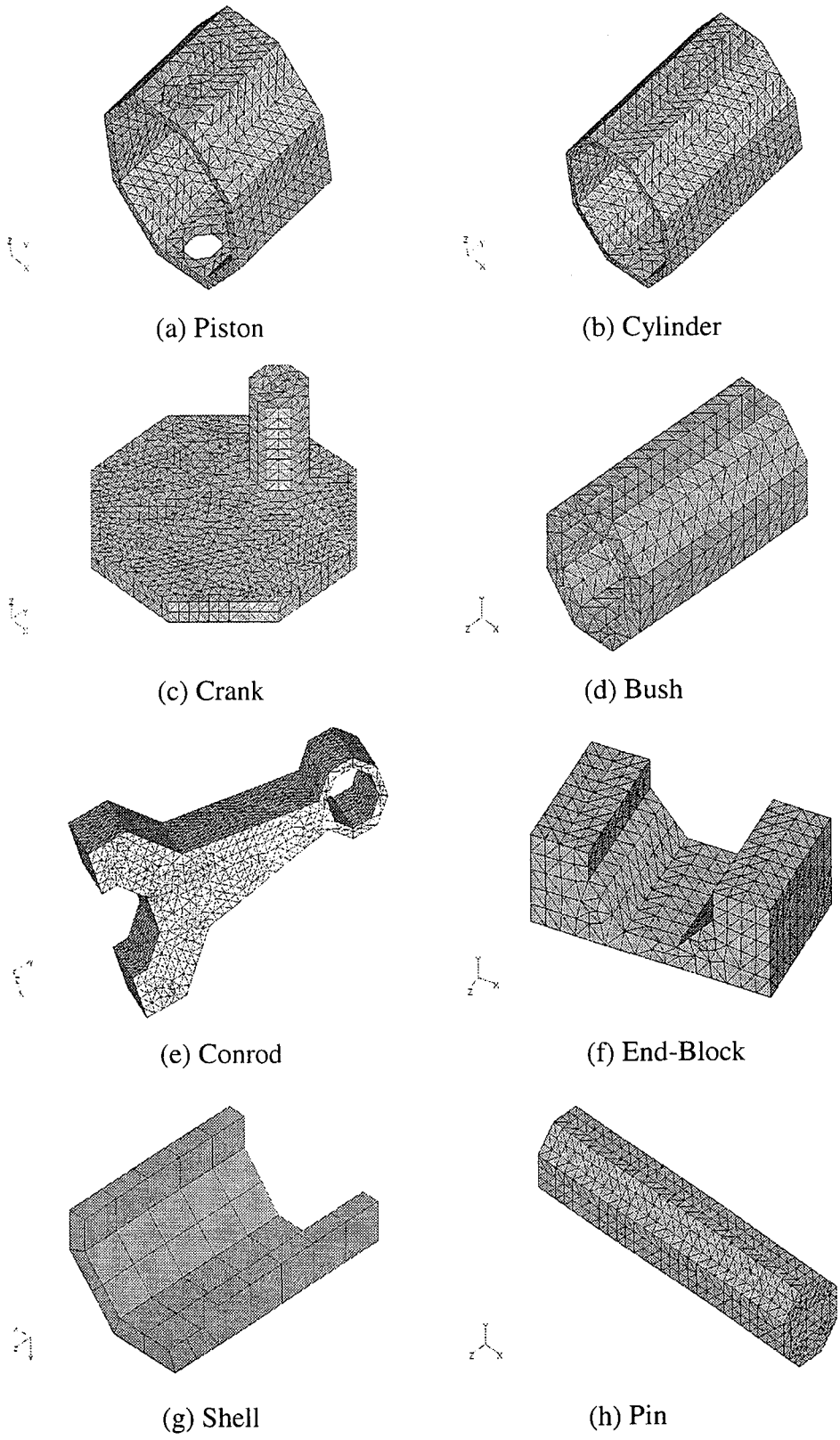


Figure 8.1. Hierarchical tree-structure.



(a) Piston (b) Cylinder

(c) Crank (d) Bush

(e) Conrod (f) End-Block

(g) Shell (h) Pin

Figure 8.2. Components of the piston-crank mechanism

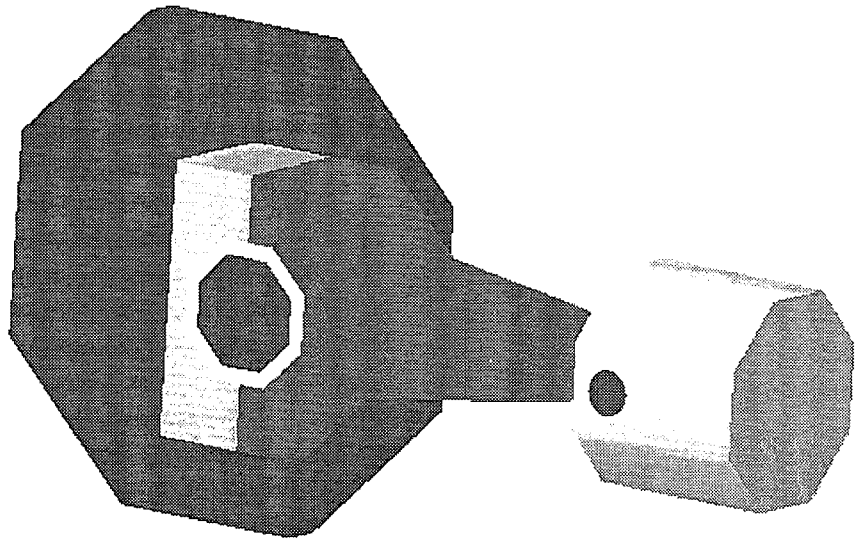


Figure 8.3. The assembled piston-crank mechanism.

Assume for the purpose of this example, that a design company, specialising in the design and manufacture of high performance engines, has recently completed the first batch of prototypes for evaluation and testing. After performing some initial endurance tests, it was established that the piston-crank mechanism was failing during normal operating conditions, and that the information obtained from further on-site testing indicated that the piston was the area of concern (assume that all the relevant design information has been established). The designer is therefore required to determine the exact cause of failure to allow for the problem to be rectified.

The usual procedure for creating and analysing a FE model could be described as follows. The engineer would initially gather the required geometrical information from the engineering drawings. He / she would then consult the information obtained from on-site testing and determine, as realistically as possible, the resultant loads and boundary conditions that are required to be applied to the FE model. At this stage it is important that the solid model is suitably created with provisions made for the application of the appropriate loads and boundary conditions. This is an important step as generally, the loads and boundary conditions are applied directly to the nodes of the FE model. However, more often than not, if the FE model has not been created taking these considerations into account, the nodes will not be situated as required, resulting in the FE model having to be modified accordingly. Once the FE model has been created, the engineer can then apply the loads and boundary conditions and finally analyse the

model. In situations similar to the piston-crank assembly, the engineer may be required to apply multiple load cases and boundary conditions to the FE model. This process (depending on the variations in the loads and boundary conditions) often requires the initial model to be modified accordingly. As a result, the application of the FE method either as a design or a design verification tool, generally imposes time and therefore cost penalties. However, this addition of time can be greatly reduced by using the integrated design system described in this dissertation.

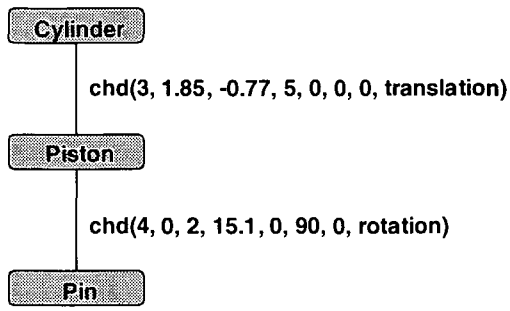


Figure 8.4. Sub-tree defining parent and child components.

Typically, the user of the integrated system would employ the following methodology when performing a finite element analysis on the specified component. The design drawings of the components of the piston-crank assembly would initially be converted into the required face set format. The separate face set data files would then be linked directly to the *Access* database along with the appropriate material properties and parent-child relations (the latter allows for the hierarchical representation of the assembly and consequently, the determination of the relative locations of the individual components). Figure 8.4 illustrates the sub-tree defining the parent-child relations of the piston, in other words the physical associations with the cylinder and pin. The two facts defined by the *functor chd*, illustrated in Figure 8.4, specify the positions of the cylinder and pin relative to the piston and also the type of constraint associated with the connection. The compound data objects, represented by the arguments *translation* and *rotation*, define the following axes of translation (the translation of the piston relative to the cylinder) and rotation (the rotation of the pin relative to the piston) respectively:

trans(16.63, 0, -6.89, 16.63, 50, -6.89)
rot(16.63, 5.7, 9.74, 16.63, 5.7, -23.52)

It should be noted that in order to clearly illustrate the contact surface recognition capability of the program, the piston was positioned so that it protruded from the lower section of the cylinder, as depicted in Figure 8.5. The figure has been colour-coded to distinguish between the contact and free surfaces. The areas represented by dark grey define the contact surfaces between the piston and the cylinder, those represented by black define the contact surfaces between the piston and the pin, and the areas in light grey represent the free surfaces.

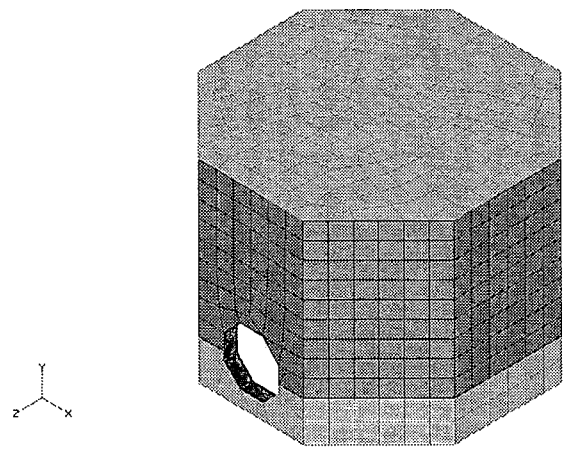


Figure 8.5. The free and contact surfaces on the piston.

By using the *chd* facts, the program is able to automatically determine the free and contact surfaces between the piston and its adjacent components. This function has the potential to significantly reduce the time required to create and analyse a FE model, especially when a number of varying load cases and / or boundary conditions are involved (as described previously in this example).

Before the virtual model can be created and analysed by the integrated system, the designer is required to specify the appropriate loads (in this case the loads that would have been determined from on-site testing) and the location of the virtual gauges. This information is supplied to the database in the following format:

load('s', 0, P, 0, 16.63, 50, -6.89)
and
15, 1.5, 9.74

The fact defined by *load* assumes that a single load of magnitude P is applied vertically to the top surface of the piston. The argument 's' specifies that the load will be applied to a surface. The second and fourth arguments imply that a load only exists along the vertical axis. The last three arguments of the fact define the location of a node used to identify the surface to which the load is applied. Finally, the designer specifies the location of the virtual gauge on the model. Naturally, the gauge should be applied to areas of interest (areas of high stress). Once the relevant information has been defined, the designer simply presses the *"analyse button"* and patiently waits for the finite element model to be created and analysed. After the results have been obtained, the designer has the option of accepting the results, or performing a series of analyses to establish the accuracy of the results (convergence testing).

CHAPTER 9

CONCLUSION

The objective of developing a virtual prototyping system, capable of providing a more efficient conceptual design process, has been achieved through the development of an integrated design - analysis system. The gains in efficiency have been realised predominantly through the implementation of the following two strategies: (1) automating the creation of the finite element analysis model from the design generation information, and (2) automating the transfer of design specific data between the various phases of the design process. Through the utilisation of these two strategies, the design system is able to provide a seamless integration, requiring minimal user interaction, between the design generation and analysis phases of the design process.

The integrated system consists of a design generation tool based on the *Open Inventor* set of graphics libraries, a centralised product data model created using *Microsoft Access* and the commercially available finite element analysis code *MSC Nastran for Windows*. The integration between the product data model and the analysis code is provided by a program developed using *Visual Prolog* (a logic programming language). The program automatically creates a script, using the relevant information obtained from the product data model, which is subsequently channelled to the analysis software, whereupon the specified virtual model is analysed. Once the model has been analysed, the designer is able to query the results for the required stresses and displacements at locations specified by the placement of virtual gauges.

As mentioned in Chapter one, an effective virtual prototyping system must exhibit characteristics such as adaptability and expandability, traits that are inherent in this integrated system due to the utilisation of a centralised product data model. This concept provides expandability by allowing for the addition of analysis codes from the various different engineering disciplines without having to modify the existing communication channels. Another important feature of the centralised product data model is the ability to exchange specific analysis codes for updated releases or even

versions offered by different vendors, simply by modifying the communication channel between the product data model and the relevant analysis code. These features allowed the integrated system to expand or grow according to the requirements of the design environment and the availability of newly developed analysis software.

A hierarchical method was employed to represent the individual components and the inter-component relationships of a mechanical assembly. This method was adopted due to its almost "natural" approach to component representation. In other words, the components are represented as they appear in reality and the physical behaviour of the system is simulated by a series of mathematical expressions defining the relative positions of the individual components. The use of a hierarchical form of representation also allowed for the development of a contact surface detection methodology, used to define the contact surface areas existing between the adjacent components of a mechanical assembly. This methodology facilitated the automation of the application of boundary conditions to the finite element model, a process that further increased the efficiency of the integrated system.

Prolog, a programming language based on first-order logic, was used for the creation of the program controlling the integration between the design and analysis codes. A logic-based programming language was selected primarily for the level of modularity offered by the language assuring both adaptability and expandability. Another beneficial feature of this type of language is its declarative nature, which consequently provides a program that is easier to learn and implement, thereby reducing the potential development time.

An application example of a piston – crank mechanism was included to demonstrate the functionality of the integrated system to a commonly encountered mechanical assembly. The example included a brief comparison of the traditional method of finite element analysis to the method employed by the integrated system.

Although the integrated design system described in this dissertation does not display the required level of sophistication or "intelligence" needed to create complex finite element models (models containing a larger number of elements in areas of high stress and lower mesh densities in non-critical areas), the application of this system to the

conceptual design phase can provide a significant reduction in the overall design time. With regards to recommendations concerning further research and development, the first area that comes to mind would be the development of a *feature-based recognition system*. The purpose of this system would be to recognise certain features (for example, circular shapes) on the virtual model, and consequently the ability to eliminate a large number of straight line segments representing the non-linear features on the model. A direct consequence of this feature would be the ability to provide a more realistic geometric representation of the physical model. Additionally, the size of the solid model created in MSCN4W could be reduced considerably as a single circular surface could be used in may circumstances (depending on the model) to replace a number of flat surfaces representing the specified cylinder. The accuracy of the FE model would then only be a function of the meshing process and the type of element used and would not depend on unnecessary geometric approximations. The user of the integrated system would then have more control over the accuracy of the results by simply modifying the mesh refinement factor in the database.

A possible extension to such a feature-based system would be the ability to associate certain boundary conditions with specific features on a virtual model. The incorporation of this function, into the integrated system, would allow standard components such as bolts, shafts and bearing to have predefined boundary conditions associated with their representation. Consequently, the designer would not have to manually identify and explicitly apply the required boundary conditions to these standard features, as these functions would be performed automatically by the integrated system. As a result, the efficiency of the design process would be further enhanced by again reducing the time required for the user to construct and analyse a FE model.

An area of research that has received considerable attention over the past years and an area that could be improved in terms of its implementation to the integrated system, is the field of automatic mesh generation. The method of mesh generation used for the integrated system considered here, is rather elementary when compared to some of the more sophisticated techniques available today. However, *MSC Nastran for Windows* does exhibit an array of powerful mesh generation algorithms that could be more comprehensively utilised by incorporating additional AI techniques into the integrated system. One such technique that could prove to be beneficial would be the ability of the

integrated system to recognise possible areas of high stress on the virtual model, and subsequently refine the mesh density in those areas. Although there is currently research being performed in similar fields, AI systems displaying this functionality have not yet been incorporated into the integrated system. Naturally the inclusion of such "intelligence" would not only improve the accuracy of the FE solution, but also shorten the analysis time by creating models having fewer elements.

Ultimately, by including intelligent features such as those described above, the gap between true virtual prototyping systems and design systems such as the one presented here would be reduced.

REFERENCES

[1] Cleenwereck, B. (2001). Wireless Lans. *Mobile and Internet Computing and Communications*, July / August 2001: 14-17.

[2] Halpern, M. (1997). Driving Towards Feature-Based Virtual Prototyping. *Computer Graphics World*, September 1997: 23-24.

[3] Rowell, A. A. (1997). Prototyping in a Digital World. *Computer Graphics World*, September 1997: 55-62.

[4] Ohsuga, S. (1989). Toward Intelligent CAD Systems. *Computer – Aided Design.* **21**, 315 – 337.

[5] Meyer, A. (1997). Virtual Prototyping in Practice. *Computer Graphics World*, September 1997: 65-71.

[6] Arabshahi, S., Barton, D. C. and Shaw, N. K. (1993). Steps Towards CAD-FEA Integration. *Engineering with Computers*, **9**, 17 – 26.

[7] Wu, J. K., Wang, J. H. and Feng, C. X. (1995). A Logic-based Mechanical System Constraint Model. *Engineering with Computers*, **11**, 157 – 166.

[8] Peak, R. S., Fulton, R. E., Nishigaki, I. and Okamato, N. (1998). Integrating Engineering Design and Analysis Using a Multi-representation Approach. *Engineering with Computers*, **14**, 93 – 114.

[9] Abdalla, J. A. and Yoon, C. J. (1992). Object-Oriented Finite Element and Graphics Data-Translation Facility. *Journal of Computing in Civil Engineering.* Vol. 6, No. 3, 302 – 322.

[10] Hardell, C. (1996). An Integrated System for Computer Aided Design and Analysis of Multibody Systems. *Engineering with Computers*, **12**, 23 – 33.

[11] Remondini, L., Leon, J. C. and Trompette, P. (1998). High-level Operations Dedicated to the Integration of Mechanical Analysis within a design Process. . *Engineering with Computers*, **14**, 81 – 92.

[12] Gabbert, U. and Wehner, P. (1998). The Product Data Model as a Pool for CAD – FEA Data. *Engineering with Computers*, **14**, 115 – 122.

[13] Mackie, R. I. (1997). Using Objects to Handle Complexity in Finite Element Software. *Engineering with Computers*, **13**, 99 – 111.

[14] Holzhauer, D. and Grosse, I. (1999). Finite Element Analysis using Component Decomposition and Knowledge-Based Control. *Engineering with Computers*, **15**, 315 – 325.

[15] Lakmazaheri, S. and Edwards, P. (1997). Linguistic Approach to 2D Geometric Modelling of Hierarchical Systems. *Journal of Computing in Civil Engineering*, Vol. 11, No. 3, July, 165 – 174.

[16] Shah, J. J. and Tadepalli, R. (1992). Feature Based Assembly Modelling. *Computers in Engineering*, Vol. 1., 253 – 260.

[17] Jonson, D., DeBeer, J. and Daya, N. (2001). The Integration of Computer Aided Design and Analysis Tools using a Logic-Based Approach. *Proceedings of The Sixth International Conference on the Application of Artificial Intelligence to Civil & Structural Engineering.* Vienna, Austria.

[18] Jonson, D.; DeBeer, J. and Daya, N. (2000). Motion Analysis using a Logic-Based modelling approach. *Proceedings of The Second International Conference on Engineering Computational Technology.* Leuven, Belgium.

[19] Vallis, P. and Colton, J. S. (1996). A Method of Object-Orientated Information Management for Layout design. *Engineering with Computers*, **12**, 34 – 45.

[20] Brown, C. E. and O'Leary, D. E. (2000). Introduction to Artificial Intelligence and Expert Systems. http://www.bus.orst.edu/faculty/brownc/es_tutor/es_tutor.htm.

[21] Blount, G. N. and Clarke, S. (1994). Artificial Intelligence and Design Automation Systems. *Journal of Engineering Design*, Vol. 5, Issue 4, 1997, 299 – 213.

[22] Winston, P. H. and Prendergast, K. A. (1984). The AI Business: Commercial Use of Artificial Intelligence. The Mit Press: Cambridge.

[23] Schalkoff, R. J. (1990). *Artificial Intelligence: An Engineering Approach.* McGraw-Hill Book Company, United States of America.

[24] Introduction to the Prolog Language. http://VIP/Html/viptechinfo/introduction.htm.

[25] Tamir, D. E. and Kandel, A. (1995). Logic Programming and the Execution Model of Prolog. *Information Sciences*, **4**, 167 – 191.

[26] Lakmazaheri, S. (1998). Logic-Based 2D Geometric Modelling in a CAD Environment. *Engineering with Computers*, **14**, 123 – 138.

[27] Brakto, I. (2001). Prolog Programming for Artificial Intelligence. Addison-Wesley Publishing Company, London.

[28] Visual Prolog Version 5.2 Language Tutorial (2000).

[29] Zohavi, E. (1992). Finite Element Method for Machine Design. Prentice Hall: Englewood Cliffs.

[30] Cook, D. C. (1995). Finite Element Modelling for Stress Analysis. John Wiley and Sons Inc.: New York.

[31] Courant, R. (1943). Variational Methods for the solution of Problems of Equilibrium and Vibration. *Bull. Am. Math.* Soc. **49**, 1 – 23.

[32] Argyris, J. H. and Kelsey, S. (1960). Energy Theorems and Structural Analysis. Butterworths: London.

[33] Turner, M. J., Clough, R. W., Martin, H. C. and Topp, L. C. (1956). Stiffness and Deflection Analysis of Complex Structures. *Aero. Sci.* 23(9), 805-824.

[34] Reddy, J. H. (1993). An Introduction to the Finite Element Method. (2^{nd} ed.). McGraw Hill: New York.

[35] Manevitz, L. and Givoli, D. (1998). Automating the Finite Element Method: A Test-Bed for Soft Computing. http://citeseer.nj.nec.com

[36] Buchanan, G. R. (1995). Theory and Problems of Finite Element Analysis. McGraw Hill: New York.

[37] Schewchuk, J. R. (1999). Lecture Notes on the Delaunay Mesh Generation. http://citeseer.nj.nec.com

[38] Owen, S. A Survey of Unstructured Mesh Generation Technology. http://www.andrew.cmu.edu/user/sowen/survey.

[39] Mesh Requirements for Finite Elements. http://www.inf.ethz.ch/personal/cetin/thesis/thesis/node20.html.

[40] Yerry, M. A. and Shephard, M. S. (1984). Three-Dimensional Mesh Generation by Modified Octree Technique. *International Journal for Numerical Methods in Engineering*, vol 20, 1965-1990.

[41] Shephard, M. S. and Georges, M. C. (1991). Three-Dimensional Mesh Generation by Finite Octree Technique. International Journal for Numerical Methods in Engineering, vol 32, 709-749.

[42] Lawson, C. L. (1977). Software for C1 Surface Interpolation. Mathematical Software III, 161-194.

[43] Chew, P. L. (1989). Guaranteed-Quality Triangular Meshes. TR 89-983, Department of Computer Science, Cornell University, Ithaca, NY.

[44] Marcum, D. L. and Weatherill, N. P. (1995). Unstructured Grid Generation Using Iterative Point Insertion and Local Reconnection. *AIAA Journal*, vol 33, no. 9, 1619-1625.

[45] Borouchaki, H., Hecht, F., Saltel, E. and George, P. L. (1995). Reasonably Efficient Delaunay Based Mesh Generator in 3 Dimensions. *Proceedings 4th International Meshing Roundtable*, 3-14.

[46] Weatherill, N. P. and Hassan, O. (1994). Efficient Three-dimensional Delaunay Triangulation with Automatic Point Creation and Imposed Boundary Constraints. *International Journal for Numerical Methods in Engineering*, vol 37, 2005-2039.

[47] George, P. L., Hecht, F. and Saltel, E. (1991). Automatic Mesh Generator with Specified Boundary. *Computer Methods in Applied Mechanics and Engineering*, North-Holland, vol 92, 269-288.

[48] Lohner, R., Parikh, P. and Gumbert, C. (1988). Interactive Generation of Unstructured Grid for Three Dimensional Problems. *Numerical Grid Generation in Computational Fluid Mechanics `88*, Pineridge Press, 687-697.

[49] Baehmann, P. L., Wittchen, S. L., Shephard, M. S., Grice, K. R. and Yerry, M. A. (1987). Robust Geometrically-based, Automatic Two-Dimensional Mesh Generation. *International Journal for Numerical Methods in Engineering*, Vol.24, 1043-1078.

[50] Quadros, W. R., Gurumoorthy, B., Ramaswami, K. and Prinz, F. B. (2001). Skeletons for Representation and Reasoning in Engineering Applications. *Engineering with Computers*, **17**, 186 – 198.

[51] Zhu, J. Z., Zienkiewicz, O. C., Hinton, E. and Wu, J. (1991). A New Approach to the Development of Automatic Quadrilateral Mesh Generation. *International Journal for Numerical Methods in Engineering*, Vol. 32, 849-866.

[52] Belongie, S., Malik, J. and Puzicha, J. Shape Context: A new Descriptor for Shape Matching and Object Reconition. http://citeseer.nj.nec.com

[53] Aggarwal, J. K., Ghosh, J., Nair, D. and Taha, I. A Comparative Study of the Three Paradigms for Object Recognition – Bayesian Statistics, Neural Netweorks and Expert Systems.

[54] Bez, H. E., Bricis, A. M. and Ascough, J. (1996). A Collision Detection Method With Applications in CAD Systems for the Apparel Industry. *Computer-Aided Design*, Vol. 28, No. 1, 27 – 32.

[55] Muylle, J. and Topping, B. H. V. (2000).Contact Detection and Enforcement Techniques for the Simulation of Member Structures in Motion. Computational Mechanics: Techniques and Developments, Civil-Comp Press, Edinburgh, 243 – 256.

[56] Wernedce, J. (1994). The Inventor Mentor. (Release 2). Addison-Wesley Publishing Company: Massachusetts.

[57] Damski, J. C. and Gero, J. S. (1996). A Logic–Based Framework for Shape Representation. *Computer – Aided Design*, **28**, 169 – 181.

[58] Chase, S. C. (1997). Logic Based Design Modelling with Shape Algebras. *Automation in Construction*, **6**, 311 – 322.

[59] Chase, S. C. (1996). Design Modelling with Shape Algebras and Formal Logic. *ACADIA '96 Proceedings*, Tucson, AZ.

[60] Chase, S. C. (1996). Using Logic to Specify Shapes and Spatial Relations in Design Grammers. *Workshop Notes, Grammatical Design, Fourth International Conference on Artificial Intelligence in Design.*

[61] Sherman, K. S. and Barcellos, A. (1992). Calculus and Analytical Geometry. (5th ed.). McGraw Hill: New York.

[62] Stroud, K. A. (1996). Further Engineering Mathematics. (3rd ed.). MacMillan Press LTD: London.

APPENDIX

A.1 FACE SET DATA

A1.1 CYLINDER

s(0.0.0.10.82.0.10.82.0.50.0)
s(10.82.0.10.82.10.82.50.10.82.0.50.0)
s(10.82.0.10.82.26.13.0.10.82.10.82.50.10.82)
s(26.13.0.10.82.26.13.50.10.82.10.82.50.10.82)
s(26.13.0.10.82.36.96.0.0.26.13.50.10.82)
s(36.96.0.0.36.96.50.0.26.13.50.10.82)
s(36.96.0.0.36.96.0.-15.31.36.96.50.0)
s(36.96.0.-15.31.36.96.50.-15.31.36.96.50.0)
s(36.96.0.-15.31.26.13.0.-26.13.36.96.50.-15.31)
s(26.13.0.-26.13.26.13.50.-26.13.36.96.50.-15.31)
s(26.13.0.-26.13.10.82.0.-26.13.26.13.50.-26.13)
s(10.82.0.-26.13.10.82.50.-26.13.26.13.50.-26.13)
s(10.82.0.-26.13.0.0.-15.31.10.82.50.-26.13)
s(0.0.-15.31.0.50.-15.31.10.82.50.-26.13)
s(0.0.-15.31.0.0.0.0.50.-15.31)
s(0.0.0.0.50.0.0.50.-15.31)
s(0.50.0.0.50.-15.31.18.50.-7.46)
s(0.50.0.10.82.50.10.82.18.50.-7.46)
s(10.82.50.10.82.26.13.50.10.82.18.50.-7.46)
s(26.13.50.10.82.36.96.50.0.18.50.-7.46)
s(36.96.50.0.36.96.50.-15.31.18.50.-7.46)
s(36.96.50.-15.31.26.13.50.-26.13.18.50.-7.46)
s(26.13.50.-26.13.10.82.50.-26.13.18.50.-7.46)
s(10.82.50.-26.13.0.50.-15.31.18.50.-7.46)
s(0.0.0.10.82.0.10.82.1.85.0.-0.77)
s(10.82.0.10.82.11.59.0.8.97.1.85.0.-0.77)
s(10.82.0.10.82.26.13.0.10.82.11.59.0.8.97)
s(26.13.0.10.82.25.37.0.8.97.11.59.0.8.97)
s(26.13.0.10.82.36.96.0.0.25.37.0.8.97)
s(36.96.0.0.35.11.0.-0.77.25.37.0.8.97)
s(36.96.0.0.36.96.0.-15.31.35.11.0.-0.77)
s(36.96.0.-15.31.35.11.0.-14.55.35.11.0.-0.77)
s(36.96.0.-15.31.26.13.0.-26.13.35.11.0.-14.55)
s(26.13.0.-26.13.25.37.0.-24.29.35.11.0.-14.55)
s(26.13.0.-26.13.10.82.0.-26.13.25.37.0.-24.29)
s(10.82.0.-26.13.11.59.0.-24.29.25.37.0.-24.29)
s(10.82.0.-26.13.0.0.-15.31.11.59.0.-24.29)
s(0.0.-15.31.1.85.0.-14.55.11.59.0.-24.29)
s(0.0.0.0.0.-15.31.1.85.0.-0.77)
s(0.0.-15.31.1.85.0.-14.55.1.85.0.-0.77)
s(1.85.0.-0.77.11.59.0.8.97.1.85.48.-0.77)
s(11.59.0.8.97.11.59.48.8.97.1.85.48.-0.77)
s(11.59.0.8.97.25.37.0.8.97.11.59.48.8.97)
s(25.37.0.8.97.25.37.48.8.97.11.59.48.8.97)
s(25.37.0.8.97.35.11.0.-0.77.25.37.48.8.97)
s(35.11.0.-0.77.35.11.48.-0.77.25.37.48.8.97)
s(35.11.0.-0.77.35.11.0.-14.55.35.11.48.-0.77)
s(35.11.0.-14.55.35.11.48.-14.55.35.11.48.-0.77)
s(35.11.0.-14.55.25.37.0.-24.29.35.11.48.-14.55)
s(25.37.0.-24.29.25.37.48.-24.29.35.11.48.-14.55)
s(25.37.0.-24.29.11.59.0.-24.29.25.37.48.-24.29)
s(11.59.0.-24.29.11.59.48.-24.29.25.37.48.-24.29)
s(11.59.0.-24.29.1.85.0.-14.55.11.59.48.-24.29)
s(1.85.0.-14.55.1.85.48.-14.55.11.59.48.-24.29)
s(1.85.0.-14.55.1.85.0.-0.77.1.85.48.-14.55)
s(1.85.0.-0.77.1.85.48.-0.77.1.85.48.-14.55)
s(1.85.48.-0.77.11.59.48.8.97.18.48.-7.46)
s(11.59.48.8.97.25.37.48.8.97.18.48.-7.46)
s(25.37.48.8.97.35.11.48.-0.77.18.48.-7.46)
s(35.11.48.-0.77.35.11.48.-14.55.18.48.-7.46)
s(35.11.48.-14.55.25.37.48.-24.29.18.48.-7.46)
s(25.37.48.-24.29.11.59.48.-24.29.18.48.-7.46)
s(11.59.48.-24.29.1.85.48.-14.55.18.48.-7.46)
s(1.85.48.-14.55.1.85.48.-0.77.18.48.-7.46)

A1.2 PISTON

s(0.0.0.9.74.0.9.74.0.25.0)
s(9.74.0.9.74.9.74.25.9.74.0.25.0)
s(23.52.0.9.74.33.26.0.0.23.52.25.9.74)
s(33.26.0.0.33.26.25.0.23.52.25.9.74)
s(33.26.0.0.33.26.0.-13.78.33.26.25.0)
s(33.26.0.-13.78.33.26.25.-13.78.33.26.25.0)
s(33.26.0.-13.78.23.52.0.-23.52.33.26.25.-13.78)
s(23.52.0.-23.52.23.52.25.-23.52.33.26.25.-13.78)
s(9.74.0.-23.52.0.0.-13.78.9.74.25.-23.52)
s(0.0.-13.78.0.25.-13.78.9.74.25.-23.52)
s(0.0.-13.78.0.0.0.0.25.-13.78)
s(0.0.0.0.25.0.0.25.-13.78)
s(9.74.0.9.74.15.1.2.9.74.12.93.4.16.9.74)
s(9.74.0.9.74.18.16.2.9.74.15.1.2.9.74)
s(9.74.0.9.74.23.52.0.9.74.18.16.2.9.74)
s(23.52.0.9.74.20.33.4.16.9.74.18.16.2.9.74)
s(23.52.0.9.74.20.33.7.23.9.74.20.33.4.16.9.74)
s(23.52.0.9.74.23.52.25.9.74.20.33.7.23.9.74)
s(23.52.25.9.74.18.16.9.39.9.74.20.33.7.23.9.74)
s(23.52.25.9.74.9.74.25.9.74.18.16.9.39.9.74)
s(9.74.25.9.74.15.1.9.39.9.74.18.16.9.39.9.74)
s(9.74.25.9.74.12.93.7.23.9.74.15.1.9.39.9.74)
s(9.74.25.9.74.9.74.0.9.74.12.93.7.23.9.74)
s(9.74.0.9.74.12.93.4.16.9.74.12.93.7.23.9.74)
s(9.74.0.-23.52.15.1.2.-23.52.12.93.4.16.-23.52)
s(9.74.0.-23.52.18.16.2.-23.52.15.1.2.-23.52)
s(9.74.0.-23.52.23.52.0.-23.52.18.16.2.-23.52)
s(23.52.0.-23.52.20.33.4.16.-23.52.18.16.2.-23.52)
s(23.52.0.-23.52.20.33.7.23.-23.52.20.33.4.16.-23.52)
s(23.52.0.-23.52.23.52.25.-23.52.20.33.7.23.-23.52)
s(23.52.25.-23.52.18.16.9.39.-23.52.20.33.7.23.-23.52)
s(23.52.25.-23.52.9.74.25.-23.52.18.16.9.39.-23.52)
s(9.74.25.-23.52.15.1.9.39.-23.52.18.16.9.39.-23.52)
s(9.74.25.-23.52.12.93.7.23.-23.52.15.1.9.39.-23.52)
s(9.74.25.-23.52.9.74.0.-23.52.12.93.7.23.-23.52)
s(9.74.0.-23.52.12.93.4.16.-23.52.12.93.7.23.-23.52)
s(0.25.0.9.74.25.9.74.16.63.25.-6.89)
s(9.74.25.9.74.23.52.25.9.74.16.63.25.-6.89)
s(23.52.25.9.74.33.26.25.0.16.63.25.-6.89)
s(33.26.25.0.33.26.25.-13.78.16.63.25.-6.89)
s(33.26.25.-13.78.23.52.25.-23.52.16.63.25.-6.89)
s(23.52.25.-23.52.9.74.25.-23.52.16.63.25.-6.89)
s(9.74.25.-23.52.0.25.-13.78.16.63.25.-6.89)
s(0.25.-13.78.0.25.0.16.63.25.-6.89)
s(0.0.0.9.74.0.9.74.1.88.0.-0.77)
s(9.74.0.9.74.10.51.0.7.9.1.88.0.-0.77)
s(9.74.0.9.74.23.52.0.9.74.10.51.0.7.9)
s(23.52.0.9.74.22.75.0.7.9.10.51.0.7.9)
s(23.52.0.9.74.33.26.0.0.22.75.0.7.9)
s(33.26.0.0.31.41.0.-0.77.22.75.0.7.9)
s(33.26.0.0.33.26.0.-13.78.31.41.0.-0.77)
s(33.26.0.-13.78.31.41.0.-13.01.31.41.0.-0.77)
s(33.26.0.-13.78.23.52.0.-23.52.31.41.0.-13.01)
s(23.52.0.-23.52.22.75.0.-21.67.31.41.0.-13.01)
s(23.52.0.-23.52.9.74.0.-23.52.22.75.0.-21.67)
s(9.74.0.-23.52.10.51.0.-21.67.22.75.0.-21.67)
s(9.74.0.-23.52.0.0.-13.78.10.51.0.-21.67)
s(0.0.-13.78.1.88.0.-13.01.10.51.0.-21.67)
s(0.0.-13.78.0.0.0.1.88.0.-13.01)
s(0.0.0.0.1.88.0.-0.77.1.88.0.-13.01)
s(1.88.0.-0.77.10.51.0.7.9.1.88.23.-0.77)
s(10.51.0.7.9.10.51.23.7.9.1.88.23.-0.77)
s(22.75.0.7.9.31.41.0.-0.77.22.75.23.7.9)
s(31.41.0.-0.77.31.41.23.-0.77.22.75.23.7.9)
s(31.41.0.-0.77.31.41.0.-13.01.31.41.23.-0.77)
s(31.41.0.-13.01.31.41.23.-13.01.31.41.23.-0.77)
s(31.41.0.-13.01.22.75.0.-21.67.31.41.23.-13.01)
s(22.75.0.-21.67.22.75.23.-21.67.31.41.23.-13.01)
s(10.51.0.-21.67.1.88.0.-13.01.10.51.23.-21.67)
s(1.88.0.-13.01.1.88.23.-13.01.10.51.23.-21.67)
s(1.88.0.-13.01.1.88.0.-0.77.1.88.23.-13.01)
s(1.88.0.-0.77.1.88.23.-0.77.1.88.23.-13.01)
s(10.51.0.7.9.15.1.2.7.9.12.93.4.16.7.9)
s(10.51.0.7.9.18.16.2.7.9.15.1.2.7.9)
s(10.51.0.7.9.22.75.0.7.9.18.16.2.7.9)
s(22.75.0.7.9.20.33.4.16.7.9.18.16.2.7.9)
s(22.75.0.7.9.20.33.7.23.7.9.20.33.4.16.7.9)
s(22.75.0.7.9.22.75.23.7.9.20.33.7.23.7.9)
s(22.75.23.7.9.18.16.9.39.7.9.20.33.7.23.7.9)
s(22.75.23.7.9.10.51.23.7.9.18.16.9.39.7.9)
s(10.51.23.7.9.15.1.9.39.7.9.18.16.9.39.7.9)
s(10.51.23.7.9.12.93.7.23.7.9.15.1.9.39.7.9)
s(10.51.23.7.9.10.51.0.7.9.12.93.7.23.7.9)
s(10.51.0.7.9.12.93.4.16.7.9.12.93.7.23.7.9)
s(10.51.0.-21.67.15.1.2.-21.67.12.93.4.16.-21.67)
s(10.51.0.-21.67.18.16.2.-21.67.15.1.2.-21.67)
s(10.51.0.-21.67.22.75.0.-21.67.18.16.2.-21.67)
s(22.75.0.-21.67.20.33.4.16.-21.67.18.16.2.-21.67)
s(22.75.0.-21.67.20.33.7.23.-21.67.20.33.4.16.-21.67)
s(22.75.0.-21.67.22.75.23.-21.67.20.33.7.23.-21.67)
s(22.75.23.-21.67.18.16.9.39.-21.67.20.33.7.23.-21.67)
s(22.75.23.-21.67.10.51.23.-21.67.18.16.9.39.-21.67)
s(10.51.23.-21.67.15.1.9.39.-21.67.18.16.9.39.-21.67)
s(10.51.23.-21.67.12.93.7.23.-21.67.15.1.9.39.-21.67)
s(10.51.23.-21.67.10.51.0.-21.67.12.93.7.23.-21.67)
s(10.51.0.-21.67.12.93.4.16.-21.67.12.93.7.23.-21.67)
s(1.88.23.-0.77.10.51.23.7.9.16.63.23.-6.89)
s(10.51.23.7.9.22.75.23.7.9.16.63.23.-6.89)
s(22.75.23.7.9.31.41.23.-0.77.16.63.23.-6.89)
s(31.41.23.-0.77.31.41.23.-13.01.16.63.23.-6.89)

s(31.41.23.-13.01.22.75.23.-21.67.16.63.23.-6.89)
s(22.75.23.-21.67.10.51.23.-21.67.16.63.23.-6.89)
s(10.51.23.-21.67.1.88.23.-13.01.16.63.23.-6.89)
s(1.88.23.-13.01.1.88.23.-0.77.16.63.23.-6.89)
s(15.1.2.9.74.18.16.2.9.74.15.1.2.7.9)
s(18.16.2.9.74.18.16.2.7.9.15.1.2.7.9)
s(18.16.2.9.74.20.33.4.16.9.74.18.16.2.7.9)
s(20.33.4.16.9.74.20.33.4.16.7.9.18.16.2.7.9)
s(20.33.4.16.9.74.20.33.7.23.9.74.20.33.4.16.7.9)
s(20.33.7.23.9.74.20.33.7.23.7.9.20.33.4.16.7.9)
s(20.33.7.23.9.74.18.16.9.39.9.74.20.33.7.23.7.9)
s(18.16.9.39.9.74.18.16.9.39.7.9.20.33.7.23.7.9)
s(18.16.9.39.9.74.15.1.9.39.9.74.18.16.9.39.7.9)
s(15.1.9.39.9.74.15.1.9.39.7.9.18.16.9.39.7.9)
s(15.1.9.39.9.74.12.93.7.23.9.74.15.1.9.39.7.9)
s(12.93.7.23.9.74.12.93.7.23.7.9.15.1.9.39.7.9)
s(12.93.7.23.9.74.12.93.4.16.9.74.12.93.7.23.7.9)
s(12.93.4.16.9.74.12.93.4.16.7.9.12.93.7.23.7.9)
s(12.93.4.16.9.74.15.1.2.9.74.12.93.4.16.7.9)
s(15.1.2.9.74.15.1.2.7.9.12.93.4.16.7.9)
s(15.1.2.-21.67.18.16.2.-21.67.15.1.2.-23.52)
s(18.16.2.-21.67.18.16.2.-23.52.15.1.2.-23.52)
s(18.16.2.-21.67.20.33.4.16.-21.67.18.16.2.-23.52)
s(20.33.4.16.-21.67.20.33.4.16.-23.52.18.16.2.-23.52)
s(20.33.4.16.-21.67.20.33.7.23.-21.67.20.33.4.16.-23.52)
s(20.33.7.23.-21.67.20.33.7.23.-23.52.20.33.4.16.-23.52)
s(20.33.7.23.-21.67.18.16.9.39.-21.67.20.33.7.23.-23.52)
s(18.16.9.39.-21.67.18.16.9.39.-23.52.20.33.7.23.-23.52)
s(18.16.9.39.-21.67.15.1.9.39.-21.67.18.16.9.39.-23.52)
s(15.1.9.39.-21.67.15.1.9.39.-23.52.18.16.9.39.-23.52)
s(15.1.9.39.-21.67.12.93.7.23.-21.67.15.1.9.39.-23.52)
s(12.93.7.23.-21.67.12.93.7.23.-23.52.15.1.9.39.-23.52)
s(12.93.7.23.-21.67.12.93.4.16.-21.67.12.93.7.23.-23.52)
s(12.93.4.16.-21.67.12.93.4.16.-23.52.12.93.7.23.-23.52)
s(12.93.4.16.-21.67.15.1.2.-21.67.12.93.4.16.-23.52)
s(15.1.2.-21.67.15.1.2.-23.52.12.93.4.16.-23.52)

A1.3 PIN

s(0.0.0.0.0.3.06.33.26.0.0)
s(0.0.3.06.33.26.0.3.06.33.26.0.0)
s(0.0.3.06.0.2.16.5.23.33.26.0.3.06)
s(0.2.16.5.23.33.26.2.16.5.23.33.26.0.3.06)
s(0.2.16.5.23.0.5.23.5.23.33.26.2.16.5.23)
s(0.5.23.5.23.33.26.5.23.5.23.33.26.2.16.5.23)
s(0.5.23.5.23.0.7.39.3.06.33.26.5.23.5.23)
s(0.7.39.3.06.33.26.7.39.3.06.33.26.5.23.5.23)
s(0.7.39.3.06.0.7.39.0.33.26.7.39.3.06)
s(0.7.39.0.33.26.7.39.0.33.26.7.39.3.06)
s(0.7.39.0.0.5.23.-2.17.33.26.7.39.0)
s(0.5.23.-2.17.33.26.5.23.-2.17.33.26.7.39.0)
s(0.5.23.-2.17.0.2.16.-2.17.33.26.5.23.-2.17)
s(0.2.16.-2.17.33.26.2.16.-2.17.33.26.5.23.-2.17)
s(0.2.16.-2.17.0.0.0.33.26.2.16.-2.17)
s(0.0.0.33.26.0.0.33.26.2.16.-2.17)
s(0.0.0.0.0.3.06.0.3.7.1.53)
s(0.0.3.06.0.2.16.5.23.0.3.7.1.53)
s(0.2.16.5.23.0.5.23.5.23.0.3.7.1.53)
s(0.5.23.5.23.0.7.39.3.06.0.3.7.1.53)
s(0.7.39.3.06.0.7.39.0.0.3.7.1.53)
s(0.7.39.0.0.5.23.-2.17.0.3.7.1.53)
s(0.5.23.-2.17.0.2.16.-2.17.0.3.7.1.53)
s(0.2.16.-2.17.0.0.0.0.3.7.1.53)
s(33.26.0.0.33.26.0.3.06.33.26.3.7.1.53)
s(33.26.0.3.06.33.26.2.16.5.23.33.26.3.7.1.53)
s(33.26.2.16.5.23.33.26.5.23.5.23.33.26.3.7.1.53)
s(33.26.5.23.5.23.33.26.7.39.3.06.33.26.3.7.1.53)
s(33.26.7.39.3.06.33.26.7.39.0.33.26.3.7.1.53)
s(33.26.7.39.0.33.26.5.23.-2.17.33.26.3.7.1.53)
s(33.26.5.23.-2.17.33.26.2.16.-2.17.33.26.3.7.1.53)
s(33.26.2.16.-2.17.33.26.0.0.33.26.3.7.1.53)

A1.4 BUSH

s(0.0.0.0.0.-20.4.6.0.-20)
s(0.0.0.4.6.0.0.4.6.0.-20)
s(-3.24.3.24.0.-3.24.3.24.-20.0.0.-20)
s(-3.24.3.24.0.0.0.0.0.0.-20)
s(-3.24.7.84.0.-3.24.7.84.-20.-3.24.3.24.-20)
s(-3.24.7.84.0.-3.24.3.24.0.-3.24.3.24.-20)
s(0.11.08.0.0.11.08.-20.-3.24.7.84.-20)
s(0.11.08.0.-3.24.7.84.0.-3.24.7.84.-20)
s(0.11.08.0.0.11.08.-20.4.6.11.08.-20)
s(0.11.08.0.4.6.11.08.0.4.6.11.08.-20)
s(4.6.11.08.0.4.6.11.08.-20.7.84.7.84.-20)
s(4.6.11.08.0.7.84.7.84.0.7.84.7.84.-20)
s(7.84.7.84.0.7.84.7.84.-20.7.84.3.24.-20)
s(7.84.7.84.0.7.84.3.24.0.7.84.3.24.-20)
s(7.84.3.24.0.7.84.3.24.-20.4.6.0.-20)
s(7.84.3.24.0.4.6.0.0.4.6.0.-20)
s(0.77.1.85.0.0.77.1.85.-20.3.83.1.85.-20)
s(0.77.1.85.0.3.83.1.85.0.3.83.1.85.-20)
s(-1.4.4.01.0.-1.4.4.01.-20.0.77.1.85.-20)
s(-1.4.4.01.0.0.77.1.85.0.0.77.1.85.-20)
s(-1.4.7.08.0.-1.4.7.08.-20.-1.4.4.01.-20)
s(-1.4.7.08.0.-1.4.4.01.0.-1.4.4.01.-20)
s(-1.4.7.08.0.-1.4.7.08.-20.0.77.9.24.-20)
s(-1.4.7.08.0.0.77.9.24.0.0.77.9.24.-20)
s(0.77.9.24.0.0.77.9.24.-20.3.83.9.24.-20)
s(0.77.9.24.0.3.83.9.24.0.3.83.9.24.-20)
s(3.83.9.24.0.3.83.9.24.-20.6.7.08.-20)
s(3.83.9.24.0.6.7.08.0.6.7.08.-20)
s(6.7.08.0.6.7.08.-20.6.4.01.-20)
s(6.7.08.0.6.4.01.0.6.4.01.-20)
s(6.4.01.0.6.4.01.-20.3.83.1.85.-20)
s(6.4.01.0.3.83.1.85.0.3.83.1.85.-20)
s(0.0.0.4.6.0.0.0.77.1.85.0)
s(4.6.0.0.3.83.1.85.0.0.77.1.85.0)
s(-3.24.3.24.0.0.0.0.-1.4.4.01.0)
s(0.0.0.0.77.1.85.0.-1.4.4.01.0)
s(-3.24.7.84.0.-3.24.3.24.0.-1.4.7.08.0)
s(-3.24.3.24.0.-1.4.4.01.0.-1.4.7.08.0)
s(0.11.08.0.-3.24.7.84.0.0.77.9.24.0)
s(-3.24.7.84.0.-1.4.7.08.0.0.77.9.24.0)
s(4.6.11.08.0.0.11.08.0.3.83.9.24.0)
s(0.11.08.0.0.77.9.24.0.3.83.9.24.0)
s(7.84.7.84.0.4.6.11.08.0.6.7.08.0)
s(4.6.11.08.0.3.83.9.24.0.6.7.08.0)
s(7.84.3.24.0.7.84.7.84.0.6.4.01.0)
s(7.84.7.84.0.6.7.08.0.6.4.01.0)
s(4.6.0.0.7.84.3.24.0.3.83.1.85.0)
s(7.84.3.24.0.6.4.01.0.3.83.1.85.0)
s(0.0.-20.4.6.0.-20.0.77.1.85.-20)
s(4.6.0.-20.3.83.1.85.-20.0.77.1.85.-20)
s(-3.24.3.24.-20.0.0.-20.-1.4.4.01.-20)
s(0.0.-20.0.77.1.85.-20.-1.4.4.01.-20)
s(-3.24.7.84.-20.-3.24.3.24.-20.-1.4.7.08.-20)
s(-3.24.3.24.-20.-1.4.4.01.-20.-1.4.7.08.-20)
s(0.11.08.-20.-3.24.7.84.-20.0.77.9.24.-20)
s(-3.24.7.84.-20.-1.4.7.08.-20.0.77.9.24.-20)
s(4.6.11.08.-20.0.11.08.-20.3.83.9.24.-20)
s(0.11.08.-20.0.77.9.24.-20.3.83.9.24.-20)
s(7.84.7.84.-20.4.6.11.08.-20.6.7.08.-20)
s(4.6.11.08.-20.3.83.9.24.-20.6.7.08.-20)
s(7.84.3.24.-20.7.84.7.84.-20.6.4.01.-20)
s(7.84.7.84.-20.6.7.08.-20.6.4.01.-20)
s(4.6.0.-20.7.84.3.24.-20.3.83.1.85.-20)
s(7.84.3.24.-20.6.4.01.-20.3.83.1.85.-20)

A1.5 CONROD

s(0.0.0.8.75.0.0.0.10.0)
s(8.75.0.0.8.75.3.83.0.0.10.0)
s(8.75.3.83.0.14.16.9.24.0.0.10.0)
s(0.10.0.14.16.9.24.0.10.20.0)
s(10.20.0.14.16.9.24.0.21.82.9.24.0)
s(10.20.0.21.82.9.24.0.26.20.0)
s(26.20.0.21.82.9.24.0.36.10.0)
s(21.82.9.24.0.27.25.3.83.0.36.10.0)
s(27.25.3.83.0.27.25.0.0.36.10.0)
s(27.25.0.0.36.0.0.36.10.0)
s(10.20.0.26.20.0.21.06.52.61.0)
s(10.20.0.14.94.52.61.0.21.06.52.61.0)
s(14.94.52.61.0.21.06.52.61.0.15.7.54.46.0)
s(15.7.54.46.0.21.06.52.61.0.20.3.54.46.0)
s(10.61.56.94.0.14.94.52.61.0.12.46.57.7.0)
s(12.46.57.7.0.14.94.52.61.0.15.7.54.46.0)
s(10.61.63.06.0.10.61.56.94.0.12.46.62.3.0)
s(12.46.62.3.0.10.61.56.94.0.12.46.57.7.0)
s(14.94.67.39.0.10.61.63.06.0.12.46.62.3.0)
s(14.94.67.39.0.12.46.62.3.0.15.7.65.54.0)
s(21.06.67.39.0.14.94.67.39.0.15.7.65.54.0)
s(21.06.67.39.0.15.7.65.54.0.20.3.65.54.0)
s(23.54.62.3.0.21.06.67.39.0.20.3.65.54.0)
s(25.39.63.06.0.21.06.67.39.0.23.54.62.3.0)
s(25.39.56.94.0.25.39.63.06.0.23.54.62.3.0)
s(25.39.56.94.0.23.54.62.3.0.23.54.57.7.0)
s(21.06.52.61.0.25.39.56.94.0.23.54.57.7.0)
s(21.06.52.61.0.23.54.57.7.0.20.3.54.46.0)
s(0.0.-20.8.75.0.-20.0.10.-20)
s(8.75.0.-20.8.75.3.83.-20.0.10.-20)
s(8.75.3.83.-20.14.16.9.24.-20.0.10.-20)
s(0.10.-20.14.16.9.24.-20.10.20.-20)
s(10.20.-20.14.16.9.24.-20.21.82.9.24.-20)
s(10.20.-20.21.82.9.24.-20.26.20.-20)
s(26.20.-20.21.82.9.24.-20.36.10.-20)
s(21.82.9.24.-20.27.25.3.83.-20.36.10.-20)
s(27.25.3.83.-20.27.25.0.-20.36.10.-20)
s(27.25.0.-20.36.0.-20.36.10.-20)
s(10.20.-20.26.20.-20.21.06.52.61.-20)
s(10.20.-20.14.94.52.61.-20.21.06.52.61.-20)
s(14.94.52.61.-20.21.06.52.61.-20.15.7.54.46.-20)
s(15.7.54.46.-20.21.06.52.61.-20.20.3.54.46.-20)

s(10.61.56.94.-20.14.94.52.61.-20.12.46.57.7.-20)
s(12.46.57.7.-20.14.94.52.61.-20.15.7.54.46.-20)
s(10.61.63.06.-20.10.61.56.94.-20.12.46.62.3.-20)
s(12.46.62.3.-20.10.61.56.94.-20.12.46.57.7.-20)
s(14.94.67.39.-20.10.61.63.06.-20.12.46.62.3.-20)
s(14.94.67.39.-20.12.46.62.3.-20.15.7.65.54.-20)
s(21.06.67.39.-20.14.94.67.39.-20.15.7.65.54.-20)
s(21.06.67.39.-20.15.7.65.54.-20.20.3.65.54.-20)
s(23.54.62.3.-20.21.06.67.39.-20.20.3.65.54.-20)
s(25.39.63.06.-20.21.06.67.39.-20.23.54.62.3.-20)
s(25.39.56.94.-20.25.39.63.06.-20.23.54.62.3.-20)
s(25.39.56.94.-20.23.54.62.3.-20.23.54.57.7.-20)
s(21.06.52.61.-20.25.39.56.94.-20.23.54.57.7.-20)
s(21.06.52.61.-20.23.54.57.7.-20.20.3.54.46.-20)
s(0.0.0.0.0.-20.0.10.-20)
s(0.0.0.0.10.0.0.10.-20)
s(0.10.0.0.10.-20.10.20.-20)
s(0.10.0.10.20.0.10.20.-20)
s(10.20.0.10.20.-20.14.94.52.61.-20)
s(10.20.0.14.94.52.61.0.14.94.52.61.-20)
s(14.94.52.61.0.14.94.52.61.-20.10.61.56.94.-20)
s(14.94.52.61.0.10.61.56.94.0.10.61.56.94.-20)
s(10.61.56.94.0.10.61.56.94.-20.10.61.63.06.-20)
s(10.61.56.94.0.10.61.63.06.0.10.61.63.06.-20)
s(10.61.63.06.0.10.61.63.06.-20.14.94.67.39.-20)
s(10.61.63.06.0.14.94.67.39.0.14.94.67.39.-20)
s(14.94.67.39.0.14.94.67.39.-20.21.06.67.39.-20)
s(14.94.67.39.0.21.06.67.39.0.21.06.67.39.-20)
s(21.06.67.39.0.21.06.67.39.-20.25.39.63.06.-20)
s(21.06.67.39.0.25.39.63.06.0.25.39.63.06.-20)
s(25.39.63.06.0.25.39.63.06.-20.25.39.56.94.-20)
s(25.39.63.06.0.25.39.56.94.0.25.39.56.94.-20)
s(25.39.56.94.0.25.39.56.94.-20.21.06.52.61.-20)
s(25.39.56.94.0.21.06.52.61.0.21.06.52.61.-20)
s(21.06.52.61.0.21.06.52.61.-20.26.20.-20)
s(21.06.52.61.0.26.20.0.26.20.-20)
s(26.20.0.26.20.-20.36.10.-20)
s(26.20.0.36.10.0.36.10.-20)
s(36.10.0.36.10.-20.36.0.-20)
s(36.10.0.36.0.0.36.0.-20)
s(27.25.0.0.27.25.0.-20.36.0.-20)
s(27.25.0.0.36.0.0.36.0.-20)
s(27.25.3.83.0.27.25.3.83.-20.27.25.0.-20)
s(27.25.3.83.0.27.25.0.0.27.25.0.-20)
s(21.82.9.24.0.21.82.9.24.-20.27.25.3.83.-20)
s(21.82.9.24.0.27.25.3.83.0.27.25.3.83.-20)
s(14.16.9.24.0.14.16.9.24.-20.21.82.9.24.-20)
s(14.16.9.24.0.21.82.9.24.0.21.82.9.24.-20)
s(8.75.3.83.0.8.75.3.83.-20.14.16.9.24.-20)
s(8.75.3.83.0.14.16.9.24.0.14.16.9.24.-20)
s(8.75.0.0.8.75.0.-20.8.75.3.83.-20)
s(8.75.0.0.8.75.3.83.0.8.75.3.83.-20)
s(0.0.0.0.0.-20.8.75.0.-20)
s(0.0.0.8.75.0.0.8.75.0.-20)
s(15.7.54.46.0.15.7.54.46.-20.20.3.54.46.-20)
s(15.7.54.46.0.20.3.54.46.0.20.3.54.46.-20)
s(12.46.57.7.0.12.46.57.7.-20.15.7.54.46.-20)
s(12.46.57.7.0.15.7.54.46.0.15.7.54.46.-20)
s(12.46.62.3.0.12.46.62.3.-20.12.46.57.7.-20)
s(12.46.62.3.0.12.46.57.7.0.12.46.57.7.-20)
s(15.7.65.54.0.15.7.65.54.-20.12.46.62.3.-20)
s(15.7.65.54.0.12.46.62.3.0.12.46.62.3.-20)
s(15.7.65.54.0.15.7.65.54.-20.20.3.65.54.-20)
s(15.7.65.54.0.20.3.65.54.0.20.3.65.54.-20)
s(20.3.65.54.0.20.3.65.54.-20.23.54.62.3.-20)
s(20.3.65.54.0.23.54.62.3.0.23.54.62.3.-20)
s(23.54.62.3.0.23.54.62.3.-20.23.54.57.7.-20)
s(23.54.62.3.0.23.54.57.7.0.23.54.57.7.-20)
s(23.54.57.7.0.23.54.57.7.-20.20.3.54.46.-20)
s(23.54.57.7.0.20.3.54.46.0.20.3.54.46.-20)

A1.6 SHELL

s(0.0.0.1.85.0.0.0.3.83.0)
s(1.85.0.0.1.85.3.06.0.0.3.83.0)
s(1.85.3.06.0.5.41.9.24.0.0.3.83.0)
s(1.85.3.06.0.5.41.9.24.0.6.18.7.4.0)
s(6.18.7.4.0.5.41.9.24.0.12.3.7.4.0)
s(5.41.9.24.0.12.3.7.4.0.13.07.9.24.0)
s(12.3.7.4.0.13.07.9.24.0.16.63.3.06.0)
s(13.07.9.24.0.16.63.3.06.0.18.5.3.83.0)
s(16.63.3.06.0.18.5.3.83.0.16.63.0.0)
s(16.63.0.0.18.5.3.83.0.18.5.0.0)
s(0.0.-20.1.85.0.-20.0.3.83.-20)
s(1.85.0.-20.1.85.3.06.-20.0.3.83.-20)
s(1.85.3.06.-20.5.41.9.24.-20.0.3.83.-20)
s(1.85.3.06.-20.5.41.9.24.-20.6.18.7.4.-20)
s(6.18.7.4.-20.5.41.9.24.-20.12.3.7.4.-20)
s(5.41.9.24.-20.12.3.7.4.-20.13.07.9.24.-20)
s(12.3.7.4.-20.13.07.9.24.-20.16.63.3.06.-20)
s(13.07.9.24.-20.16.63.3.06.-20.18.5.3.83.-20)
s(16.63.3.06.-20.18.5.3.83.-20.16.63.0.-20)
s(16.63.0.-20.18.5.3.83.-20.18.5.0.-20)
s(0.0.0.0.3.83.-20.0.0.-20)
s(0.0.0.0.0.3.83.-20.0.3.83.0)
s(0.3.83.0.0.3.83.-20.5.41.9.24.-20)
s(0.3.83.0.5.41.9.24.-20.5.41.9.24.0)
s(5.41.9.24.0.5.41.9.24.-20.13.07.9.24.-20)
s(5.41.9.24.0.13.07.9.24.-20.13.07.9.24.0)
s(13.07.9.24.0.13.07.9.24.-20.18.5.3.83.-20)
s(13.07.9.24.0.18.5.3.83.-20.18.5.3.83.0)
s(18.5.3.83.0.18.5.3.83.-20.18.5.0.-20)
s(18.5.3.83.0.18.5.0.-20.18.5.0.0)
s(1.85.0.0.1.85.0.-20.1.85.3.06.-20)
s(1.85.0.0.1.85.3.06.-20.1.85.3.06.0)
s(1.85.3.06.0.1.85.3.06.-20.6.18.7.4.-20)
s(1.85.3.06.0.6.18.7.4.-20.6.18.7.4.0)
s(6.18.7.4.0.6.18.7.4.-20.12.3.7.4.-20)
s(6.18.7.4.0.12.3.7.4.-20.12.3.7.4.0)
s(12.3.7.4.0.12.3.7.4.-20.16.63.3.06.-20)
s(12.3.7.4.0.16.63.3.06.-20.16.63.3.06.0)
s(16.63.3.06.0.16.63.3.06.-20.16.63.0.-20)
s(16.63.3.06.0.16.63.0.-20.16.63.0.0)
s(0.0.0.1.85.0.0.1.85.0.-20)
s(0.0.0.1.85.0.-20.0.0.-20)
s(16.63.0.0.18.5.0.0.18.5.0.-20)
s(16.63.0.0.18.5.0.-20.16.63.0.-20)

A1.7 END-BLOCK

s(0.0.0.36.0.0.21.82.2.76.0)
s(0.0.0.21.82.2.76.0.14.16.2.76.0)
s(0.0.0.14.16.2.76.0.8.75.8.17.0)
s(0.0.0.8.75.8.17.0.0.12.0)
s(36.0.0.27.25.8.17.0.21.82.2.76.0)
s(36.0.0.36.12.0.27.25.8.17.0)
s(0.12.0.8.75.8.17.0.8.75.12.0)
s(36.12.0.27.25.12.0.27.25.8.17.0)
s(0.0.-20.36.0.-20.21.82.2.76.-20)
s(0.0.-20.21.82.2.76.-20.14.16.2.76.-20)
s(0.0.-20.14.16.2.76.-20.8.75.8.17.-20)
s(0.0.-20.8.75.8.17.-20.0.12.-20)
s(36.0.-20.27.25.8.17.-20.21.82.2.76.-20)
s(36.0.-20.36.12.-20.27.25.8.17.-20)
s(0.12.-20.8.75.8.17.-20.8.75.12.-20)
s(36.12.-20.27.25.12.-20.27.25.8.17.-20)
s(0.0.0.0.0.-20.0.12.0)
s(0.0.-20.0.12.-20.0.12.0)
s(8.75.12.0.0.12.0.0.12.-20)
s(8.75.12.0.8.75.12.-20.0.12.-20)
s(8.75.8.17.0.8.75.12.0.8.75.12.-20)
s(8.75.8.17.0.8.75.8.17.-20.8.75.12.-20)
s(8.75.8.17.0.14.16.2.76.0.8.75.8.17.-20)
s(14.16.2.76.0.14.16.2.76.-20.8.75.8.17.-20)
s(14.16.2.76.0.21.82.2.76.0.14.16.2.76.-20)
s(21.82.2.76.0.21.82.2.76.-20.14.16.2.76.-20)
s(21.82.2.76.0.27.25.8.17.0.27.25.8.17.-20)
s(21.82.2.76.0.21.82.2.76.-20.27.25.8.17.-20)
s(27.25.8.17.0.27.25.12.0.27.25.12.-20)
s(27.25.8.17.0.27.25.8.17.-20.27.25.12.-20)
s(27.25.12.0.36.12.0.27.25.12.-20)
s(36.12.0.36.12.-20.27.25.12.-20)
s(36.12.0.36.0.0.36.0.-20)
s(36.12.0.36.12.-20.36.0.-20)
s(0.0.0.36.0.0.36.0.-20)
s(0.0.0.0.0.-20.36.0.-20)

A1.8 CRANK

s(0.0.0.26.79.0.0.16.45.44.28.0)
s(0.0.0.16.45.44.28.0.10.33.44.28.0)
s(0.0.0.10.33.44.28.0.6.48.62.0)
s(0.0.0.6.48.62.0.-18.94.18.94.0)
s(-18.94.18.94.0.6.48.62.0.-18.94.45.73.0)
s(-18.94.45.73.0.6.48.62.0.6.54.74.0)
s(-18.94.45.73.0.6.54.74.0.0.64.67.0)
s(0.64.67.0.6.54.74.0.10.33.59.08.0)
s(0.64.67.0.10.33.59.08.0.26.79.64.67.0)
s(26.79.64.67.0.10.33.59.08.0.16.45.59.08.0)
s(26.79.64.67.0.16.45.59.08.0.20.78.54.74.0)
s(26.79.64.67.0.20.78.54.74.0.45.73.45.73.0)
s(45.73.45.73.0.20.78.54.74.0.20.78.48.62.0)
s(45.73.45.73.0.20.78.48.62.0.45.73.18.94.0)
s(45.73.18.94.0.20.78.48.62.0.16.45.44.28.0)
s(45.73.18.94.0.16.45.44.28.0.26.79.0.0)
s(10.33.44.28.0.16.45.44.28.0.10.33.44.28.20.25)
s(10.33.44.28.20.25.16.45.44.28.0.16.45.44.28.20.25)
s(6.48.62.0.10.33.44.28.0.6.48.62.20.25)
s(10.33.44.28.0.10.33.44.28.20.25.6.48.62.20.25)
s(6.54.74.0.6.48.62.0.6.54.74.20.25)
s(6.48.62.0.6.48.62.20.25.6.54.74.20.25)
s(10.33.59.08.0.6.54.74.0.10.33.59.08.20.25)
s(6.54.74.0.6.54.74.20.25.10.33.59.08.20.25)

s(16.45.59.08.0.10.33.59.08.0.16.45.59.08.20.25)
s(10.33.59.08.0.10.33.59.08.20.25.16.45.59.08.20.25)
s(20.78.54.74.0.16.45.59.08.0.20.78.54.74.20.25)
s(16.45.59.08.0.16.45.59.08.20.25.20.78.54.74.20.25)
s(20.78.48.62.0.20.78.54.74.0.20.78.48.62.20.25)
s(20.78.54.74.0.20.78.54.74.20.25.20.78.48.62.20.25)
s(16.45.44.28.0.20.78.48.62.0.16.45.44.28.20.25)
s(20.78.48.62.0.20.78.48.62.20.25.16.45.44.28.20.25)
s(10.33.44.28.20.25.16.45.44.28.20.25.13.39.51.67.20.25)
s(6.48.62.20.25.10.33.44.28.20.25.13.39.51.67.20.25)
s(6.54.74.20.25.6.48.62.20.25.13.39.51.67.20.25)
s(10.33.59.08.20.25.6.54.74.20.25.13.39.51.67.20.25)
s(16.45.59.08.20.25.10.33.59.08.20.25.13.39.51.67.20.25)
s(20.78.54.74.20.25.16.45.59.08.20.25.13.39.51.67.20.25)
s(20.78.48.62.20.25.20.78.54.74.20.25.13.39.51.67.20.25)
s(16.45.44.28.20.25.20.78.48.62.20.25.13.39.51.67.20.25)
s(0.0.0.26.79.0.0.0.-5)
s(26.79.0.0.26.79.0.-5.0.0.-5)
s(-18.94.18.94.0.0.0.0.-18.94.18.94.-5)
s(0.0.0.0.0.-5.-18.94.18.94.-5)
s(-18.94.45.73.0.-18.94.18.94.0.-18.94.45.73.-5)
s(-18.94.18.94.0.-18.94.18.94.-5.-18.94.45.73.-5)
s(0.64.67.0.-18.94.45.73.0.0.64.67.-5)
s(-18.94.45.73.0.-18.94.45.73.-5.0.64.67.-5)
s(26.79.64.67.0.0.64.67.0.26.79.64.67.-5)
s(0.64.67.0.0.64.67.-5.26.79.64.67.-5)
s(45.73.45.73.0.26.79.64.67.0.45.73.45.73.-5)
s(26.79.64.67.0.26.79.64.67.-5.45.73.45.73.-5)
s(45.73.18.94.0.45.73.45.73.0.45.73.18.94.-5)
s(45.73.45.73.0.45.73.45.73.-5.45.73.18.94.-5)
s(26.79.0.0.45.73.18.94.0.26.79.0.-5)
s(45.73.18.94.0.45.73.18.94.-5.26.79.0.-5)
s(0.0.-5.26.79.0.-5.13.39.32.34.-5)
s(-18.94.18.94.-5.0.0.-5.13.39.32.34.-5)
s(-18.94.45.73.-5.-18.94.18.94.-5.13.39.32.34.-5)
s(0.64.67.-5.-18.94.45.73.-5.13.39.32.34.-5)
s(26.79.64.67.-5.0.64.67.-5.13.39.32.34.-5)
s(45.73.45.73.-5.26.79.64.67.-5.13.39.32.34.-5)
s(45.73.18.94.-5.45.73.45.73.-5.13.39.32.34.-5)
s(26.79.0.-5.45.73.18.94.-5.13.39.32.34.-5)

A2 INTEGRATED SYSTEM CODE

A2.1 DATA RETRIEVAL

```
(1) getdata(H):-
        sql_FetchNext(H).
        getcurrent(H).!.
        getdata(H).

(2) getcurrent(H):-
        ID = sql_GetInteger(H,1).
        Shape = sql_GetString(H,3).
        decider(H,ID,Shape).

(3) decider(H,ID,"faceset"):-
        Name = sql_GetString(H,2).
        I = sql_GetString(H,4).
        G = sql_GetString(H,5).
        Nu = sql_GetString(H,6).
        Mr = sql_GetString(H,11).
        assertz(entity(ID,Name,faceset(Name),I,G,Nu,Mr),entity1).!.
        Con = sql_GetString(H,7).
        get_connect(Name,Con).!.
        Load = sql_GetString(H,8).
        makefact(ID,Load).!.
        BC = sql_GetString(H,9).
        makefact2(ID,BC).!.
        Coord = sql_GetString(H,10).
        make_vg(ID,Coord).!.

(4) decider(H,ID,"global"):-
        Name = sql_GetString(H,2).
        Con = sql_GetString(H,7).
        get_connect(Name,Con).!.
        assertz(gcs(ID,Name)).

(5) get_connect(Name,Con):-
        Con <> "na".
        openwrite(temp,"C:\\Jason\\Data\\File1").
        writedevice(temp).
        write(Con).
        closefile(temp).
        consult("C:\\Jason\\Data\\File1",entity1).
        deletefile("C:\\Jason\\Data\\File1").
        get_connect1(Name,[]).!:
        assertz(connected(Name,[]),entity1).

(6) get_connect1(Name,List):-
        chd(ID1,X,Y,Z,Rx,Ry,Rz,Type).
        retract(chd(ID1,X,Y,Z,Rx,Ry,Rz,Type)).
        NList = [e(ID1,X,Y,Z,Rx,Ry,Rz,Type)|List].
        get_connect1(Name,NList);
        assertz(connected(Name,List),entity1).

(7) makefact(ID,Fact):-
        Fact <> "na".
        openwrite(temp,"C:\\Jason\\Data\\File1").
        writedevice(temp).
        closefile(temp).
        consult("C:\\Jason\\Data\\File1",entity1).
        deletefile("C:\\Jason\\Data\\File1").
        makefact1(ID).!:
        true.

(8) makefact1(ID):-
        load(Type,Lx,Ly,Lz,X,Y,Z).
        retract(load(Type,Lx,Ly,Lz,X,Y,Z),entity1).
        asserta(load1(ID,Type,Lx,Ly,Lz,X,Y,Z),entity1).
        makefact1(ID);
        true.

(9) makefact2(ID,Fact):-
        Fact <> "na".
        openwrite(temp,"C:\\Jason\\Data\\File1").
        writedevice(temp).
        closefile(temp).
        consult("C:\\Jason\\Data\\File1",entity1).
        deletefile("C:\\Jason\\Data\\File1").
        makefact3(ID).!:
        true.

(10) makefact3(ID):-
        con(T1,T2,X,Y,Z).
        retract(con(T1,T2,X,Y,Z),entity1).
        asserta(con1(ID,T1,T2,X,Y,Z),entity1).
        makefact3(ID);
        true.

(11) make_vg(ID,Coord):-
        Coord <> "na".
        entity(ID,Name,_,_,_,_,_).
        assertz(vg(Name,Coord),entity1);
        true.

(12) decider1(ID):-
        entity(ID,Name,_,_,_,_,_).
        connected(Name,_).
        asserta(used(Name)).
        decider2(ID,Name).!:
        gcs(ID,_).
        dlg_note("The global coordinate system cannot be analysed!");
        Rest = " is not part of the database!",
        str_int(IDs,ID).
        concat(IDs,Rest,End).
        Start = "ID number ".
        concat(Start,End,New).
        dlg_Note(New).

(13) decider2(ID,Name):-
        connected(Name1,List1),
        not(gcs(_,Name1)).
        not(used(Name1)).
        asserta(used(Name1)).
        getname(ID1,Name1).
        decider3(ID,Name,ID1,List1),.!.
        decider2(ID,Name):
        retractall(used(_)).
        nextstep(Name,faceset(Name)).!.

(14) decider3(_,_,_,[]).
(15) decider3(ID,Name,ID1,[HT]):-
        H = e(ID,X,Y,Z,Rx,Ry,Rz,Type),
        Nx = -1*X,
        Ny = -1*Y,
        Nz = -1*Z,
        Nrx = -1*Rx,
        Nry = -1*Ry,
        Nrz = -1*Rz,
        retract(connected(Name,List),entity1),
        asserta(connected(Name,[e(ID1,Nx,Ny,Nz,Nrx,Nry,Nrz,Type)|List]),en
        tity1).
        decider3(ID,Name,ID1,T):
        decider3(ID,Name,ID1,T).

(16) getname(ID1,Name):-
        entity(ID1,Name,_,_,_,_,_):
        gcs(ID1,Name).
```

A2.2 GEOMETRIC REPRESENTATION

```
(1) nextstep(Name,faceset(Name)):-
        asserta(big(0.0)).
        concat("C:\\Jason\\data\\",Name,Newfile).
        consult(NewFile,surface),
        changedata.!,
        consult(NewFile,surface),
        changedata2.!,
        makeplane.!,
        makelist.!,
        retractall(used(_)).
        nextlist.!,
        retractall(used(_)),
        curvelist1(FL,List),
        asserta(used1(FL)),
        makecurve(List,List,FL),.!,
        retractall(used(_)).
        curvelist(Name1,CList),
        asserta(used(Name1)),
        checksolid(Name1,CList),.!,
        connected(Name,EList),
        retractall(counter(_)),
        asserta(counter1(0)),
        makechild(EList),.!,
        asserta(last(2)),
        createcsys(EList,EList),.!,
        retractall(last(_)),
        senddata(Name),.!,
        v_gauge(Name),.!.

(2) changedata:-
        s(X1,Y1,Z1,X2,Y2,Z2,X3,Y3,Z3).
        retract(s(X1,Y1,Z1,X2,Y2,Z2,X3,Y3,Z3)).
        asserta(s1("s1",p(X1,Y1,Z1),p(X2,Y2,Z2),p(X3,Y3,Z3))).
        changedata1.!:
        dlg_note("There is no data, or the data in the file is of the incorrect
        format!").

(3) changedata1:-
        s(X1,Y1,Z1,X2,Y2,Z2,X3,Y3,Z3).
        retract(s(X1,Y1,Z1,X2,Y2,Z2,X3,Y3,Z3)).
        s1(LastName,_,_,_).
        frontstr(1,LastName,_,Nos).
        str_real(Nos,Nor).
        Nor1 = Nor + 1.
```

```
        str_real(Nos1,Nor1),
        concat("s",Nos1,NewName),
        asserta(s1(NewName,p(X1,Y1,Z1),p(X2,Y2,Z2),p(X3,Y3,Z3))),
        changedata1;
        true.

(4) changedata2:-
        s(X1,Y1,Z1,X2,Y2,Z2,X3,Y3,Z3),
        retract(s(X1,Y1,Z1,X2,Y2,Z2,X3,Y3,Z3)),
        asserta(c("c1",p(X1,Y1,Z1),p(X2,Y2,Z2))),
        asserta(c("c2",p(X2,Y2,Z2),p(X3,Y3,Z3))),
        asserta(c("c3",p(X1,Y1,Z1),p(X3,Y3,Z3))),
        asserta(s2("s1",["c1","c2","c3"])),
        changedata3,!;
        dlg_note("There is no data, or the data in the file is of the incorrect
        format!").

(5) changedata3:-
        s(X1,Y1,Z1,X2,Y2,Z2,X3,Y3,Z3),
        retract(s(X1,Y1,Z1,X2,Y2,Z2,X3,Y3,Z3)),
        s2(LastName,_),
        frontstr(1,LastName,_,Nos),
        str_real(Nos,Nor),
        Nor1 = Nor + 1,
        str_real(Nos1,Nor1),
        concat("s",Nos1,NewName),
        asserta(s2(NewName,[])),
        curvedata1(NewName,X1,Y1,Z1,X2,Y2,Z2),
        curvedata2(NewName,X2,Y2,Z2,X3,Y3,Z3),
        curvedata3(NewName,X1,Y1,Z1,X3,Y3,Z3),
        changedata3;
        true.

(6) curvedata1(Name,X1,Y1,Z1,X2,Y2,Z2):-
        s2(Name,List),
        not(c(_,p(X1,Y1,Z1),p(X2,Y2,Z2))),
        not(c(_,p(X2,Y2,Z2),p(X1,Y1,Z1))),
        c(Last,_,_),
        frontstr(1,Last,_,Nos),
        str_real(Nos,Nor),
        Nor1 = Nor + 1,
        str_real(Nos1,Nor1),
        concat("c",Nos1,NewName),
        asserta(c(NewName,p(X1,Y1,Z1),p(X2,Y2,Z2))),
        retract(s2(Name,List)),
        asserta(s2(Name,[NewName|List])),
        c(C,p(X1,Y1,Z1),p(X2,Y2,Z2)),
        retract(s2(Name,List)),
        asserta(s2(Name,[C|List])),
        c(C,p(X2,Y2,Z2),p(X1,Y1,Z1)),
        retract(s2(Name,List)),
        asserta(s2(Name,[C|List])).

(7) curvedata2(Name,X2,Y2,Z2,X3,Y3,Z3):-
        s2(Name,List),
        not(c(_,p(X2,Y2,Z2),p(X3,Y3,Z3))),
        not(c(_,p(X3,Y3,Z3),p(X2,Y2,Z2))),
        c(Last,_,_),
        frontstr(1,Last,_,Nos),
        str_real(Nos,Nor),
        Nor1 = Nor + 1,
        str_real(Nos1,Nor1),
        concat("c",Nos1,NewName),
        asserta(c(NewName,p(X2,Y2,Z2),p(X3,Y3,Z3))),
        retract(s2(Name,List)),
        asserta(s2(Name,[NewName|List])),
        c(C,p(X2,Y2,Z2),p(X3,Y3,Z3)),
        retract(s2(Name,List)),
        asserta(s2(Name,[C|List])),
        c(C,p(X3,Y3,Z3),p(X2,Y2,Z2)),
        retract(s2(Name,List)),
        asserta(s2(Name,[C|List])).

(8) curvedata3(Name,X1,Y1,Z1,X3,Y3,Z3):-
        s2(Name,List),
        not(c(_,p(X1,Y1,Z1),p(X3,Y3,Z3))),
        not(c(_,p(X3,Y3,Z3),p(X1,Y1,Z1))),
        c(Last,_,_),
        frontstr(1,Last,_,Nos),
        str_real(Nos,Nor),
        Nor1 = Nor + 1,
        str_real(Nos1,Nor1),
        concat("c",Nos1,NewName),
        asserta(c(NewName,p(X1,Y1,Z1),p(X3,Y3,Z3))),
        retract(s2(Name,List)),
        asserta(s2(Name,[NewName|List])),
        c(C,p(X1,Y1,Z1),p(X3,Y3,Z3)),
        retract(s2(Name,List)),
        asserta(s2(Name,[C|List])),
        c(C,p(X3,Y3,Z3),p(X1,Y1,Z1)),
        retract(s2(Name,List)),
        asserta(s2(Name,[C|List])).

(9) makeplane:-
        s1(Name,p(X1,Y1,Z1),p(X2,Y2,Z2),p(X3,Y3,Z3)),
        not(plane(Name,_,_,_,_)),
        A = X1-X2, B = Y1-Y2, C = Z1-Z2,
        D = X1-X3, E = Y1-Y3, F = Z1-Z3,
        I = (B*F)-(C*E),
        J = -1*((A*F)-(C*D)),
        K = (A*E)-(B*D),
        H = I*(-X1)+J*(-Y1)+K*(-Z1),
        makeplane(Name,I,J,K,H);
        true.

(10) makeplane(Name,I,J,K,H):-
        H < 0+0.001,
        H > 0-0.001,
        makeplane1(Name,I,J,K,H);
        makeplane2(Name,I,J,K,H).

(11) makeplane1(Name,I,J,K,H):-
        I < 0+0.001,
        I > 0-0.001,
        J < 0+0.001,
        J > 0-0.001,
        K <> 0,
        asserta(plane(Name,I,J,1.0,H)),
        makeplane;
        I < 0+0.001,
        I > 0-0.001,
        J <> 0,
        K < 0+0.001,
        K > 0-0.001,
        asserta(plane(Name,I,1.0,K,H)),
        makeplane;
        I <> 0,
        J < 0+0.001,
        J > 0-0.001,
        K < 0+0.001,
        K > 0-0.001,
        asserta(plane(Name,1.0,J,K,H)),
        makeplane;
        asserta(plane(Name,I,J,K,H)),
        makeplane.

(12) makeplane2(Name,I,J,K,H):-
        I < 0+0.001,
        I > 0-0.001,
        J < 0+0.001,
        J > 0-0.001,
        K <> 0,
        H1 = H/K,
        asserta(plane(Name,I,J,1.0,H1)),
        makeplane;
        I < 0+0.001,
        I > 0-0.001,
        J <> 0,
        K < 0+0.001,
        K > 0-0.001,
        H1 = H/J,
        asserta(plane(Name,I,1.0,K,H1)),
        makeplane;
        I <> 0,
        J < 0+0.001,
        J > 0-0.001,
        K < 0+0.001,
        K > 0-0.001,
        H1 = H/I,
        asserta(plane(Name,1.0,J,K,H1)),
        makeplane;
        asserta(plane(Name,I,J,K,H)),
        makeplane.

(13) makelist:-
        plane(Name,_,_,_,_),
        asserta(facelist("L1",[Name])),
        asserta(used(Name)),
        makelist1("L1"),!.

(14) makelist1(FL):-
        plane(Name,A,B,C,D),
        not(used(Name)),
        facelist(FL,List),
        List = [Name1|_],
        plane(Name1,A1,B1,C1,D1),
        A > A1-0.001, A < A1+0.001,
        B > B1-0.001, B < B1+0.001,
        C > C1-0.001, C < C1+0.001,
        D > D1-0.001, D < D1+0.001,
        retract(facelist(FL,List)),
        asserta(facelist(FL,[Name|List])),
        asserta(used(Name)),
        makelist1(FL);
        plane(Name,A,B,C,D),
        not(used(Name)),
        facelist(FL,List),
        List = [Name1|_],
        plane(Name1,A1,B1,C1,D1),
        A2 = -1*A1, B2= -1*B1, C2 = -1*C1, D2 = -1*D1,
        A > A2-0.001, A < A2+0.001,
        B > B2-0.001, B < B2+0.001,
        C > C2-0.001, C < C2+0.001,
        D > D2-0.001, D < D2+0.001,
        retract(facelist(FL,List)),
        asserta(facelist(FL,[Name|List])),
        asserta(used(Name)),
        makelist1(FL).

(15) makelist1(FL):-
        retractall(end),
        retractall(check(_,_)).
```

```
                plane(Name,A,B,C,D).
                not(used(Name)).
                facelist(FL,List).
                List = [Name1|_].
                plane(Name1,A1,B1,C1,D1).
                checknumber(A,B,C,D,A1,B1,C1,D1).
                retract(facelist(FL,List)).
                asserta(facelist(FL,[Name1|List])).
                asserta(used(Name)).
                makelist1(FL):
                plane(Name,_,_,_,_).
                not(used(Name)).
                facelist(OldFL,_).
                frontstr(1,OldFL,_,Nos).
                str_real(Nos,Nor).
                Nor1 = Nor + 1.
                str_real(Nos1,Nor1).
                concat("L",Nos1,NewFL).
                asserta(facelist(NewFL,[Name])).
                asserta(used(Name)).
                makelist1(NewFL):
                true.

(16) checknumber(A,B,C,D,A1,B1,C1,D1):-
                asserta(last(0)).
                checknumber1(A,A1),!.
                not(end).
                checknumber1(B,B1),!.
                not(end).
                checknumber1(C,C1),!.
                not(end).
                checknumber1(D,D1),!.
                not(end).
                retractall(last(_)),
                checknumber2.

(17) checknumber1(N1,N2):-
                N1 < 0 + 0.001, N1 > 0 - 0.001,
                N2 < 0 + 0.001, N2 > 0 - 0.001,
                last(Num),
                New = Num + 1,
                asserta(last(New)),
                asserta(check(New,0.0)):
                N1 < 0 + 0.001, N1 > 0 - 0.001,
                asserta(end):
                N2 < 0 + 0.001, N2 > 0 - 0.001,
                asserta(end):
                N = N1/N2,
                last(Num),
                New = Num + 1,
                asserta(last(New)),
                asserta(check(New,N)).

(18) checknumber2:-
                check(A,N1),
                N1 = 0,
                check(B,N2),
                B <> A,
                N2 = 0,
                check(C,N3),
                C <> B, C <> A,
                N3 <> 0,
                check(D,N4),
                D <> C, D <> B, D <> A,
                N4 <> 0,
                N3 < N4 + 0.001, N3 > N4 - 0.001;
                check(A,N1),
                N1 = 0,
                check(B,N2),
                B <> A,
                N2 <> 0,
                check(C,N3),
                C <> B, C <> A,
                N3 <> 0,
                check(D,N4),
                D <> C, D <> B, D <> A,
                N4 <> 0,
                N2 < N3 + 0.001, N2 > N3 - 0.001,
                N2 < N4 + 0.001, N2 > N4 - 0.001;
                check(A,N1),
                N1 <> 0,
                check(B,N2),
                B <> A,
                N2 <> 0,
                check(C,N3),
                C <> B, C <> A,
                N3 <> 0,
                check(D,N4),
                D <> C, D <> B, D <> A,
                N4 <> 0,
                N1 < N2 + 0.001, N1 > N2 - 0.001,
                N1 < N3 + 0.001, N1 > N3 - 0.001,
                N1 < N4 + 0.001, N1 > N4 - 0.001.

(19) nextlist:-
                facelist(Name,[H|T]).
                not(used(Name)).
                asserta(used(Name)).
                assertz(templist(Name,[H|T])).
                nextlist:
                retractall(used(_)).
                templist(Name,[H|T]).
                retract(templist(Name,[H|T])).
                asserta(templist(Name,T)).
                asserta(used(Name)).
                asserta(curvelist1("L1",[H|[]])).
                asserta(curvelist("L1",[])).
                asserta(counter(1)).
                asserta(lastlist([])).
                nextlist1(Name,"L1",[H|[]]),!.

(20) nextlist1(Name,Name1,[]):-
                lastlist(List).
                List = [_|_].
                retract(lastlist(List)).
                asserta(lastlist([])).
                nextlist1(Name,Name1,List);
                templist(Name,TList).
                TList = [H|T].
                retract(lastlist(_)).
                asserta(lastlist([])).
                retract(templist(Name,_)).
                asserta(templist(Name,T)).
                counter(Noi).
                Noi1 = Noi+1.
                asserta(counter(Noi1)).
                str_int(Nos1,Noi1).
                concat("L",Nos1,New).
                asserta(curvelist1(New,[H|[]])).
                asserta(curvelist(New,[])).
                nextlist1(Name,New,[H|[]]);
                templist(Name2,[H|T]).
                not(used(Name2)).
                retract(lastlist(_)).
                asserta(lastlist([])).
                retract(templist(Name2,[H|T])).
                asserta(templist(Name2,T)).
                asserta(used(Name2)).
                counter(Noi).
                Noi1 = Noi+1.
                str_int(Nos1,Noi1).
                concat("L",Nos1,New).
                asserta(curvelist1(New,[H|[]])).
                asserta(curvelist(New,[])).
                asserta(counter(Noi1)).
                nextlist1(Name2,New,[H|[]]);
                retractall(lastlist(_)).
                true.
(21) nextlist1(Name,Name1,[H|T]):-
                s2(H,[C1,C2,C3]).
                templist(Name,List1).
                checkcurve1(Name,List1,Name1,C1,S1).
                lastlist(L1).
                nextstep1(L1,S1).
                templist(Name,List2).
                checkcurve2(Name,List2,Name1,C2,S2).
                lastlist(L2).
                nextstep1(L2,S2).
                templist(Name,List3).
                checkcurve3(Name,List3,Name1,C3,S3).
                lastlist(L3).
                nextstep1(L3,S3).
                nextlist1(Name,Name1,T).

(22) nextstep1(L,S):-
                S <> "".
                retract(lastlist(L)).
                asserta(lastlist([S|L])):
                true.

(23) checkcurve1(_,[],_,_,"").
(24) checkcurve1(Name,[H|T],Name1,C1,S1):-
                templist(Name,List).
                member(H,List).
                s2(H,CList).
                member(C1,CList).
                subtract(H,List,NList).
                retract(templist(Name,_)).
                asserta(templist(Name,NList)).
                curvelist1(Name1,L).
                retract(curvelist1(Name1,_)).
                asserta(curvelist1(Name1,[H|L])).
                S1 = H.
                true:
                checkcurve1(Name,T,Name1,C1,S1).

(25) checkcurve2(_,[],_,_,"").
(26) checkcurve2(Name,[H|T],Name1,C2,S2):-
                templist(Name,List).
                member(H,List).
                s2(H,CList).
                member(C2,CList).
                subtract(H,List,NList).
                retract(templist(Name,_)).
                asserta(templist(Name,NList)).
                curvelist1(Name1,L).
                retract(curvelist1(Name1,_)).
                asserta(curvelist1(Name1,[H|L])).
                S2 = H.
                true:
                checkcurve2(Name,T,Name1,C2,S2).
```

```
(27) checkcurve3(_,[],_,_,"").
(28) checkcurve3(Name,[H|T],Name1,C3,S3):-
        templist(Name,List).
        member(H,List).
        s2(H,CList).
        member(C3,CList).
        subtract(H,List,NList).
        retract(templist(Name,_)).
        asserta(templist(Name,NList)).
        curvelist1(Name1,L).
        retract(curvelist1(Name1,_)).
        asserta(curvelist1(Name1,[H|L])).
        S3 = H.
        true;
        checkcurve3(Name,T,Name1,C3,S3).

(29) makecurve([],_,_):-
        curvelist1(FL,List).
        not(used1(FL)).
        asserta(used1(FL)).
        makecurve(List,List,FL):
        true.
(30) makecurve([H|T],List,FL):-
        makecurve1(H,List,FL).
        makecurve2(H,List,FL).
        makecurve3(H,List,FL).
        makecurve(T,List,FL).

(31) makecurve1(H,[],FL):-
        s2(H,[C1,_,_]).
        curvelist(FL,List).
        not(member(C1,List)).
        retract(curvelist(FL,_)).
        asserta(curvelist(FL,[C1|List])):
        true.
(32) makecurve1(H,[H1|T],FL):-
        H <> H1.
        s2(H,[C1,_,_]).
        s2(H1,List).
        not(member(C1,List)).
        makecurve1(H,T,FL):
        H = H1.
        makecurve1(H,T,FL):
        H <> H1. true.

(33) makecurve2(H,[],FL):-
        s2(H,[_,C2,_]).
        curvelist(FL,List).
        not(member(C2,List)).
        retract(curvelist(FL,_)).
        asserta(curvelist(FL,[C2|List])):
        true.
(34) makecurve2(H,[H1|T],FL):-
        H <> H1.
        s2(H,[_,C2,_]).
        s2(H1,List).
        not(member(C2,List)).
        makecurve2(H,T,FL):
        H = H1.
        makecurve2(H,T,FL):
        H <> H1. true.

(35) makecurve3(H,[],FL):-
        s2(H,[_,_,C3]).
        curvelist(FL,List).
        not(member(C3,List)).
        retract(curvelist(FL,_)).
        asserta(curvelist(FL,[C3|List])):
        true.
(36) makecurve3(H,[H1|T],FL):-
        H <> H1.
        s2(H,[_,_,C3]).
        s2(H1,List).
        not(member(C3,List)).
        makecurve3(H,T,FL):
        H = H1.
        makecurve3(H,T,FL):
        H <> H1. true.
(37) checksolid(_,[]):-
        curvelist(C,List).
        not(used(C)).
        asserta(used(C)).
        checksolid(C,List),!:
        asserta(solid("S1")).
(38) checksolid(C1,[H1|T1]):-
        not(used1(H1)).
        checksolid1(C1,[H1|T1]),!:
        checksolid(C1,T1).

(39) checksolid1(C1,[H1|T1]):-
        curvelist(C2,List).
        not(used(C2)).
        member(H1,List).
        asserta(used1(H1)).
        checksolid(C1,T1),!:
        true.
```

A2.3 CONTACT SURFACE DETECTION

```
(1) makechild([]):-
        retractall(used(_)).
        checksurface.!.
        phoundary.!.
        shoundary.!.
        makeref(point.!.
        poboundary.!.
        pre_soboundary.!.
        interpobsob.!.
        intersobsob.!.
        removecurve.!.
        pribackup.!.
        secbackup.!.
        cleanup.!.
        groupscurve.!.
        grouppcurve.!.
        secshared.!.
        makefreesurf.!.
        freesurfsib.!.
        insidesib.!.
        overlapsurf.!.
        insideconsurf.!.
(2) makechild([H|T]):-
        H = e(ID,_,_,_,_,_,_,Type).
        getname(ID,Name).
        counter1(Num).
        New = Num + 1.
        retract(counter1(Num)).
        asserta(counter1(New)).
        str_int(News,New).
        concat(News,Name,Group).
        asserta(group(Group,Type)).
        concat("C:\\Jason\\data\\",Name,Newfile).
        consult(NewFile,surface).
        makechild1(H,Group),!.
        changechild1(Group),!.
        childplane(Group),!.
        childlist(Group),!.
        compareplane(Group),!.
        retractall(used(_)).
        changechild2(Group),!.
        childlist2(Group),!.
        retractall(counter(_)).
        retractall(used(_)).
        makechild_next(Group,T).

(3) makechild_next(Group,T):-
        scurvelist1(FL,Group,List).
        retractall(used1(_)).
        asserta(used(FL)).
        makescurve(List,Group,List,FL),!.
        retractall(used(_)).
        retractall(used1(_)).
        makechild(T):
        makechild([]).

(4) makechild1(H,Group):-
        H = e(_,_,_,_,_,Rx,Ry,Rz,_).
        Rxr = Rx*3.14159265359/180.
        Ryr = Ry*3.14159265359/180.
        Rzr = Rz*3.14159265359/180.
        rotate_x(Rxr,Group),!.
        rotate_y(Ryr,Group),!.
        rotate_z(Rzr,Group),!.
        makechild2(H,Group),!.

(5) rotate_x(Rx,Group):-
        Rx = 0.
        s(X1,Y1,Z1,X2,Y2,Z2,X3,Y3,Z3).
        retract(s(X1,Y1,Z1,X2,Y2,Z2,X3,Y3,Z3)).
        asserta(sc1(Group,X1,Y1,Z1,X2,Y2,Z2,X3,Y3,Z3)).
        rotate_x(Rx,Group):
        s(X1,Y1,Z1,X2,Y2,Z2,X3,Y3,Z3).
        checkangle(Z1,Y1,A),!.
        fixangle(Z1,Y1,A,A1),!.
        B = A1 + Rx.
        L1 = sqrt(Y1*Y1 + Z1*Z1).
        Ny1 = L1*cos(B).
        Nz1 = L1*sin(B).
        checkangle(Z2,Y2,C),!.
        fixangle(Z2,Y2,C,C1),!.
        D = C1 + Rx.
        L2 = sqrt(Y2*Y2 + Z2*Z2).
        Ny2 = L2*cos(D).
        Nz2 = L2*sin(D).
        checkangle(Z3,Y3,E),!.
        fixangle(Z3,Y3,E,E1),!.
        F = E1 + Rx.
        L3 = sqrt(Y3*Y3 + Z3*Z3).
        Ny3 = L3*cos(F).
        Nz3 = L3*sin(F).
        retract(s(X1,Y1,Z1,X2,Y2,Z2,X3,Y3,Z3)).
        asserta(sc1(Group,X1,Ny1,Nz1,X2,Ny2,Nz2,X3,Ny3,Nz3)).
        rotate_x(Rx,Group):
```

```
        true.

(6) rotate_y(Ry,Group):-
        Ry = 0.
        sc1(Group.X1,Y1,Z1,X2,Y2,Z2,X3,Y3,Z3).
        retract(sc1(Group.X1,Y1,Z1,X2,Y2,Z2,X3,Y3,Z3)).
        asserta(sc2(Group.X1,Y1,Z1,X2,Y2,Z2,X3,Y3,Z3)).
        rotate_y(Ry,Group):
        sc1(Group.X1,Y1,Z1,X2,Y2,Z2,X3,Y3,Z3).
        checkangle(X1,Z1,A),!.
        fixangle(X1,Z1,A,A1),!.
        B = A1 + Ry.
        L1 = sqrt(Z1*Z1 + X1*X1).
        Nz1 = L1*cos(B).
        Nx1 = L1*sin(B).
        checkangle(X2,Z2,C),!.
        fixangle(X2,Z2,C,C1),!.
        D = C1 + Ry.
        L2 = sqrt(Z2*Z2 + X2*X2).
        Nz2 = L2*cos(D).
        Nx2 = L2*sin(D).
        checkangle(X3,Z3,E),!.
        fixangle(X3,Z3,E,E1),!.
        F = E1 + Ry.
        L3 = sqrt(Z3*Z3 + X3*X3).
        Nz3 = L3*cos(F).
        Nx3 = L3*sin(F).
        retract(sc1(Group.X1,Y1,Z1,X2,Y2,Z2,X3,Y3,Z3)).
        asserta(sc2(Group.Nx1,Y1,Nz1,Nx2,Y2,Nz2,Nx3,Y3,Nz3)).
        rotate_y(Ry,Group):
        true.

(7) rotate_z(Rz,Group):-
        Rz = 0.
        sc2(Group.X1,Y1,Z1,X2,Y2,Z2,X3,Y3,Z3).
        retract(sc2(Group.X1,Y1,Z1,X2,Y2,Z2,X3,Y3,Z3)).
        asserta(sc1(Group.X1,Y1,Z1,X2,Y2,Z2,X3,Y3,Z3)).
        rotate_z(Rz,Group):
        sc2(Group.X1,Y1,Z1,X2,Y2,Z2,X3,Y3,Z3).
        checkangle(Y1,X1,A),!.
        fixangle(Y1,X1,A,A1),!.
        B = A1 + Rz.
        L1 = sqrt(X1*X1 + Y1*Y1).
        Nx1 = L1*cos(B).
        Ny1 = L1*sin(B).
        checkangle(Y2,X2,C),!.
        fixangle(Y2,X2,C,C1),!.
        D = C1 + Rz.
        L2 = sqrt(X2*X2 + Y2*Y2).
        Nx2 = L2*cos(D).
        Ny2 = L2*sin(D).
        checkangle(Y3,X3,E),!.
        fixangle(Y3,X3,E,E1),!.
        F = E1 + Rz.
        L3 = sqrt(X3*X3 + Y3*Y3).
        Nx3 = L3*cos(F).
        Ny3 = L3*sin(F).
        retract(sc2(Group.X1,Y1,Z1,X2,Y2,Z2,X3,Y3,Z3)).
        asserta(sc1(Group.Nx1,Ny1,Z1,Nx2,Ny2,Z2,Nx3,Ny3,Z3)).
        rotate_z(Rz,Group):
        true.

(8) checkangle(A,B,C):-
        B <> 0.
        C = arctan(A/B):
        B = 0.
        A > 0.
        C = 3.141592653359/2:
        B = 0.
        A < 0.
        C = 3*3.141592653359/2:
        A = 0.
        C = 0.

(9) fixangle(A1,A2,A,New):-
        A1 < 0.
        A2 < 0.
        New = A + 3.141592653359:
        A1 > 0.
        A2 < 0.
        New = A + 3.141592653359:
        New = A.

(10) makechild2(H,Group):-
        H = e(_,Tx,Ty,Tz,_,_,_,_).
        sc1(Group.X1,Y1,Z1,X2,Y2,Z2,X3,Y3,Z3).
        Nx1 = X1 + Tx.
        Ny1 = Y1 + Ty.
        Nz1 = Z1 + Tz.
        Nx2 = X2 + Tx.
        Ny2 = Y2 + Ty.
        Nz2 = Z2 + Tz.
        Nx3 = X3 + Tx.
        Ny3 = Y3 + Ty.
        Nz3 = Z3 + Tz.
        retract(sc1(Group.X1,Y1,Z1,X2,Y2,Z2,X3,Y3,Z3)).
        asserta(sc2(Group.Nx1,Ny1,Nz1,Nx2,Ny2,Nz2,Nx3,Ny3,Nz3)).
        makechild2(H,Group):
        true.

(11) changechild1(Group):-
        sc2(Group.X1,Y1,Z1,X2,Y2,Z2,X3,Y3,Z3).
        checkpoint(X1,Y1,Z1,Xa,Ya,Za),!.
        checkpoint(X2,Y2,Z2,Xb,Yb,Zb),!.
        checkpoint(X3,Y3,Z3,Xc,Yc,Zc),!.
        retract(sc2(Group.X1,Y1,Z1,X2,Y2,Z2,X3,Y3,Z3)).
        assertz(sc1(Group,Xa,Ya,Za,Xb,Yb,Zb,Xc,Yc,Zc)).
        checkforsurf1(LastName.Group).
        frontstr(1,LastName,_,Nos).
        str_real(Nos,Nor).
        Nor1 = Nor + 1.
        str_real(Nos1,Nor1).
        concat("s",Nos1,NewName).
        asserta(ss1(NewName,Group,p(Xa,Ya,Za),p(Xb,Yb,Zb),p(Xc,Yc,Zc))).
        changechild1(Group):
        true.

(12) checkpoint(X,Y,Z,A,B,C):-
        s1(Name,p(Xa,Ya,Za),p(Xb,Yb,Zb),p(Xc,Yc,Zc)).
        retract(s1(Name,p(Xa,Ya,Za),p(Xb,Yb,Zb),p(Xc,Yc,Zc))).
        asserta(s1_temp(Name,p(Xa,Ya,Za),p(Xb,Yb,Zb),p(Xc,Yc,Zc))).
        checkpoint1(X,Y,Z,A,B,C,Xa,Ya,Za,Xb,Yb,Zb,Xc,Yc,Zc),!:
        ss1(NewName,Group,p(Xa,Ya,Za),p(Xb,Yb,Zb),p(Xc,Yc,Zc)).
        retract(ss1(NewName,Group,p(Xa,Ya,Za),p(Xb,Yb,Zb),p(Xc,Yc,Zc))).
        asserta(ss1_temp(NewName,Group,p(Xa,Ya,Za),p(Xb,Yb,Zb),p(Xc,Yc
        ,Zc))).
        checkpoint2(X,Y,Z,A,B,C,Xa,Ya,Za,Xb,Yb,Zb,Xc,Yc,Zc),!:
        A = X.
        B = Y.
        C = Z.
        revert.
        revert2.

(13) checkpoint1(X,Y,Z,A,B,C,Xa,Ya,Za,Xb,Yb,Zb,Xc,Yc,Zc):-
        X > Xa-0.001, X < Xa+0.001.
        Y > Ya-0.001, Y < Ya+0.001.
        Z > Za-0.001, Z < Za+0.001.
        A = Xa.
        B = Ya.
        C = Za.
        revert.
        revert2:
        X > Xb-0.001, X < Xb+0.001.
        Y > Yb-0.001, Y < Yb+0.001.
        Z > Zb-0.001, Z < Zb+0.001.
        A = Xb.
        B = Yb.
        C = Zb.
        revert.
        revert2:
        X > Xc-0.001, X < Xc+0.001.
        Y > Yc-0.001, Y < Yc+0.001.
        Z > Zc-0.001, Z < Zc+0.001.
        A = Xc.
        B = Yc.
        C = Zc.
        revert.
        revert2:
        checkpoint(X,Y,Z,A,B,C).

(14) checkpoint2(X,Y,Z,A,B,C,Xa,Ya,Za,Xb,Yb,Zb,Xc,Yc,Zc):-
        X > Xa-0.001, X < Xa+0.001.
        Y > Ya-0.001, Y < Ya+0.001.
        Z > Za-0.001, Z < Za+0.001.
        A = Xa.
        B = Ya.
        C = Za.
        revert.
        revert2:
        X > Xb-0.001, X < Xb+0.001.
        Y > Yb-0.001, Y < Yb+0.001.
        Z > Zb-0.001, Z < Zb+0.001.
        A = Xb.
        B = Yb.
        C = Zb.
        revert.
        revert2:
        X > Xc-0.001, X < Xc+0.001.
        Y > Yc-0.001, Y < Yc+0.001.
        Z > Zc-0.001, Z < Zc+0.001.
        A = Xc.
        B = Yc.
        C = Zc.
        revert.
        revert2:
        checkpoint(X,Y,Z,A,B,C).

(15) revert:-
        s1_temp(Name,p(Xa,Ya,Za),p(Xb,Yb,Zb),p(Xc,Yc,Zc)).
        retract(s1_temp(Name,p(Xa,Ya,Za),p(Xb,Yb,Zb),p(Xc,Yc,Zc))).
        asserta(s1(Name,p(Xa,Ya,Za),p(Xb,Yb,Zb),p(Xc,Yc,Zc))).
        revert:
        true.

(16) revert2:-
        ss1_temp(Name,Group,p(Xa,Ya,Za),p(Xb,Yb,Zb),p(Xc,Yc,Zc)).
        retract(ss1_temp(Name,Group,p(Xa,Ya,Za),p(Xb,Yb,Zb),p(Xc,Yc,Zc))
        ).
        asserta(ss1(Name,Group,p(Xa,Ya,Za),p(Xb,Yb,Zb),p(Xc,Yc,Zc))).
        revert2:
        true.
```

```
(17) checkforsurf1(LastName,Group):-
        ss1(LastName,Group,_,_,_):
        LastName = "s0".

(18) childplane(Group):-
        ss1(Name,Group,p(X1,Y1,Z1),p(X2,Y2,Z2),p(X3,Y3,Z3)),
        not(splane(Name,Group,_,_,_,_)),
        A = X1-X2, B = Y1-Y2, C = Z1-Z2,
        D = X1-X3, E = Y1-Y3, F = Z1-Z3,
        I = (B*F)-(C*E),
        J = -1*((A*F)-(C*D)),
        K = (A*E)-(B*D),
        H = I*(-X1)+J*(-Y1)+K*(-Z1),
        childplane(Name,Group,I,J,K,H);
        true.

(19) childplane(Name,Group,I,J,K,H):-
        H < 0+0.001,
        H > 0-0.001,
        childplane1(Name,Group,I,J,K,H);
        childplane2(Name,Group,I,J,K,H).

(20) childplane1(Name,Group,I,J,K,H):-
        I < 0+0.001,
        I > 0-0.001,
        J < 0+0.001,
        J > 0-0.001,
        K <> 0,
        asserta(splane(Name,Group,I,J,1.0,H)),
        childplane(Group);
        I < 0+0.001,
        I > 0-0.001,
        J <> 0,
        K < 0+0.001,
        K > 0-0.001,
        asserta(splane(Name,Group,I,1.0,K,H)),
        childplane(Group);
        I <> 0,
        J < 0+0.001,
        J > 0-0.001,
        K < 0+0.001,
        K > 0-0.001,
        asserta(splane(Name,Group,1.0,J,K,H)),
        childplane(Group);
        asserta(splane(Name,Group,I,J,K,H)),
        childplane(Group).

(21) childplane2(Name,Group,I,J,K,H):-
        I < 0+0.001,
        I > 0-0.001,
        J < 0+0.001,
        J > 0-0.001,
        K <> 0,
        H1 = H/K,
        asserta(splane(Name,Group,I,J,1.0,H1)),
        childplane(Group);
        I < 0+0.001,
        I > 0-0.001,
        J <> 0,
        K < 0+0.001,
        K > 0-0.001,
        H1 = H/J,
        asserta(splane(Name,Group,I,1.0,K,H1)),
        childplane(Group);
        I <> 0,
        J < 0+0.001,
        J > 0-0.001,
        K < 0+0.001,
        K > 0-0.001,
        H1 = H/I,
        asserta(splane(Name,Group,1.0,J,K,H1)),
        childplane(Group);
        asserta(splane(Name,Group,I,J,K,H)),
        childplane(Group).

(22) childlist(Group):-
        splane(Name,Group,_,_,_,_),
        asserta(sfacelist("L1",Group,[Name])),
        asserta(used(Name)),
        childlist1("L1",Group),!.

(23) childlist1(FL,Group):-
        splane(Name,Group,A,B,C,D),
        not(used(Name)),
        sfacelist(FL,Group,List),
        List = [Name1|_],
        splane(Name1,Group,A1,B1,C1,D1),
        A > A1-0.001, A < A1+0.001,
        B > B1-0.001, B < B1+0.001,
        C > C1-0.001, C < C1+0.001,
        D > D1-0.001, D < D1+0.001,
        retract(sfacelist(FL,Group,List)),
        asserta(sfacelist(FL,Group,[Name|List])),
        asserta(used(Name)),
        childlist1(FL,Group);
        splane(Name,Group,A,B,C,D),
        not(used(Name)),
        sfacelist(FL,Group,List),
        List = [Name1|_],
        splane(Name1,Group,A1,B1,C1,D1),
        A2 = -1*A1, B2= -1*B1, C2 = -1*C1, D2 = -1*D1,
        A > A2-0.001, A < A2+0.001,
        B > B2-0.001, B < B2+0.001,
        C > C2-0.001, C < C2+0.001,
        D > D2-0.001, D < D2+0.001,
        retract(sfacelist(FL,Group,List)),
        asserta(sfacelist(FL,Group,[Name|List])),
        asserta(used(Name)),
        childlist1(FL,Group).
(24) childlist1(FL,Group):-
        retractall(end),
        retractall(check(_,_)),
        splane(Name,Group,A,B,C,D),
        not(used(Name)),
        sfacelist(FL,Group,List),
        List = [Name1|_],
        splane(Name1,Group,A1,B1,C1,D1),
        checknumber(A,B,C,D,A1,B1,C1,D1),
        retract(sfacelist(FL,Group,List)),
        asserta(sfacelist(FL,Group,[Name|List])),
        asserta(used(Name)),
        childlist1(FL,Group);
        splane(Name,Group,_,_,_,_),
        not(used(Name)),
        sfacelist(OldFL,Group,_),
        frontstr(1,OldFL,_,Nos),
        str_real(Nos,Nor),
        Nor1 = Nor + 1,
        str_real(Nos1,Nor1),
        concat("L",Nos1,NewFL),
        asserta(sfacelist(NewFL,Group,[Name])),
        asserta(used(Name)),
        childlist1(NewFL,Group);
        true.

(25) compareplane(Group):-
        sfacelist(Name,Group,[HIT]),
        retract(sfacelist(Name,Group,[HI_])),
        asserta(tsfacelist(Name,Group,[HIT])),
        compare1(Name,Group,[HIT]),!,
        compareplane(Group);
        revert1.

(26) compare1(Name,Group,[HIT]):-
        splane(H,Group,A,B,C,D),
        plane(_,A1,B1,C1,D1),
        A > A1-0.001, A < A1+0.001,
        B > B1-0.001, B < B1+0.001,
        C > C1-0.001, C < C1+0.001,
        D > D1-0.001, D < D1+0.001;
        splane(H,Group,A,B,C,D),
        plane(_,A1,B1,C1,D1),
        A2 = -1*A1, B2= -1*B1, C2 = -1*C1, D2 = -1*D1,
        A > A2-0.001, A < A2+0.001,
        B > B2-0.001, B < B2+0.001,
        C > C2-0.001, C < C2+0.001,
        D > D2-0.001, D < D2+0.001;
        splane(H,Group,A,B,C,D),
        plane(_,A1,B1,C1,D1),
        retractall(end),
        retractall(check(_,_)),
        checknumber(A,B,C,D,A1,B1,C1,D1);
        retract(tsfacelist(Name,Group,_)),
        removeface(Group,[HIT]),!.

(27) removeface(_,[]).
(28) removeface(Group,[HIT]):-
        ss1(H,Group,p(Xa,Ya,Za),p(Xb,Yb,Zb),p(Xc,Yc,Zc)),
        retract(ss1(H,Group,p(Xa,Ya,Za),p(Xb,Yb,Zb),p(Xc,Yc,Zc))),
        retract(sc1(Group,Xa,Ya,Za,Xb,Yb,Zb,Xc,Yc,Zc)),
        removeface(Group,T).

(29) revert1:-
        tsfacelist(Name,Group,List),
        retract(tsfacelist(Name,Group,List)),
        asserta(sfacelist(Name,Group,List)),
        revert1;
        true.

(30) changechild2(Group):-
        ss1(Name,Group,p(X1,Y1,Z1),p(X2,Y2,Z2),p(X3,Y3,Z3)),
        not(used(Name)),
        asserta(used(Name)),
        asserta(ss2(Name,Group,[])),
        scurve1(Name,Group,X1,Y1,Z1,X2,Y2,Z2),
        scurve2(Name,Group,X2,Y2,Z2,X3,Y3,Z3),
        scurve3(Name,Group,X1,Y1,Z1,X3,Y3,Z3),
        changechild2(Group);
        retractall(used(_)),
        true.

(31) scurve1(Name,Group,X1,Y1,Z1,X2,Y2,Z2):-
        ss2(Name,Group,List),
        not(c(_,p(X1,Y1,Z1),p(X2,Y2,Z2))),
        not(c(_,p(X2,Y2,Z2),p(X1,Y1,Z1))),
        c(Last,_,_),
        frontstr(1,Last,_,Nos),
        str_real(Nos,Nor),
        Nor1 = Nor + 1,
        str_real(Nos1,Nor1),
        concat("c",Nos1,NewName),
        asserta(c(NewName,p(X1,Y1,Z1),p(X2,Y2,Z2))),
```

```
                retract(ss2(Name.Group.List)).
                asserta(ss2(Name.Group.[NewNamelList])):
                c(C.p(X1.Y1.Z1).p(X2.Y2.Z2)).
                retract(ss2(Name.Group.List)).
                asserta(ss2(Name.Group.[ClList])):
                c(C.p(X2.Y2.Z2).p(X1.Y1.Z1)).
                retract(ss2(Name.Group.List)).
                asserta(ss2(Name.Group.[ClList])).

(32) scurve2(Name.Group.X2.Y2.Z2.X3.Y3.Z3):-
                ss2(Name.Group.List).
                not(c(_.p(X2.Y2.Z2).p(X3.Y3.Z3))).
                not(c(_.p(X3.Y3.Z3).p(X2.Y2.Z2))).
                c(Last._._).
                frontstr(1.Last._.Nos).
                str_real(Nos.Nor).
                Nor1 = Nor + 1.
                str_real(Nos1.Nor1).
                concat("c".Nos1.NewName).
                asserta(c(NewName.p(X2.Y2.Z2).p(X3.Y3.Z3))).
                retract(ss2(Name.Group.List)).
                asserta(ss2(Name.Group.[NewNamelList])):
                c(C.p(X2.Y2.Z2).p(X3.Y3.Z3)).
                retract(ss2(Name.Group.List)).
                asserta(ss2(Name.Group.[ClList])):
                c(C.p(X3.Y3.Z3).p(X2.Y2.Z2)).
                retract(ss2(Name.Group.List)).
                asserta(ss2(Name.Group.[ClList])).

(33) scurve3(Name.Group.X1.Y1.Z1.X3.Y3.Z3):-
                ss2(Name.Group.List).
                not(c(_.p(X1.Y1.Z1).p(X3.Y3.Z3))).
                not(c(_.p(X3.Y3.Z3).p(X1.Y1.Z1))).
                c(Last._._).
                frontstr(1.Last._.Nos).
                str_real(Nos.Nor).
                Nor1 = Nor + 1.
                str_real(Nos1.Nor1).
                concat("c".Nos1.NewName).
                asserta(c(NewName.p(X1.Y1.Z1).p(X3.Y3.Z3))).
                retract(ss2(Name.Group.List)).
                asserta(ss2(Name.Group.[NewNamelList])):
                c(C.p(X1.Y1.Z1).p(X3.Y3.Z3)).
                retract(ss2(Name.Group.List)).
                asserta(ss2(Name.Group.[ClList])):
                c(C.p(X3.Y3.Z3).p(X1.Y1.Z1)).
                retract(ss2(Name.Group.List)).
                asserta(ss2(Name.Group.[ClList])).

(34) childlist2(Group):-
                sfacelist(Name.Group.[HIT]).
                not(used(Name)).
                asserta(used(Name)).
                assertz(stemplist(Name.Group.[HIT])).
                childlist2(Group):
                retractall(used(_)).
                stemplist(Name.Group.[HIT]).
                retract(stemplist(Name.Group.[HIT])).
                asserta(stemplist(Name.Group.T)).
                asserta(used(Name)).
                asserta(scurvelist1("L1".Group.[H|[]])).
                splane(H.Group.A.B.C.D).
                asserta(scurvelist("L1".Group.[].pl(A.B.C.D))).
                asserta(counter(1)).
                asserta(slastlist([])).
                childlist3(Name.Group."L1".[H|[]]).!:
                true.

(35) childlist3(Name.Group.Name1.[]):-
                slastlist(List).
                List = [_|_].
                retract(slastlist(List)).
                asserta(slastlist([])).
                childlist3(Name.Group.Name1.List):
                stemplist(Name.Group.TList).
                TList = [HIT].
                retract(slastlist(_)).
                asserta(slastlist([])).
                retract(stemplist(Name.Group._)).
                asserta(stemplist(Name.Group.T)).
                counter(Noi).
                Noi1 = Noi+1.
                asserta(counter(Noi1)).
                str_int(Nos1.Noi1).
                concat("L".Nos1.New).
                asserta(scurvelist1(New.Group.[H[]])).
                splane(H.Group.A.B.C.D).
                asserta(scurvelist(New.Group.[].pl(A.B.C.D))).
                childlist3(Name.Group.New.[H|[]]):
                stemplist(Name2.Group.[HIT]).
                not(used(Name2)).
                retract(slastlist(_)).
                asserta(slastlist([])).
                retract(stemplist(Name2.Group.[HIT])).
                asserta(stemplist(Name2.Group.T)).
                asserta(used(Name2)).
                counter(Noi).
                Noi1 = Noi+1.
                str_int(Nos1.Noi1).
                concat("L".Nos1.New).
                asserta(scurvelist1(New.Group.[H|[]])).
                splane(H.Group.A.B.C.D).
                asserta(scurvelist(New.Group.[].pl(A.B.C.D))).
                asserta(counter(Noi1)).
                childlist3(Name2.Group.New.[H|[]]):
                true.
(36) childlist3(Name.Group.Name1.[HIT]):-
                ss2(H.Group.[C1.C2.C3]).
                stemplist(Name.Group.List1).
                checkscurve1(Name.Group.List1.Name1.C1.S1).
                slastlist(L1).
                nextstep2(L1.S1).
                stemplist(Name.Group.List2).
                checkscurve2(Name.Group.List2.Name1.C2.S2).
                slastlist(L2).
                nextstep2(L2.S2).
                stemplist(Name.Group.List3).
                checkscurve3(Name.Group.List3.Name1.C3.S3).
                slastlist(L3).
                nextstep2(L3.S3).
                childlist3(Name.Group.Name1.T).

(37) nextstep2(L.S):-
                S <> "".
                retract(slastlist(L)).
                asserta(slastlist([SIL])):
                true.

(38) checkscurve1(_._.[]._._."").
(39) checkscurve1(Name.Group.[HIT].Name1.C1.S1):-
                stemplist(Name.Group.List).
                member(H.List).
                ss2(H.Group.CList).
                member(C1.CList).
                subtract(H.List.NList).
                retract(stemplist(Name.Group._)).
                asserta(stemplist(Name.Group.NList)).
                scurvelist1(Name1.Group.L).
                retract(scurvelist1(Name1.Group._)).
                asserta(scurvelist1(Name1.Group.[HIL])).
                S1 = H.
                true:
                checkscurve1(Name.Group.T.Name1.C1.S1).

(40) checkscurve2(_._.[]._._."").
(41) checkscurve2(Name.Group.[HIT].Name1.C2.S2):-
                stemplist(Name.Group.List).
                member(H.List).
                ss2(H.Group.CList).
                member(C2.CList).
                subtract(H.List.NList).
                retract(stemplist(Name.Group._)).
                asserta(stemplist(Name.Group.NList)).
                scurvelist1(Name1.Group.L).
                retract(scurvelist1(Name1.Group._)).
                asserta(scurvelist1(Name1.Group.[HIL])).
                S2 = H.
                true:
                checkscurve2(Name.Group.T.Name1.C2.S2).

(42) checkscurve3(_._.[]._._."").
(43) checkscurve3(Name.Group.[HIT].Name1.C3.S3):-
                stemplist(Name.Group.List).
                member(H.List).
                ss2(H.Group.CList).
                member(C3.CList).
                subtract(H.List.NList).
                retract(stemplist(Name.Group._)).
                asserta(stemplist(Name.Group.NList)).
                scurvelist1(Name1.Group.L).
                retract(scurvelist1(Name1.Group._)).
                asserta(scurvelist1(Name1.Group.[HIL])).
                S3 = H.
                true:
                checkscurve3(Name.Group.T.Name1.C3.S3).

(44) makescurve([].Group._._):-
                scurvelist1(FL.Group.List).
                not(used1(FL)).
                asserta(used1(FL)).
                makescurve(List.Group.List.FL):
                true.
(45) makescurve([HIT].Group.List.FL):-
                makescurve1(H.Group.List.FL).
                makescurve2(H.Group.List.FL).
                makescurve3(H.Group.List.FL).
                makescurve(T.Group.List.FL).

(47) makescurve1(H.Group.[].FL):-
                ss2(H.Group.[C1._._]).
                scurvelist(FL.Group.List.Plane).
                not(member(C1.List)).
                retract(scurvelist(FL.Group._._)).
                asserta(scurvelist(FL,Group.[C1|List].Plane)):
                true.
(48) makescurve1(H.Group.[H1|T].FL):-
                H <> H1.
                ss2(H.Group.[C1._._]).
                ss2(H1.Group.List).
                not(member(C1.List)).
                makescurve1(H.Group.T.FL):
                H = H1.
```

```
        makescurve1(H,Group,T,FL):
        H <> H1, true.

(49) makescurve2(H,Group,[],FL):-
        ss2(H,Group,[_,C2,_]).
        scurvelist(FL,Group,List,Plane).
        not(member(C2,List)).
        retract(scurvelist(FL,Group,_,_)).
        asserta(scurvelist(FL,Group,[C2|List],Plane)):
        true.
(50) makescurve2(H,Group,[H1|T],FL):-
        H <> H1.
        ss2(H,Group,[_,C2,_]).
        ss2(H1,Group,List).
        not(member(C2,List)).
        makescurve2(H,Group,T,L):
        H = H1.
        makescurve2(H,Group,T,L):
        H <> H1, true.

(51) makescurve3(H,Group,[],FL):-
        ss2(H,Group,[_,_,C3]).
        scurvelist(FL,Group,List,Plane).
        not(member(C3,List)).
        retract(scurvelist(FL,Group,_,_)).
        asserta(scurvelist(FL,Group,[C3|List],Plane)):
        true.
(52) makescurve3(H,Group,[H1|T],FL):-
        H <> H1.
        ss2(H,Group,[_,_,C3]).
        ss2(H1,Group,List).
        not(member(C3,List)).
        makescurve3(H,Group,T,FL):
        H = H1.
        makescurve3(H,Group,T,FL):
        H <> H1, true.

(53) checksurface:-
        curvelist(Name,_).
        not(used(Name)).
        asserta(used(Name)).
        curvelist1(Name,[S1|_]).
        plane(S1,A1,B1,C1,D1).
        asserta(status(Name,[],pl(A1,B1,C1,D1))).
        checksurf1(Name),!.
        checksurface:
        retractall(used(_)).
        true.

(54) checksurf1(Name1):-
        status(Name1,List1,pl(A1,B1,C1,D1)).
        scurvelist(Name2,Group,_,pl(A2,B2,C2,D2)).
        not(pair(Name2,Group)).
        A1 > A2-0.001, A1 < A2+0.001,
        B1 > B2-0.001, B1 < B2+0.001,
        C1 > C2-0.001, C1 < C2+0.001,
        D1 > D2-0.001, D1 < D2+0.001,
        asserta(pair(Name2,Group)).
        retract(status(Name1,List1,pl(A1,B1,C1,D1))).
        asserta(status(Name1,[pair(Name2,Group)|List1],pl(A1,B1,C1,D1))).
        checksurf1(Name1):
        status(Name1,List1,pl(A1,B1,C1,D1)).
        scurvelist(Name2,Group,_,pl(A2,B2,C2,D2)).
        not(pair(Name2,Group)).
        A3 = -1*A2,
        B3 = -1*B2,
        C3 = -1*C2,
        D3 = -1*D2,
        A1 > A3-0.001, A1 < A3+0.001,
        B1 > B3-0.001, B1 < B3+0.001,
        C1 > C3-0.001, C1 < C3+0.001,
        D1 > D3-0.001, D1 < D3+0.001,
        asserta(pair(Name2,Group)).
        retract(status(Name1,List1,pl(A1,B1,C1,D1))).
        asserta(status(Name1,[pair(Name2,Group)|List1],pl(A1,B1,C1,D1))).
        checksurf1(Name1):
        status(Name1,List1,pl(A1,B1,C1,D1)).
        scurvelist(Name2,Group,_,pl(A2,B2,C2,D2)).
        not(pair(Name2,Group)).
        retractall(end).
        retractall(check(_,_)).
        checknumber(A1,B1,C1,D1,A2,B2,C2,D2).
        asserta(pair(Name2,Group)).
        retract(status(Name1,List1,pl(A1,B1,C1,D1))).
        asserta(status(Name1,[pair(Name2,Group)|List1],pl(A1,B1,C1,D1))).
        checksurf1(Name1):
        retractall(end).
        retractall(check(_,_)).
        retractall(pair(_,_)).

(55) pboundary:-
        status(Name,L,_).
        not(used(Name)).
        L = [_|_].
        curvelist(Name,[H|T]).
        asserta(used(Name)).
        asserta(pblist(Name,[])).
        pboundary1(Name,[H|T]),!.
        pboundary:
        status(Name,L,_).
        not(used(Name)).
        L = [].
        curvelist(Name,[H|T]).
        asserta(used(Name)).
        asserta(freesurf(Name,Name,[H|T])).
        pboundary:
        retractall(used(_)).
        true.

(56) pboundary1(_,[]).
(57) pboundary1(Name,[H|T]):-
        asserta(pb(Name,[H|[]])).
        c(H,P1,P2).
        P2 = p(X,Y,Z).
        getbig(X,Y,Z).
        pboundary2(Name,T,T,P1,P2),!.

(58) pboundary2(Name,[],List,P1,P2):-
        List = [_|_].
        pboundary2(Name,List,List,P1,P2):
        true.
(59) pboundary2(Name,[H|T],List,P1,P2):-
        c(H,P2,P3).
        P3 = p(X,Y,Z).
        getbig(X,Y,Z).
        pboundary3(Name,[H|T],List,P1,P3),!:
        c(H,P3,P2).
        P3 = p(X,Y,Z).
        getbig(X,Y,Z).
        pboundary3(Name,[H|T],List,P1,P3),!:
        not(c(H,P2,_)).
        not(c(H,_,P2)).
        pboundary2(Name,T,List,P1,P2).

(60) pboundary3(Name,[H|T],List,P1,P3):-
        P1 = P3,
        pb(Name,CL),
        retract(pb(Name,CL)),
        asserta(pb(Name,[H|CL])).
        subtract(H,List,NList).
        retract(pblist(Name,CLL)).
        asserta(pblist(Name,[[H|CL]|CLL])).
        pboundary1(Name,Nlist),!:
        pb(Name,CL).
        retract(pb(Name,CL)).
        asserta(pb(Name,[H|CL])).
        subtract(H,List,NList).
        pboundary2(Name,T,NList,P1,P3).

(61) sboundary:-
        scurvelist(Name,Group,List,_).
        not(pair(Name,Group)).
        asserta(pair(Name,Group)).
        asserta(sblist(Name,Group,[])).
        sboundary1(Name,Group,List),!.
        sboundary:
        retractall(pair(_,_)).
        true.

(62) sboundary1(_,_,[]).
(63) sboundary1(Name,Group,[H|T]):-
        asserta(sb(Name,Group,[H|[]])).
        c(H,P1,P2).
        sboundary2(Name,Group,T,T,P1,P2),!.

(64) sboundary2(Name,Group,[],List,P1,P2):-
        List = [_|_].
        sboundary2(Name,Group,List,List,P1,P2):
        true.
(65) sboundary2(Name,Group,[H|T],List,P1,P2):-
        c(H,P2,P3).
        sboundary3(Name,Group,[H|T],List,P1,P3),!:
        c(H,P3,P2).
        sboundary3(Name,Group,[H|T],List,P1,P3),!:
        not(c(H,P2,_)).
        not(c(H,_,P2)).
        sboundary2(Name,Group,T,List,P1,P2).

(66) sboundary3(Name,Group,[H|T],List,P1,P3):-
        P1 = P3,
        sb(Name,Group,CL).
        retract(sb(Name,Group,CL)).
        asserta(sb(Name,Group,[H|CL])).
        subtract(H,List,NList).
        retract(sblist(Name,Group,CLL)).
        asserta(sblist(Name,Group,[[H|CL]|CLL])).
        sboundary1(Name,Group,Nlist),!:
        sb(Name,Group,CL).
        retract(sb(Name,Group,CL)).
        asserta(sb(Name,Group,[H|CL])).
        subtract(H,List,NList).
        sboundary2(Name,Group,T,NList,P1,P3).

(67) getbig(A,B,C):-
        A1 = abs(A).
        B1 = abs(B).
        C1 = abs(C).
        asserta(num(A1)).
        asserta(num(B1)).
        asserta(num(C1)).
        num(N1).
        num(N2).
```

```
                N2 >= N1.
                num(N3).
                N3 >= N2.
                big(Num).
                Num < N3.
                retract(big(Num)).
                retractall(num(_)).
                asserta(big(N3)):
                true.

(68) makerefpoint:-
                status(Name,List,Plane).
                List = [_|_].
                not(used(Name)).
                asserta(used(Name)).
                reference1(Name,Plane).!.
                makerefpoint:
                retractall(used(_)).

(69) reference1(Name,Plane):-
                Plane = pl(A,B,C,D).
                A <> 0.
                B <> 0.
                C <> 0.
                big(Num).
                New = 5*Num.
                %New1 = 1.1*New.
                Z = (-1*D-A*New-B*New)/C.
                asserta(prp(Name,New,New,Z)).
(70) reference1(Name,Plane):-
                Plane = pl(A,B,C,D).
                A <> 0.
                B > 0-0.001.
                B < 0+0.001.
                C > 0-0.001.
                C < 0+0.001.
                big(Num).
                New = 5*Num.
                %New1 = 1.1*New.
                D1 = -1*D.
                asserta(prp(Name,D1,New,New)):
                Plane =pl(A,B,C,D).
                A > 0-0.001.
                A < 0+0.001.
                B <> 0.
                C > 0-0.001.
                C < 0+0.001.
                big(Num).
                New = 5*Num.
                %New1 = 1.1*New.
                D1 = -1*D.
                asserta(prp(Name,New,D1,New)):
                Plane = pl(A,B,C,D).
                A > 0-0.001.
                A < 0+0.001.
                B > 0-0.001.
                B < 0+0.001.
                C <> 0.
                big(Num).
                New = 5*Num.
                %New1 = 1.1*New.
                D1 = -1*D.
                asserta(prp(Name,New,New,D1)).
(71) reference1(Name,Plane):-
                Plane = pl(A,B,C,D).
                A > 0-0.001.
                A < 0+0.001.
                B <> 0.
                C <> 0.
                big(Num).
                New = 5*Num.
                %New1 = 1.1*New.
                Z = (-1*D-B*New)/C.
                asserta(prp(Name,New,New,Z)):
                Plane = pl(A,B,C,D).
                A <> 0.
                B > 0-0.001.
                B < 0+0.001.
                C <> 0.
                big(Num).
                New = 5*Num.
                %New1 = 1.1*New.
                X = (-1*D-C*New)/A.
                asserta(prp(Name,X,New,New)):
                Plane = pl(A,B,C,D).
                A <> 0.
                B <> 0.
                C > 0-0.001.
                C < 0+0.001.
                big(Num).
                New = 5*Num.
                %New1 = 1.1*New.
                Y = (-1*D-A*New)/B.
                asserta(prp(Name,New,Y,New)).

(72) poboundary:-
                pblist(Name,[H|T]).
                not(used(Name)).
                T = |_|_|.
                asserta(used(Name)).
                asserta(last(0)).
                poboundary1(Name,[H|T]).!.
                retractall(end).
                poboundary:
                pblist(Name,[H|T]).
                not(used(Name)).
                T = [].
                asserta(used(Name)).
                asserta(pob(Name,H)).
                poboundary:
                retractall(lastlist1(_)).
                retractall(c1(_,_,_)).
                retractall(counter(_)).
                retractall(last(_)).
                retractall(used(_)).

(73) poboundary1(Name,[H|T]):-
                H = [H1|_].
                c(HL,p(X1,Y1,Z1),_).
                prp(Name,X2,Y2,Z2).
                asserta(c1(HL,p(X1,Y1,Z1),p(X2,Y2,Z2))).
                poboundary2(Name,T,HL).!.

(74) poboundary2(Name,[],HL):-
                not(end).
                not(newprp).
                pblist(Name,[H|_]).
                not(pob(Name,H)).
                asserta(pob(Name,H)):
                not(newprp).
                end;
                newprp.
                retractall(counter(_)).
                retractall(newprp).
                retract(c1(HL,p(X1,Y1,Z1),p(X2,Y2,Z2))).
                retract(prp(Name,X2,Y2,Z2)).
                Nx = 1.10*X2.
                Ny = 1.15*Y2.
                Nz = 1.20*Z2.
                asserta(prp(Name,Nx,Ny,Nz)).
                asserta(c1(HL,p(X1,Y1,Z1),p(Nx,Ny,Nz))).
                lastlist1(List).
                poboundary2(Name,List,HL).
(75) poboundary2(Name,[H|T],HL):-
                not(end).
                not(newprp).
                asserta(counter(0)).
                poboundary3(Name,H,H,HL).!.
                retractall(counter(_)).
                retractall(lastlist1(_)).
                asserta(lastlist1([H|T])).
                poboundary2(Name,T,HL):
                not(newprp).
                end;
                newprp.
                retractall(counter(_)).
                retractall(newprp).
                retract(c1(HL,p(X1,Y1,Z1),p(X2,Y2,Z2))).
                retract(prp(Name,X2,Y2,Z2)).
                Nx = 1.10*X2.
                Ny = 1.15*Y2.
                Nz = 1.20*Z2.
                asserta(prp(Name,Nx,Ny,Nz)).
                asserta(c1(HL,p(X1,Y1,Z1),p(Nx,Ny,Nz))).
                lastlist1(List).
                poboundary2(Name,List,HL).

(76) poboundary3(Name,[],List1,_):-
                not(newprp).
                counter(Num).
                checknum(Num,Status).
                Status = "Odd".
                pblist(Name,List2).
                subtract1(List1,List2,NewList).
                asserta(end).
                asserta(pob(Name,List1)).
                piboundary(Name,NewList).!.
                retractall(ipoint(_,_,_)):
                not(newprp).
                counter(Num).
                checknum(Num,Status).
                Status = "Even".
                last(N).
                New = N + 1.
                str_int(News,New).
                concat("p",News,P).
                asserta(pib(Name,P,List1)).
                retract(last(N)).
                retractall(ipoint(_,_,_)).
                asserta(last(New)):
                newprp.
                retractall(ipoint(_,_,_)).
(77) poboundary3(Name,[H1|T1],List,HL):-
                not(newprp).
                c1(HL,p(X1,Y1,Z1),p(X2,Y2,Z2)).
                c(H1,p(X3,Y3,Z3),p(X4,Y4,Z4)).
                asserta(c2(H1,p(X3,Y3,Z3),p(X4,Y4,Z4))).
                intersection(X1,Y1,Z1,X2,Y2,Z2,X3,Y3,Z3,X4,Y4,Z4,"Ignore").!.
                retract(c2(H1,p(X3,Y3,Z3),p(X4,Y4,Z4))).
                retractall(end1).
                poboundary3(Name,T1,List,HL):
                newprp.
```

```
(78) pre_soboundary:-
        sblist(_,Group,_).
        not(gused(Group)).
        asserta(gused(Group)).
        soboundary(Group).
        pre_soboundary:
        retractall(lastlist1(_)).
        retractall(c1(_,_,_)).
        retractall(counter(_)).
        retractall(pair(_,_)).
        retractall(gused(_)).

(79) soboundary(Group):-
        sblist(Name,Group,[HIT]).
        not(pair(Name,Group)).
        T = [_|_].
        asserta(pair(Name,Group)).
        asserta(last(0)).
        soboundary1(Name,Group,[HIT]).
        retractall(end).
        soboundary(Group):
        sblist(Name,Group,[HIT]).
        not(pair(Name,Group)).
        T = [].
        asserta(pair(Name,Group)).
        asserta(sob(Name,Group,H)).
        soboundary(Group):
        retractall(last(_)).

(80) soboundary1(Name,Group,[HIT]):-
        H = [HL|_].
        c(HL,p(X1,Y1,Z1),_).
        status(Name1,List,_).
        member1(pair(Name,Group),List).
        prp(Name1,X2,Y2,Z2).
        asserta(c1(HL,p(X1,Y1,Z1),p(X2,Y2,Z2))).
        soboundary2(Name,Group,T,Name1,HL),!.

(81) soboundary2(Name,Group,[],Name1,HL):-
        not(newprp).
        not(end).
        sblist(Name,Group,[HI_]).
        not(sob(Name,Group,H)).
        asserta(sob(Name,Group,H)):
        not(newprp).
        end:
        newprp.
        retractall(counter(_)).
        retractall(newprp).
        retract(c1(HL,p(X1,Y1,Z1),p(X2,Y2,Z2))).
        retract(prp(Name1,X2,Y2,Z2)).
        Nx = 1.10*X2.
        Ny = 1.15*Y2.
        Nz = 1.20*Z2.
        asserta(prp(Name1,Nx,Ny,Nz)).
        asserta(c1(HL,p(X1,Y1,Z1),p(Nx,Ny,Nz))).
        lastlist1(List).
        soboundary2(Name,Group,List,Name1,HL).
(82) soboundary2(Name,Group,[HIT],Name1,HL):-
        not(newprp).
        not(end).
        asserta(counter(0)).
        soboundary3(Name,Group,H,H).
        retractall(counter(_)).
        retractall(lastlist1(_)).
        asserta(lastlist1([HIT])).
        soboundary2(Name,Group,T,Name1,HL):
        not(newprp).
        end:
        newprp.
        retractall(counter(_)).
        retractall(newprp).
        retract(c1(HL,p(X1,Y1,Z1),p(X2,Y2,Z2))).
        retract(prp(Name1,X2,Y2,Z2)).
        Nx = 1.10*X2.
        Ny = 1.15*Y2.
        Nz = 1.20*Z2.
        asserta(prp(Name1,Nx,Ny,Nz)).
        asserta(c1(HL,p(X1,Y1,Z1),p(Nx,Ny,Nz))).
        lastlist1(List).
        soboundary2(Name,Group,List,Name1,HL).

(83) soboundary3(Name,Group,[],List1):-
        not(newprp).
        counter(Num),
        checknum(Num,Status),
        Status = "Odd",
        sblist(Name,Group,List2),
        subtract1(List1,List2,NewList),
        asserta(end),
        asserta(sob(Name,Group,List1)),
        siboundary(Name,Group,NewList),!.
        retractall(ipoint(_,_,_)):
        not(newprp).
        counter(Num).
        checknum(Num,Status).
        Status = "Even".
        last(N).
        New = N + 1.
        str_int(News,New).
        concat("s",News,P).
        asserta(sib(Name,Group,P,List1)).
        retract(last(N)).
        retractall(ipoint(_,_,_)).
        asserta(last(New)):
        newprp.
        retractall(ipoint(_,_,_)).
(84) soboundary3(Name,Group,[H1IT1],List):-
        not(newprp).
        c(H1,p(X3,Y3,Z3),p(X4,Y4,Z4)).
        c1(_,p(X1,Y1,Z1),p(X2,Y2,Z2)).
        asserta(c2(H1,p(X3,Y3,Z3),p(X4,Y4,Z4))).
        intersection(X1,Y1,Z1,X2,Y2,Z2,X3,Y3,Z3,X4,Y4,Z4,"Ignore"),!.
        retractall(end1).
        retract(c2(H1,p(X3,Y3,Z3),p(X4,Y4,Z4))).
        soboundary3(Name,Group,T1,List):
        newprp.

(85) intersection(X1,Y1,Z1,X2,Y2,Z2,X3,Y3,Z3,X4,Y4,Z4,Option1):-
        asserta(counter1(0)).
        asserta(counter2(0)).
        xyplane(X1,Y1,X2,Y2,X3,Y3,X4,Y4).
        not(end1).
        empty1.
        asserta(counter1(0)).
        yzplane(Y1,Z1,Y2,Z2,Y3,Z3,Y4,Z4).
        not(end1).
        empty1.
        asserta(counter1(0)).
        zxplane(Z1,X1,Z2,X2,Z3,X3,Z4,X4).
        not(end1).
        empty1.
        checkresult.
        intersect_point(Option1).
        counter(Num).
        New = Num+1.
        retractall(colinear).
        retractall(counter(_)).
        retractall(counter2(_)).
        retractall(status1(_,_)).
        asserta(counter(New));
        retractall(xy(_,_,_)).
        retractall(yz(_,_,_)).
        retractall(zx(_,_,_)).
        retractall(colinear).
        retractall(counter1(_)).
        retractall(counter2(_)).
        retractall(curve1(_,_,_,_)).
        retractall(result(_,_,_)).
        retractall(status1(_,_)).

(86) empty1:-
        retractall(curve1(_,_,_,_)).
        retractall(result(_,_,_)).
        retractall(counter1(_)).

(87) xyplane(X1,Y1,X2,Y2,X3,Y3,X4,Y4):-
        DY1 = (Y2-Y1).
        DX1 = (X2-X1).
        checktype(DX1,DY1,Type1),!.
        maketype(X1,Y1,DX1,DY1,Type1),!.
        DY2 = (Y4-Y3).
        DX2 = (X4-X3).
        checktype(DX2,DY2,Type2),!.
        maketype(X3,Y3,DX2,DY2,Type2),!.
        findintersect,!.
        result(Type3,Xp,Yp).
        asserta(xy(Type3,Xp,Yp)).
        checkintersect(Type3,X1,Y1,X2,Y2,X3,Y3,X4,Y4),!.

(88) yzplane(Y1,Z1,Y2,Z2,Y3,Z3,Y4,Z4):-
        DZ1 = (Z2-Z1).
        DY1 = (Y2-Y1).
        checktype(DY1,DZ1,Type1),!.
        maketype(Y1,Z1,DY1,DZ1,Type1),!.
        DZ2 = (Z4-Z3).
        DY2 = (Y4-Y3).
        checktype(DY2,DZ2,Type2),!.
        maketype(Y3,Z3,DY2,DZ2,Type2),!.
        findintersect,!.
        result(Type3,Yp,Zp).
        asserta(yz(Type3,Yp,Zp)).
        checkintersect(Type3,Y1,Z1,Y2,Z2,Y3,Z3,Y4,Z4),!.

(89) zxplane(Z1,X1,Z2,X2,Z3,X3,Z4,X4):-
        DX1 = (X2-X1).
        DZ1 = (Z2-Z1).
        checktype(DZ1,DX1,Type1),!.
        maketype(Z1,X1,DZ1,DX1,Type1),!.
        DX2 = (X4-X3).
        DZ2 = (Z4-Z3).
        checktype(DZ2,DX2,Type2),!.
        maketype(Z3,X3,DZ2,DX2,Type2),!.
        findintersect,!.
        result(Type3,Zp,Xp).
        asserta(zx(Type3,Zp,Xp)).
        checkintersect(Type3,Z1,X1,Z2,X2,Z3,X3,Z4,X4),!.

(90) checktype(DA,DB,Type):-
        DA > 0.001,
        DB > 0.001,
```

```
                Type = "normal":
                DA < 0-0.001.
                DB < 0-0.001.
                Type = "normal":
                DA > 0.001.
                DB < 0-0.001.
                Type = "normal":
                DA < 0-0.001.
                DB > 0.001.
                Type = "normal".
(91) checktype(DA,DB,Type):-
                DB > 0.001.
                DA < 0.001.
                DA > 0-0.001.
                Type = "infinite":
                DB < 0-0.001.
                DA < 0.001.
                DA > 0-0.001.
                Type = "infinite".
(92) checktype(DA,DB,Type):-
                DB < 0.001.
                DB > 0-0.001.
                DA > 0.001.
                Type = "zero":
                DB < 0.001.
                DB > 0-0.001.
                DA < 0-0.001.
                Type = "zero".
(93) checktype(DA,DB,Type):-
                DA < 0.001.
                DA > 0-0.001.
                DB < 0.001.
                DB > 0-0.001.
                Type = "point".

(94) maketype(A,B,DA,DB,"normal"):-
                M = DB/DA.
                C = B-M*A.
                counter1(Num).
                New = Num+1.
                retract(counter1(Num)).
                asserta(counter1(New)).
                asserta(curve1("normal",New,M,C)).
(95) maketype(A,_,_,_,"infinite"):-
                counter1(Num).
                New = Num+1.
                retract(counter1(Num)).
                asserta(counter1(New)).
                asserta(curve1("infinite",New,A,0,0)).
(96) maketype(_,B,_,_,"zero"):-
                counter1(Num).
                New = Num+1.
                retract(counter1(Num)).
                asserta(counter1(New)).
                asserta(curve1("zero",New,B,0,0)).
(97) maketype(A,B,_,_,"point"):-
                counter1(Num).
                New = Num+1.
                retract(counter1(Num)).
                asserta(counter1(New)).
                asserta(curve1("point",New,A,B)).

(98) findintersect:-
                curve1("normal",Num1,M1,C1).
                curve1("normal",Num2,M2,C2).
                Num1 <> Num2.
                M1 <> M2.
                A = (C2-C1)/(M1-M2).
                B = M1*A+C1.
                str_real(Ast,A).
                str_real(Bst,B).
                asserta(result("intersect",Ast,Bst)):
                curve1("normal",Num1,M1,C1).
                curve1("normal",Num2,M2,C2).
                Num1 <> Num2.
                M1 < M2+0.001.
                M1 > M2-0.001.
                C1 < C2+0.001.
                C1 > C2-0.001.
                str_real(M1st,M1).
                str_real(C1st,C1).
                asserta(result("colinear",M1st,C1st)):
                curve1("normal",Num1,M1,C1).
                curve1("normal",Num2,M2,C2).
                Num1 <> Num2.
                M1 = M2.
                C1 <> C2.
                asserta(result("none","M","M")).
(99) findintersect:-
                curve1("normal",_,M1,C1).
                curve1("zero",_,B,_).
                A = (B-C1)/M1.
                str_real(Ast,A).
                str_real(Bst,B).
                asserta(result("intersect",Ast,Bst)).
(100) findintersect:-
                curve1("normal",_,M1,C1).
                curve1("infinite",_,A,_).
                B = M1*A+C1.
                str_real(Ast,A).
                str_real(Bst,B).
                asserta(result("intersect",Ast,Bst)).
(101) findintersect:-
                curve1("normal",_,M1,C1).
                curve1("point",_,A,B).
                B < (M1*A+C1)+0.001.
                B > (M1*A+C1)-0.001.
                str_real(Ast,A).
                str_real(Bst,B).
                asserta(result("intersect",Ast,Bst)):
                curve1("normal",_,_,_).
                curve1("point",_,_,_).
                asserta(result("none","M","M")).
(102) findintersect:-
                curve1("zero",Num1,B1,_).
                curve1("zero",Num2,B2,_).
                Num1 <> Num2.
                B1 < B2+0.001.
                B1 > B2-0.001.
                str_real(B1st,B1).
                asserta(result("colinear","M",B1st)):
                curve1("zero",Num1,_,_).
                curve1("zero",Num2,_,_).
                Num1 <> Num2.
                asserta(result("none","M","M")).
(103) findintersect:-
                curve1("zero",_,B,_).
                curve1("infinite",_,A,_).
                str_real(Ast,A).
                str_real(Bst,B).
                asserta(result("intersect",Ast,Bst)).
(104) findintersect:-
                curve1("zero",_,B,_).
                curve1("point",_,A1,B1).
                B < B1+0.001.
                B > B1-0.001.
                str_real(A1st,A1).
                str_real(B1st,B1).
                asserta(result("intersect",A1st,B1st));
                curve1("zero",_,_,_).
                curve1("point",_,_,_).
                asserta(result("none","M","M")).
(105) findintersect:-
                curve1("infinite",Num1,A1,_).
                curve1("infinite",Num2,A2,_).
                Num1 <> Num2.
                A1 < A2+0.001.
                A1 > A2-0.001.
                str_real(A1st,A1).
                asserta(result("colinear",A1st,"M")):
                curve1("infinite",Num1,_,_).
                curve1("infinite",Num2,_,_).
                Num1 <> Num2.
                asserta(result("none","M","M")).
(106) findintersect:-
                curve1("infinite",_,A,_).
                curve1("point",_,A1,B1).
                A < A1+0.001.
                A > A1-0.001.
                str_real(A1st,A1).
                str_real(B1st,B1).
                asserta(result("intersect",A1st,B1st)):
                curve1("infinite",_,_,_).
                curve1("point",_,_,_).
                asserta(result("none","M","M")).
(107) findintersect:-
                curve1("point",Num1,A1,B1).
                curve1("point",Num2,A2,B2).
                Num1 <> Num2.
                A1 < A2+0.001.
                A1 > A2-0.001.
                B1 < B2+0.001.
                B2 > B2-0.001.
                str_real(A1s,A1).
                str_real(B1s,B1).
                asserta(result("coincident",A1s,B1s)):
                curve1("point",Num1,_,_).
                curve1("point",Num2,_,_).
                Num1 <> Num2.
                asserta(result("none","M","M")).

(108) checkintersect("intersect",A1,B1,A2,B2,A3,B3,A4,B4):-
                result("intersect",Aints,Bints).
                str_real(Aints,Aint).
                str_real(Bints,Bint).
                getbig1(A1,A2,Ab1,As1).
                getbig1(A3,A4,Ab2,As2).
                getbig1(B1,B2,Bb1,Bs1).
                getbig1(B3,B4,Bb2,Bs2).
                Ab1 >= Aint-0.001.
                As1 <= Aint+0.001.
                Bb1 >= Bint-0.001.
                Bs1 <= Bint+0.001.
                Ab2 >= Aint-0.001.
                As2 <= Aint+0.001.
                Bb2 >= Bint-0.001.
                Bs2 <= Bint+0.001.
                counter2(Num).
                New = Num+1.
                retract(counter2(Num)).
                asserta(counter2(New)).
                asserta(status1("intersect",New)):
```

```
            asserta(end1).
(109) checkintersect("colinear",A1,B1,A2,B2,A3,B3,A4,B4):-
            checkposition1(A1,B1,A2,B2,A3,B3,A4,B4).
            counter2(Num).
            New = Num+1.
            retract(counter2(Num)).
            asserta(counter2(New)).
            asserta(status1("colinear",New)):
            asserta(end1).
(110) checkintersect("coincident",_,_,_,_,_,_,_,_):-
            counter2(Num).
            New = Num+1.
            retract(counter2(Num)).
            asserta(counter2(New)).
            asserta(status1("coincident",New)).
(111) checkintersect("none",_,_,_,_,_,_,_,_):-
            asserta(end1).

(112) checkresult:-
            status1("intersect",N1).
            status1("intersect",N2).
            N2 <> N1.
            status1("intersect",N3).
            N3 <> N1.
            N3 <> N2,!.
            checkposition.
(113) checkresult:-
            status1("intersect",N1).
            status1("intersect",N2).
            N2 <> N1.
            status1("colinear",N3).
            N3 <> N1.
            N3 <> N2,!.
            checkposition.
(114) checkresult:-
            status1("intersect",N1).
            status1("colinear",N2).
            N2 <> N1.
            status1("colinear",N3).
            N3 <> N1.
            N3 <> N2,!.
            checkposition.
(115) checkresult:-
            status1("coincident",N1).
            status1("colinear",N2).
            N2 <> N1.
            status1("colinear",N3).
            N3 <> N1.
            N3 <> N2.
            asserta(colinear).
(116) checkresult:-
            status1("colinear",N1).
            status1("colinear",N2).
            N2 <> N1.
            status1("colinear",N3).
            N3 <> N1.
            N3 <> N2.
            asserta(colinear).

(117) checkposition:-
            c1(_,P1,P2).
            not(c2(_,P1,_)).
            not(c2(_,_,P1)).
            not(c2(_,P2,_)).
            not(c2(_,_,P2)).

(118) checkposition1(A1,B1,A2,B2,A3,B3,A4,B4):-
            L1 = sqrt((A2-A1)*(A2-A1)+(B2-B1)*(B2-B1)).
            L2 = sqrt((A3-A1)*(A3-A1)+(B3-B1)*(B3-B1)).
            L3 = sqrt((A3-A2)*(A3-A2)+(B3-B2)*(B3-B2)).
            L2 <> 0.
            L3 <> 0.
            L1 < (L2+L3)+0.001.
            L1 > (L2+L3)-0.001;
            L1 = sqrt((A2-A1)*(A2-A1)+(B2-B1)*(B2-B1)).
            L2 = sqrt((A4-A1)*(A4-A1)+(B4-B1)*(B4-B1)).
            L3 = sqrt((A4-A2)*(A4-A2)+(B4-B2)*(B4-B2)).
            L2 <> 0.
            L3 <> 0.
            L1 < (L2+L3)+0.001.
            L1 > (L2+L3)-0.001.
(119) checkposition1(A1,B1,A2,B2,A3,B3,A4,B4):-
            L1 = sqrt((A4-A3)*(A4-A3)+(B4-B3)*(B4-B3)).
            L2 = sqrt((A3-A1)*(A3-A1)+(B3-B1)*(B3-B1)).
            L3 = sqrt((A4-A1)*(A4-A1)+(B4-B1)*(B4-B1)).
            L2 <> 0.
            L3 <> 0.
            L1 < (L2+L3)+0.001.
            L1 > (L2+L3)-0.001;
            L1 = sqrt((A4-A3)*(A4-A3)+(B4-B3)*(B4-B3)).
            L2 = sqrt((A3-A2)*(A3-A2)+(B3-B2)*(B3-B2)).
            L3 = sqrt((A4-A2)*(A4-A2)+(B4-B2)*(B4-B2)).
            L2 <> 0.
            L3 <> 0.
            L1 < (L2+L3)+0.001.
            L1 > (L2+L3)-0.001.
(120) checkposition1(A1,B1,A2,B2,A3,B3,A4,B4):-
            A1 < A3+0.001, A1 > A3-0.001,
            B1 < B3+0.001, B1 > B3-0.001,
            A2 < A4+0.001, A2 > A4-0.001,
            B2 < B4+0.001, B2 > B4-0.001;
            A1 < A4+0.001, A1 > A4-0.001,
            B1 < B4+0.001, B1 > B4-0.001,
            A2 < A3+0.001, A2 > A3-0.001,
            B2 < B3+0.001, B2 > B3-0.001.

(121) getbig1(N1,N2,Nb,Ns):-
            N1 >= N2.
            Nb = N1.
            Ns = N2;
            Nb = N2.
            Ns = N1.

(122) checknum(Num,Status):-
            Num = 0.
            Status = "Even",!:
            checknum1(Num,2,Status),!.

(123) checknum1(Num,N,Status):-
            Num - N = 0.
            Status = "Even",!:
            Num - N < 0.
            Status = "Odd",!:
            New = N + 2.
            checknum1(Num,New,Status).

(124) intersect_point("Ignore"):-
            not(colinear).
            findxyz(X,Y,Z).
            str_real(X,Xr).
            str_real(Y,Yr).
            str_real(Z,Zr).
            retractall(xy(_,_,_)).
            retractall(yz(_,_,_)).
            retractall(zx(_,_,_)),!.
            compare_intersect(Xr,Yr,Zr).
            asserta(ipoint(Xr,Yr,Zr)):
            colinear.
            asserta(newprp).
            retractall(xy(_,_,_)).
            retractall(yz(_,_,_)).
            retractall(zx(_,_,_)).
(125) intersect_point("Make Point"):-
            not(colinear).
            findxyz(X,Y,Z).
            str_real(X,Xr).
            str_real(Y,Yr).
            str_real(Z,Zr).
            asserta(ipoint(Xr,Yr,Zr)).
            retractall(xy(_,_,_)).
            retractall(yz(_,_,_)).
            retractall(zx(_,_,_)):
            colinear.
            retractall(xy(_,_,_)).
            retractall(yz(_,_,_)).
            retractall(zx(_,_,_)).
            checkcolinear.

(126) findxyz(X,Y,Z):-
            xy("Colinear",A,B).
            A <> "M".
            B <> "M".
            findxyz1(X,Y,Z);
            yz("Colinear",A,B).
            A <> "M".
            B <> "M".
            findxyz2(X,Y,Z);
            zx("Colinear",A,B).
            A <> "M".
            B <> "M".
            findxyz3(X,Y,Z):
            findx(X).
            findy(Y).
            findz(Z).

(127) findxyz1(X,Y,Z):-
            findy(Y).
            findz(Z).
            xy(_,A,B).
            str_real(A,Ar).
            str_real(B,Br).
            str_real(Y,Yr),
            Xr = (Yr-Br)/Ar,
            str_real(X,Xr).

(128) findxyz2(X,Y,Z):-
            findx(X).
            findz(Z).
            yz(_,A,B).
            str_real(A,Ar).
            str_real(B,Br).
            str_real(Z,Zr).
            Yr = (Zr-Br)/Ar,
            str_real(Y,Yr).

(130) findxyz3(X,Y,Z):-
            findx(X).
            findy(Y).
            zx(_,A,B).
            str_real(A,Ar).
            str_real(B,Br).
            str_real(X,Xr).
```

```
                Zr = (Xr-Br)/Ar.
                str_real(Z,Zr).

(131) findx(X):-
                xy(_,X,_).
                X <> "M".
                zx(_,_,X).
                X <> "M".
(132) findy(Y):-
                xy(_,_,Y).
                Y <> "M".
                yz(_,Y,_).
                Y <> "M".
(133) findz(Z):-
                yz(_,_,Z).
                Z <> "M".
                zx(_,Z,_).
                Z <> "M".

(134) compare_intersect(Xr,Yr,Zr):-
                ipoint(X,Y,Z).
                Xr < X+0.001, Xr > X-0.001.
                Yr < Y+0.001, Yr > Y-0.001.
                Zr < Z+0.001, Zr > Z-0.001.
                asserta(newprp),!.
                fail:
                true.

(135) piboundary(_,[]).
(136) piboundary(Name,[H|T]):-
                not(pib(Name,_,H)).
                last(Num).
                New = Num + 1.
                str_int(News,New).
                concat("p",News,P).
                asserta(pib(Name,P,H)).
                retract(last(Num)).
                asserta(last(New)).
                piboundary(Name,T):
                piboundary(Name,T).

(137) siboundary(_,_,[]).
(138) siboundary(Name,Group,[H|T]):-
                not(sib(Name,Group,_,H)).
                last(Num).
                New = Num + 1.
                str_int(News,New).
                concat("s",News,P).
                asserta(sib(Name,Group,P,H)).
                retract(last(Num)).
                asserta(last(New)).
                siboundary(Name,Group,T):
                siboundary(Name,Group,T).

(139) checkcolinear:-
                c1(_,p(X1,Y1,Z1),p(X2,Y2,Z2)).
                c2(_,p(X3,Y3,Z3),p(X4,Y4,Z4)).
                L1 = sqrt((X2-X1)*(X2-X1)+(Y2-Y1)*(Y2-Y1)+(Z2-Z1)*(Z2-Z1)).
                L2 = sqrt((X4-X3)*(X4-X3)+(Y4-Y3)*(Y4-Y3)+(Z4-Z3)*(Z4-Z3)).
                checkcolinear1(L1,L2).

(140) checkcolinear1(L1,L2):-
                L1 >= L2.
                checkcolinear2:
                checkcolinear3.

(141) checkcolinear2:-
                c1(H1,P1,P2).
                c2(_,P1,P4).
                commonpoint(H1,P1,P4).
                commonpoint(H1,P4,P2):
                c1(H1,P1,P2).
                c2(_,P3,P1).
                commonpoint(H1,P1,P3).
                commonpoint(H1,P3,P2):
                c1(H1,P1,P2).
                c2(_,P2,P4).
                commonpoint(H1,P2,P4).
                commonpoint(H1,P4,P1):
                c1(H1,P1,P2).
                c2(_,P3,P2).
                commonpoint(H1,P2,P3).
                commonpoint(H1,P3,P1).
(142) checkcolinear2:-
                c1(H1,p(X1,Y1,Z1),p(X2,Y2,Z2)).
                c2(_,p(X3,Y3,Z3),p(X4,Y4,Z4)).
                shortdist(X1,Y1,Z1,X3,Y3,Z3,X4,Y4,Z4,Xs,Ys,Zs,Xl,Yl,Zl),
                L1 = sqrt((X2-X1)*(X2-X1)+(Y2-Y1)*(Y2-Y1)+(Z2-Z1)*(Z2-Z1)).
                L2 = sqrt((X4-X3)*(X4-X3)+(Y4-Y3)*(Y4-Y3)+(Z4-Z3)*(Z4-Z3)).
                L3 = sqrt((Xs-X1)*(Xs-X1)+(Ys-Y1)*(Ys-Y1)+(Zs-Z1)*(Zs-Z1)).
                L4 = sqrt((Xl-X2)*(Xl-X2)+(Yl-Y2)*(Yl-Y2)+(Zl-Z2)*(Zl-Z2)).
                L1 < (L3+L2+L4)+0.001.
                L1 > (L3+L2+L4)-0.001.
                commonpoint(H1,p(X1,Y1,Z1),p(Xs,Ys,Zs)).
                commonpoint(H1,p(Xs,Ys,Zs),p(Xl,Yl,Zl)).
                commonpoint(H1,p(Xl,Yl,Zl),p(X2,Y2,Z2)).
(143) checkcolinear2:-
                c1(H1,p(X1,Y1,Z1),p(X2,Y2,Z2)).
                c2(H2,p(X3,Y3,Z3),p(X4,Y4,Z4)).
                shortdist(X4,Y4,Z4,X1,Y1,Z1,X2,Y2,Z2,Xs,Ys,Zs,_,_,_),
                L1 = sqrt((X2-X1)*(X2-X1)+(Y2-Y1)*(Y2-Y1)+(Z2-Z1)*(Z2-Z1)).
                L2 = sqrt((X1-X3)*(X1-X3)+(Y1-Y3)*(Y1-Y3)+(Z1-Z3)*(Z1-Z3)).
                L3 = sqrt((X3-X2)*(X3-X2)+(Y3-Y2)*(Y3-Y2)+(Z3-Z2)*(Z3-Z2)).
                L1 < (L3+L2)+0.001.
                L1 > (L3+L2)-0.001.
                commonpoint(H1,p(X1,Y1,Z1),p(X3,Y3,Z3)).
                commonpoint(H1,p(X3,Y3,Z3),p(X2,Y2,Z2)).
                commonpoint(H2,p(X3,Y3,Z3),p(Xs,Ys,Zs)).
                commonpoint(H2,p(X4,Y4,Z4),p(Xs,Ys,Zs)).
(144) checkcolinear2:-
                c1(H1,p(X1,Y1,Z1),p(X2,Y2,Z2)).
                c2(H2,p(X3,Y3,Z3),p(X4,Y4,Z4)).
                shortdist(X3,Y3,Z3,X1,Y1,Z1,X2,Y2,Z2,Xs,Ys,Zs,_,_,_),
                L1 = sqrt((X2-X1)*(X2-X1)+(Y2-Y1)*(Y2-Y1)+(Z2-Z1)*(Z2-Z1)).
                L2 = sqrt((X1-X4)*(X1-X4)+(Y1-Y4)*(Y1-Y4)+(Z1-Z4)*(Z1-Z4)).
                L3 = sqrt((X4-X2)*(X4-X2)+(Y4-Y2)*(Y4-Y2)+(Z4-Z2)*(Z4-Z2)).
                L1 < (L3+L2)+0.001.
                L1 > (L3+L2)-0.001.
                commonpoint(H1,p(X1,Y1,Z1),p(X4,Y4,Z4)).
                commonpoint(H1,p(X4,Y4,Z4),p(X2,Y2,Z2)).
                commonpoint(H2,p(X4,Y4,Z4),p(Xs,Ys,Zs)).
                commonpoint(H2,p(X3,Y3,Z3),p(Xs,Ys,Zs)).

(145) checkcolinear3:-
                c2(H1,P1,P2).
                c1(_,P1,P4).
                commonpoint(H1,P1,P4).
                commonpoint(H1,P4,P2):
                c2(H1,P1,P2).
                c1(_,P3,P1).
                commonpoint(H1,P1,P3).
                commonpoint(H1,P3,P2):
                c2(H1,P1,P2).
                c1(_,P2,P4).
                commonpoint(H1,P2,P4).
                commonpoint(H1,P4,P1):
                c2(H1,P1,P2).
                c1(_,P3,P2).
                commonpoint(H1,P2,P3).
                commonpoint(H1,P3,P1).
(146) checkcolinear3:-
                c2(H1,p(X1,Y1,Z1),p(X2,Y2,Z2)).
                c1(_,p(X3,Y3,Z3),p(X4,Y4,Z4)).
                shortdist(X1,Y1,Z1,X3,Y3,Z3,X4,Y4,Z4,Xs,Ys,Zs,Xl,Yl,Zl).
                L1 = sqrt((X2-X1)*(X2-X1)+(Y2-Y1)*(Y2-Y1)+(Z2-Z1)*(Z2-Z1)).
                L2 = sqrt((X4-X3)*(X4-X3)+(Y4-Y3)*(Y4-Y3)+(Z4-Z3)*(Z4-Z3)).
                L3 = sqrt((Xs-X1)*(Xs-X1)+(Ys-Y1)*(Ys-Y1)+(Zs-Z1)*(Zs-Z1)).
                L4 = sqrt((Xl-X2)*(Xl-X2)+(Yl-Y2)*(Yl-Y2)+(Zl-Z2)*(Zl-Z2)).
                L1 < (L3+L2+L4)+0.001.
                L1 > (L3+L2+L4)-0.001.
                commonpoint(H1,p(X1,Y1,Z1),p(Xs,Ys,Zs)).
                commonpoint(H1,p(Xs,Ys,Zs),p(Xl,Yl,Zl)).
                commonpoint(H1,p(Xl,Yl,Zl),p(X2,Y2,Z2)).
(147) checkcolinear3:-
                c2(H1,p(X1,Y1,Z1),p(X2,Y2,Z2)).
                c1(H2,p(X3,Y3,Z3),p(X4,Y4,Z4)).
                shortdist(X4,Y4,Z4,X1,Y1,Z1,X2,Y2,Z2,Xs,Ys,Zs,_,_,_),
                L1 = sqrt((X2-X1)*(X2-X1)+(Y2-Y1)*(Y2-Y1)+(Z2-Z1)*(Z2-Z1)).
                L2 = sqrt((X1-X3)*(X1-X3)+(Y1-Y3)*(Y1-Y3)+(Z1-Z3)*(Z1-Z3)).
                L3 = sqrt((X3-X2)*(X3-X2)+(Y3-Y2)*(Y3-Y2)+(Z3-Z2)*(Z3-Z2)).
                L1 < (L3+L2)+0.001.
                L1 > (L3+L2)-0.001.
                commonpoint(H1,p(X1,Y1,Z1),p(X3,Y3,Z3)).
                commonpoint(H1,p(X3,Y3,Z3),p(X2,Y2,Z2)).
                commonpoint(H2,p(X3,Y3,Z3),p(Xs,Ys,Zs)).
                commonpoint(H2,p(X4,Y4,Z4),p(Xs,Ys,Zs)).
(148) checkcolinear3:-
                c2(H1,p(X1,Y1,Z1),p(X2,Y2,Z2)).
                c1(H2,p(X3,Y3,Z3),p(X4,Y4,Z4)).
                shortdist(X3,Y3,Z3,X1,Y1,Z1,X2,Y2,Z2,Xs,Ys,Zs,_,_,_),
                L1 = sqrt((X2-X1)*(X2-X1)+(Y2-Y1)*(Y2-Y1)+(Z2-Z1)*(Z2-Z1)).
                L2 = sqrt((X1-X4)*(X1-X4)+(Y1-Y4)*(Y1-Y4)+(Z1-Z4)*(Z1-Z4)).
                L3 = sqrt((X4-X2)*(X4-X2)+(Y4-Y2)*(Y4-Y2)+(Z4-Z2)*(Z4-Z2)).
                L1 < (L3+L2)+0.001.
                L1 > (L3+L2)-0.001.
                commonpoint(H1,p(X1,Y1,Z1),p(X4,Y4,Z4)).
                commonpoint(H1,p(X4,Y4,Z4),p(X2,Y2,Z2)).
                commonpoint(H2,p(X4,Y4,Z4),p(Xs,Ys,Zs)).
                commonpoint(H2,p(X3,Y3,Z3),p(Xs,Ys,Zs)).

(149) commonpoint(H,P1,P2):-
                P1 = p(X1,Y1,Z1).
                P2 = p(X2,Y2,Z2).
                X1 < X2 + 0.001, X1 > X2 - 0.001.
                Y1 < Y2 + 0.001, Y1 > Y2 - 0.001.
                Z1 < Z2 + 0.001, Z1 > Z2 - 0.001:
                asserta(templine(H,P1,P2)).

(150) shortdist(X1,Y1,Z1,X3,Y3,Z3,X4,Y4,Z4,Xs,Ys,Zs,Xl,Yl,Zl):-
                L1 = sqrt((X3-X1)*(X3-X1)+(Y3-Y1)*(Y3-Y1)+(Z3-Z1)*(Z3-Z1)).
                L2 = sqrt((X4-X1)*(X4-X1)+(Y4-Y1)*(Y4-Y1)+(Z4-Z1)*(Z4-Z1)).
                L1 < L2.
                Xs = X3.
                Ys = Y3.
                Zs = Z3.
                Xl = X4.
                Yl = Y4.
                Zl = Z4:
                Xs = X4.
                Ys = Y4.
                Zs = Z4.
                Xl = X3.
```

```
                Y1 = Y3.
                Z1 = Z3.

(151) interpobsob:-
                status(Name1,List1,_).
                not(used(Name1)).
                List1 = [_|_].
                asserta(used(Name1)).
                interpobsob1(Name1,List1).
                interpobsob:
                status(Name1,List1,_).
                not(used(Name1)).
                List1 = [].
                asserta(used(Name1)).
                interpobsob:
                retractall(used(_)).
                retractall(used1(_)).

(152) interpobsob1(_,[]).
(153) interpobsob1(Name1,[H2|T2]):-
                H2 = pair(Name2.Group).
                curvelist(Name1.List1).
                scurvelist(Name2.Group.List3._).
                interpobsob2(Name1.Name2.Group.List1.List3).!.
                retractall(pair1(_,_)).
                retractall(used1(_)).
                interpobsob1(Name1.T2).

(154) interpobsob2(_,_,_.[],_).
(155) interpobsob2(Name1.Name2.Group.[H1|T1],List3):-
                not(used1(H1)).
                c(H1,p(X1a,Y1a,Z1a),p(X1b,Y1b,Z1b)).
                asserta(c1(H1,p(X1a,Y1a,Z1a),p(X1b,Y1b,Z1b))).
                interpobsob3(Name1,Name2.Group,[H1|T1],List3).
                not(intersect).
                asserta(used1(H1)).
                retractall(c1(_,_,_)).
                interpobsob2(Name1.Name2.Group.T1.List3):
                not(intersect).
                interpobsob2(Name1.Name2.Group.T1.List3):
                intersect.
                retractall(c1(_,_,_)).
                curvelist(Name1.NList1).
                scurvelist(Name2.Group.NList3._).
                retractall(intersect).
                interpobsob2(Name1.Name2.Group.NList1.NList3).

(156) interpobsob3(_,_,_,_.[]).
(157) interpobsob3(Name1.Name2.Group.[H1|T1],[H3|T3]):-
                H1 <> H3.
                not(pair1(H1,H3)).
                not(intersect).
                c(H1,p(X1a,Y1a,Z1a),p(X1b,Y1b,Z1b)).
                c(H3,p(X3a,Y3a,Z3a),p(X3b,Y3b,Z3b)).
                asserta(c2(H3,p(X3a,Y3a,Z3a),p(X3b,Y3b,Z3b))).
                asserta(counter(0)).

                intersection(X1a,Y1a,Z1a,X1b,Y1b,Z1b,X3a,Y3a,Z3a,X3b,Y3b,Z3b,"
Make Point").!.
                retractall(end1).
                retractall(c2(_,_,_)).
                asserta(pair1(H1,H3)).
                checkforintersect(Name1,Name2,Group,H1,H3).!.
                interpobsob3(Name1.Name2.Group,[H1|T1],T3):
                not(intersect).
                interpobsob3(Name1.Name2.Group,[H1|T1],T3):
                intersect.
                true.

(158) checkforintersect(Name1,Name2.Group,H1,H3):-
                counter(Num).
                Num = 1.
                ipoint(Xint,Yint,Zint).
                Pint = p(Xint,Yint,Zint).
                curvelist(Name1.List1).
                scurvelist(Name2,Group,List3.Plane).
                subtract(H1.List1,N1List1).
                subtract(H3.List3,N1list3).
                retract(curvelist(Name1.List1)).
                retract(scurvelist(Name2.Group.List3.Plane)).
                asserta(curvelist(Name1,N1List1)).
                asserta(scurvelist(Name2.Group.N1List3.Plane)).
                findallP(H1.NameP.Name1).
                asserta(pair2(Name2.Group)).
                findallS(H3.NameS.GroupS).
                retract(pair2(Name2.Group)).
                findallF(H1.NameF.Name1).
                c(H1.P1.P2).
                c(H3.P3.P4).
                checkforintersect1(Name1.H1.P1.Pint.NameP.NameF).!.
                checkforintersect1(Name1.H1.P2.Pint.NameP.NameF).!.
                checkforintersect2(Name2.Group.H3.P3.Pint.NameS.GroupS).!.
                checkforintersect2(Name2.Group.H3.P4.Pint.NameS.GroupS).!.
                pdel(Name1.H1).!.
                pdel(NameP.H1).!.
                sdel(Name2.Group.H3).!.
                sdel(NameS.GroupS.H3).!.
                retractall(counter(_)).
                retract(ipoint(Xint,Yint,Zint)).
                asserta(intersect).
(159) checkforintersect(Name1.Name2.Group.H1.H3):-
                counter(Num).
                Num = 1.
                templine(H1,_,_).
                findallP(H1.NameP.Name1).
                findallF(H1.NameF.Name1).
                checkforintersect3(Name1.H1.NameP.NameF).
                retract(curvelist(Name1.List)).
                subtract(H1.List.NList).
                asserta(curvelist(Name1.NList)).
                pdel(Name1.H1).
                pdel(NameP.H1).
                retractall(counter(_)).
                asserta(intersect).
                checkforintersect_next(Name2.Group.H3).
(160) checkforintersect(_,Name2.Group._,H3):-
                counter(Num).
                Num = 1.
                templine(H3,_,_).
                asserta(pair2(Name2.Group)).
                findallS(H3.NameS.GroupS).
                retract(pair2(Name2.Group)).
                checkforintersect4(Name2.Group.H3.NameS.GroupS).
                retract(scurvelist(Name2.Group.List.Plane)).
                subtract(H3.List.NList).
                asserta(scurvelist(Name2.Group.NList.Plane)).
                sdel(Name2.Group.H3).
                sdel(NameS.GroupS.H3).
                retractall(counter(_)).
                asserta(intersect).
(161) checkforintersect(_,_,_,_,_):-
                counter(Num).
                Num = 0.
                retractall(counter(_)).

(162) checkforintersect_next(Name2.Group.H3):-
                templine(H3,_,_).
                asserta(pair2(Name2.Group)).
                findallS(H3.NameS.GroupS).
                retract(pair2(Name2.Group)).
                checkforintersect4(Name2.Group.H3.NameS.GroupS).
                retract(scurvelist(Name2.Group.List.Plane)).
                subtract(H3.List.NList).
                asserta(scurvelist(Name2.Group.NList.Plane)).
                sdel(Name2.Group.H3).
                sdel(NameS.GroupS.H3):
                true.

(163) checkforintersect1(_,_,p(X,Y,Z),p(Xi,Yi,Zi),_,_):-
                X = X1i.
                Y = Y1i.
                Z = Z1i.
                X1i < Xi+0.001,                 X1i > Xi-0.001.
                Y1i < Yi+0.001, Y1i > Yi-0.001.
                Z1i < Zi+0.001.                 Z1i > Zi-0.001.!.
(164) checkforintersect1(Name1,H,p(X,Y,Z),p(Xi,Yi,Zi),NameP,NameF):-
                c(Old,p(X1,Y1,Z1),p(X2,Y2,Z2)).
                X1 < X+0.001, X1 > X-0.001.
                Y1 < Y+0.001, Y1 > Y-0.001.
                Z1 < Z+0.001, Z1 > Z-0.001.
                X2 < Xi+0.001, X2 > Xi-0.001.
                Y2 < Yi+0.001, Y2 > Yi-0.001.
                Z2 < Zi+0.001, Z2 > Zi-0.001.
                curvelist(Name1.List).
                not(member(Old.List)).
                retract(curvelist(Name1.List)).
                asserta(curvelist(Name1,[Old|List])).
                findallP1(NameP,H.Old).
                findallF1(NameF,H.Old).
                pupdate(Name1,H.Old).!:
                c(Old,p(X1,Y1,Z1),p(X2,Y2,Z2)).
                X1 < Xi+0.001, X1 > Xi-0.001.
                Y1 < Yi+0.001, Y1 > Yi-0.001.
                Z1 < Zi+0.001, Z1 > Zi-0.001.
                X2 < X+0.001, X2 > X-0.001.
                Y2 < Y+0.001, Y2 > Y-0.001.
                Z2 < Z+0.001, Z2 > Z-0.001.
                curvelist(Name1.List).
                not(member(Old.List)).
                retract(curvelist(Name1.List)).
                asserta(curvelist(Name1,[Old|List])).
                findallP1(NameP,H.Old).
                findallF1(NameF,H.Old).
                pupdate(Name1,H.Old).!.
(165) checkforintersect1(Name1,H,p(X,Y,Z),p(Xi,Yi,Zi),NameP,NameF):-
                c(Last,_,_).
                frontstr(1,Last,_,Nos).
                str_int(Nos,Noi).
                New1 = Noi+1.
                str_int(New1s,New1).
                concat("c",New1s,C).
                checkpoint2(X,Y,Z,X1,Y1,Z1).
                checkpoint2(Xi,Yi,Zi,X2,Y2,Z2).
                asserta(c(C,p(X1,Y1,Z1),p(X2,Y2,Z2))).
                retract(curvelist(Name1,List)).
                asserta(curvelist(Name1,[C|List])).
                findallP1(NameP,H,C).
                findallF1(NameF,H,C).
                pupdate(Name1,H,C).!.

(166) checkforintersect2(_,_,_,p(X,Y,Z),p(Xi,Yi,Zi),_,_):-
                X = X1i.
```

```
                Y = Y1i.
                Z = Z1i.
                X1i < Xi+0.001.          X1i > Xi-0.001.
                Y1i < Yi+0.001.          Y1i > Yi-0.001.
                Z1i < Zi+0.001.          Z1i > Zi-0.001.!.
(167) checkforintersect2(Name,Group,H,p(X,Y,Z),p(Xi,Yi,Zi),NameS,GroupS):-
                c(Old,p(X1,Y1,Z1),p(X2,Y2,Z2)).
                X1 < X+0.001. X1 > X-0.001.
                Y1 < Y+0.001. Y1 > Y-0.001.
                Z1 < Z+0.001. Z1 > Z-0.001.
                X2 < Xi+0.001. X2 > Xi-0.001.
                Y2 < Yi+0.001. Y2 > Yi-0.001.
                Z2 < Zi+0.001. Z2 > Zi-0.001.
                scurvelist(Name,Group,List,Plane).
                not(member(Old,List)).
                retract(scurvelist(Name,Group,List,Plane)).
                assert(scurvelist(Name,Group,[Old|List],Plane)).
                findallS1(NameS,GroupS,H,Old).
                supdate(Name,Group,H,Old).!.
                c(Old,p(X1,Y1,Z1),p(X2,Y2,Z2)).
                X1 < Xi+0.001. X1 > Xi-0.001.
                Y1 < Yi+0.001. Y1 > Yi-0.001.
                Z1 < Zi+0.001. Z1 > Zi-0.001.
                X2 < X+0.001. X2 > X-0.001.
                Y2 < Y+0.001. Y2 > Y-0.001.
                Z2 < Z+0.001. Z2 > Z-0.001.
                scurvelist(Name,Group,List,Plane).
                not(member(Old,List)).
                retract(scurvelist(Name,Group,List,Plane)).
                assert(scurvelist(Name,Group,[Old|List],Plane)).
                findallS1(NameS,GroupS,H,Old).
                supdate(Name,Group,H,Old).!.
(168) checkforintersect2(Name,Group,H,p(X,Y,Z),p(Xi,Yi,Zi),NameS,GroupS):-
                c(Last,_,_).
                frontstr(1,Last,_,Nos).
                str_int(Nos,Noi).
                New1 = Noi+1.
                str_int(New1s,New1).
                concat("c",New1s,C).
                checkpoint2(X,Y,Z,X1,Y1,Z1).
                checkpoint2(Xi,Yi,Zi,X2,Y2,Z2).
                assert(c(C,p(X1,Y1,Z1),p(X2,Y2,Z2))).
                retract(scurvelist(Name,Group,List,Plane)).
                assert(scurvelist(Name,Group,[C|List],Plane)).
                findallS1(NameS,GroupS,H,C).
                supdate(Name,Group,H,C).!.

(169) checkforintersect3(Name1,H1,NameP,NameF):-
                templine(H1,P1,P2).
                existcurve(C,P1,P2).
                curvelist(Name1,List).
                not(member(C,List)).
                retract(curvelist(Name1,List)).
                assert(curvelist(Name1,[C|List])).
                findallP1(NameP,H1,C).
                findallF1(NameF,H1,C).
                pupdate(Name1,H1,C).!.
                retract(templine(H1,P1,P2)).
                checkforintersect3(Name1,H1,NameP,NameF):
                true.

(170) checkforintersect4(Name,Group,H,NameS,GroupS):-
                templine(H,P1,P2).
                existcurve(C,P1,P2).
                scurvelist(Name,Group,List,Plane).
                not(member(C,List)).
                retract(scurvelist(Name,Group,List,Plane)).
                assert(scurvelist(Name,Group,[C|List],Plane)).
                findallS1(NameS,GroupS,H,C).
                supdate(Name,Group,H,C).!.
                retract(templine(H,P1,P2)).
                checkforintersect4(Name,Group,H,NameS,GroupS):
                true.

(171) pupdate(Name,H,N):-
                pob(Name,List).
                member(H,List).
                retract(pob(Name,List)).
                asserta(pob(Name,[N|List])):
                pib(Name,P,List).
                member(H,List).
                retract(pib(Name,P,List)).
                asserta(pib(Name,P,[N|List])):
                true.

(172) supdate(Name,Group,H,N):-
                sob(Name,Group,List).
                member(H,List).
                retract(sob(Name,Group,List)).
                asserta(sob(Name,Group,[N|List])):
                sib(Name,Group,P,List).
                member(H,List).
                retract(sib(Name,Group,P,List)).
                asserta(sib(Name,Group,P,[N|List])):
                true.

(173) pdel(Name,H):-
                pob(Name,List).
                member(H,List).
                subtract(H,List,NList).
                retract(pob(Name,List)).
                asserta(pob(Name,NList)):
                pib(Name,P,List).
                member(H,List).
                subtract(H,List,NList).
                retract(pib(Name,P,List)).
                asserta(pib(Name,P,NList)):
                true.

(174) sdel(Name,Group,H):-
                sob(Name,Group,List).
                member(H,List).
                subtract(H,List,NList).
                retract(sob(Name,Group,List)).
                asserta(sob(Name,Group,NList)):
                sib(Name,Group,P,List).
                member(H,List).
                subtract(H,List,NList).
                retract(sib(Name,Group,P,List)).
                asserta(sib(Name,Group,P,NList)):
                true.

(175) existcurve(C,p(X1,Y1,Z1),p(X2,Y2,Z2)):-
                c(C,p(X11,Y11,Z11),p(X22,Y22,Z22)).
                X11 < X1+0.001. X11 > X1-0.001.
                Y11 < Y1+0.001. Y11 > Y1-0.001.
                Z11 < Z1+0.001. Z11 > Z1-0.001.
                X22 < X2+0.001. X22 > X2-0.001.
                Y22 < Y2+0.001. Y22 > Y2-0.001.
                Z22 < Z2+0.001. Z22 > Z2-0.001.!:
                c(C,p(X11,Y11,Z11),p(X22,Y22,Z22)).
                X11 < X2+0.001. X11 > X2-0.001.
                Y11 < Y2+0.001. Y11 > Y2-0.001.
                Z11 < Z2+0.001. Z11 > Z2-0.001.
                X22 < X1+0.001. X22 > X1-0.001.
                Y22 < Y1+0.001. Y22 > Y1-0.001.
                Z22 < Z1+0.001. Z22 > Z1-0.001.!:
                c(Last,_,_).
                frontstr(1,Last,_,Nos).
                str_int(Nos,Noi).
                New1 = Noi+1.
                str_int(New1s,New1).
                concat("c",New1s,C).
                checkpoint2(X1,Y1,Z1,Xa,Ya,Za).
                checkpoint2(X2,Y2,Z2,Xb,Yb,Zb).
                asserta(c(C,p(Xa,Ya,Za),p(Xb,Yb,Zb))).

(176) checkpoint2(X,Y,Z,Nx,Ny,Nz):-
                c(_,p(X1,Y1,Z1),_).
                X1 < X+0.001. X1 > X-0.001.
                Y1 < Y+0.001. Y1 > Y-0.001.
                Z1 < Z+0.001. Z1 > Z-0.001.
                Nx = X1.
                Ny = Y1.
                Nz = Z1;
                c(_,_,p(X1,Y1,Z1)).
                X1 < X+0.001. X1 > X-0.001.
                Y1 < Y+0.001. Y1 > Y-0.001.
                Z1 < Z+0.001. Z1 > Z-0.001.
                Nx = X1.
                Ny = Y1.
                Nz = Z1;
                Nx = X.
                Ny = Y.
                Nz = Z.

(177) findallP(H,Name,Old):-
                curvelist(Name,List).
                Name <> Old.
                member(H,List).
                subtract(H,List,NList).
                retract(curvelist(Name,List)).
                asserta(curvelist(Name,Nlist)):
                Name = "Ignore".

(178) findallS(H,Name,Group):-
                scurvelist(Name,Group,List,P).
                not(pair2(Name,Group)).
                member(H,List).
                subtract(H,List,NList).
                retract(scurvelist(Name,Group,List,P)).
                asserta(scurvelist(Name,Group,NList,P)):
                Name = "Ignore".
                Group = "".

(179) findallF(H,Name,Old):-
                freesurf(Name,_,List).
                Name <> Old.
                member(H,List).
                subtract(H,List,NList).
                retract(freesurf(Name,Name,List)).
                asserta(freesurf(Name,Name,NList)):
                Name = "".

(180) findallP1(Name,H,C):-
                Name <> "Ignore".
                curvelist(Name,List).
                not(member(C,List)).
                retract(curvelist(Name,List)).
                asserta(curvelist(Name,[C|List])).
                pupdate(Name,H,C):
                true.
```

```
(181) findallS1(Name,Group,H,C):-
        Name <> "Ignore".
        scurvelist(Name,Group,List,P).
        not(member(C,List)).
        retract(scurvelist(Name,Group,List,P)).
        asserta(scurvelist(Name,Group,[C|List],P)).
        supdate(Name,Group,H,C);
        true.

(182) findallF1(Name,H,C):-
        Name <> "Ignore".
        freesurf(Name,_,List).
        not(member(H,List)).
        retract(freesurf(Name,Name,List)).
        asserta(freesurf(Name,Name,[C|List])):
        true.

(183) intersobsob:-
        status(Name1,List1,_).
        not(used(Name1)).
        List1 = [_|_].
        asserta(used(Name1)).
        intersobsob1(List1).
        intersobsob:
        status(Name1,List1,_).
        not(used(Name1)).
        List1 = [].
        asserta(used(Name1)).
        intersobsob:
        retractall(used(_)).
        retractall(used1(_)).

(184) intersobsob1([]).
(185) intersobsob1([H|T1]):-
        T1 = [_|_].
        H1 = pair(Name1,Group1),
        scurvelist(Name1,Group1,List1,_).
        intersobsob2(Name1,Group1,List1,T1),!.
        intersobsob1(T1):
        true.

(186) intersobsob2(_,_,_,[]).
(187) intersobsob2(Name1,Group1,List1,[H2|T2]):-
        H2 = pair(Name2,Group2).
        scurvelist(Name2,Group2,List2,_).
        intersobsob3(Name1,Group1,Name2,Group2,List1,List2),!.
        retractall(pair1(_,_)).
        retractall(used1(_)).
        intersobsob2(Name1,Group1,List1,T2).

(188) intersobsob3(_,_,_,_,[],_).
(189) intersobsob3(Name1,Group1,Name2,Group2,[H1|T1],List2):-
        not(used1(H1)).
        c(H1,p(X1a,Y1a,Z1a),p(X1b,Y1b,Z1b)),
        asserta(c1(H1,p(X1a,Y1a,Z1a),p(X1b,Y1b,Z1b))).
        intersobsob4(Name1,Group1,Name2,Group2,H1,List2),
        not(intersect).
        retract(c1(H1,p(X1a,Y1a,Z1a),p(X1b,Y1b,Z1b))).
        intersobsob3(Name1,Group1,Name2,Group2,T1,List2):
        not(intersect).
        intersobsob3(Name1,Group1,Name2,Group2,T1,List2):
        intersect.
        retractall(c1(_,_,_)).
        retractall(intersect).
        scurvelist(Name1,Group1,NList1,_).
        scurvelist(Name2,Group2,NList2,_).
        intersobsob3(Name1,Group1,Name2,Group2,NList1,NList2).

(190) intersobsob4(_,_,_,_,_,[]).
(191) intersobsob4(Name1,Group1,Name2,Group2,H1,[H2|T2]):-
        H1 <> H2.
        not(pair1(H1,H2)).
        not(intersect).
        c(H1,p(X1a,Y1a,Z1a),p(X1b,Y1b,Z1b)).
        c(H2,p(X2a,Y2a,Z2a),p(X2b,Y2b,Z2b)).
        asserta(c2(H2,p(X2a,Y2a,Z2a),p(X2b,Y2b,Z2b))).
        asserta(counter(0)).

        intersection(X1a,Y1a,Z1a,X1b,Y1b,Z1b,X2a,Y2a,Z2a,X2b,Y2b,Z2b,"
Make Point").!.
        retractall(end1).
        retractall(c2(_,_,_)).
        asserta(pair1(H1,H2)).
        scheckforintersect(Name1,Group1,Name2,Group2,H1,H2),!.
        intersobsob4(Name1,Group1,Name2,Group2,H1,T2):
        not(intersect).
        intersobsob4(Name1,Group1,Name2,Group2,H1,T2):
        intersect.
        true.

(192) scheckforintersect(Name1,Group1,Name2,Group2,H1,H2):-
        counter(Num).
        Num = 1.
        ipoint(Xint,Yint,Zint).
        Pint = p(Xint,Yint,Zint).
        scurvelist(Name1,Group1,List1,Plane1).
        scurvelist(Name2,Group2,List2,Plane2).
        subtract(H1,List1,NList1).
        subtract(H2,List2,NList2).
        retract(scurvelist(Name1,Group1,List1,Plane1)).
        retract(scurvelist(Name2,Group2,List2,Plane2)).
        asserta(scurvelist(Name1,Group1,NList1,Plane1)).
        asserta(scurvelist(Name2,Group2,NList2,Plane2)).
        asserta(pair2(Name1,Group1)).
        asserta(pair2(Name2,Group2)).
        findallS(H1,NameS1,GroupS1).
        findallS(H2,NameS2,GroupS2).
        retract(pair2(Name1,Group1)).
        retract(pair2(Name2,Group2)).
        c(H1,P1,P2).
        c(H2,P3,P4).
        checkforintersect2(Name1,Group1,H1,P1,Pint,NameS1,GroupS1),!.
        checkforintersect2(Name1,Group1,H1,P2,Pint,NameS1,GroupS1),!.
        checkforintersect2(Name2,Group2,H2,P3,Pint,NameS2,GroupS2),!.
        checkforintersect2(Name2,Group2,H2,P4,Pint,NameS2,GroupS2),!.
        sdel(Name1,Group1,H1),!.
        sdel(Name2,Group2,H2),!.
        sdel(NameS1,GroupS1,H1).
        sdel(NameS2,GroupS2,H2).
        retractall(counter(_)).
        retract(ipoint(Xint,Yint,Zint)).
        asserta(intersect).
(193) scheckforintersect(Name1,Group1,Name2,Group2,H1,H2):-
        counter(Num).
        Num = 1.
        templine(H1,_,_).
        asserta(pair2(Name1,Group1)).
        findallS(H1,NameS1,GroupS1).
        retract(pair2(Name1,Group1)).
        checkforintersect4(Name1,Group1,H1,NameS1,GroupS1).
        retract(scurvelist(Name1,Group1,List1,Plane1)).
        subtract(H1,List1,NList1).
        asserta(scurvelist(Name1,Group1,NList1,Plane1)).
        sdel(Name1,Group1,H1).
        sdel(NameS1,GroupS1,H1).
        retractall(counter(_)).
        asserta(intersect).
        scheckforintersect_next(Name2,Group2,H2).
(194) scheckforintersect(_,_,Name2,Group2,_,H2):-
        counter(Num).
        Num = 1.
        templine(H2,_,_).
        asserta(pair2(Name2,Group2)).
        findallS(H2,NameS2,GroupS2).
        retract(pair2(Name2,Group2)).
        checkforintersect4(Name2,Group2,H2,NameS2,GroupS2).
        retract(scurvelist(Name2,Group2,List2,Plane2)).
        subtract(H2,List2,NList2).
        asserta(scurvelist(Name2,Group2,NList2,Plane2)).
        sdel(Name2,Group2,H2).
        sdel(NameS2,GroupS2,H2).
        retractall(counter(_)).
        asserta(intersect).
(195) scheckforintersect(_,_,_,_,_,_):-
        counter(Num).
        Num = 0.
        retractall(counter(_)).

(196) scheckforintersect_next(Name2,Group2,H2):-
        templine(H2,_,_).
        asserta(pair2(Name2,Group2)).
        findallS(H2,NameS2,GroupS2).
        retract(pair2(Name2,Group2)).
        checkforintersect4(Name2,Group2,H2,NameS2,GroupS2).
        retract(scurvelist(Name2,Group2,List2,Plane2)).
        subtract(H2,List2,NList2).
        asserta(scurvelist(Name2,Group2,NList2,Plane2)).
        sdel(Name2,Group2,H2).
        sdel(NameS2,GroupS2,H2):
        true.

(197) removecurve:-
        status(Name1,List2,_).
        not(used(Name1)).
        List2 = [_|_].
        asserta(used(Name1)).
        curvelist1(Name1,List1).
        removecurve1(Name1,List1,List2).
        removecurve:
        status(Name1,List2,_).
        not(used(Name1)).
        List2 = [].
        asserta(used(Name1)).
        removecurve:
        retractall(used(_)).

(198) removecurve1(_,_,[]).
(199) removecurve1(Name1,List1,List2):-
        List2 = [pair(Name2,Group)|T].
        scurvelist(Name2,Group,List3,_).
        removecurve2(Name1,Name2,Group,List1,List3).
        removecurve1(Name1,List1,T).

(200) removecurve2(_,_,_,_,[]).
(201) removecurve2(Name1,Name2,Group,List1,[H3|T3]):-
        curvelist(Name1,CL).
        not(member(H3,CL)).
        c(H3,p(X3,Y3,Z3),p(X4,Y4,Z4)).
        Nx = X3+(X4-X3)/2.
        Ny = Y3+(Y4-Y3)/2.
        Nz = Z3+(Z4-Z3)/2.
```

```
                asserta(midpoint(Nx,Ny,Nz)).
                removecurve3(Name2.Group.List1.H3).
                retractall(midpoint(_,_,_)).
                removecurve2(Name1.Name2.Group.List1.T3):
                removecurve2(Name1.Name2.Group.List1.T3).

(202) removecurve3(Name2.Group.[].H3):-
                scurvelist(Name2.Group.List.Plane).
                subtract(H3.List.NList).
                retract(scurvelist(Name2.Group.List.Plane)).
                asserta(scurvelist(Name2.Group.NList.Plane)).
                sdel(Name2.Group.H3).
(203) removecurve3(Name2.Group.[H|T1],H3):-
                removecurve4(H1).
                not(ptbound).
                removecurve3(Name2.Group.T1.H3):
                ptbound.
                retractall(ptbound).
                true.

(204) removecurve4(Name):-
                s1(Name.p(X1,Y1,Z1).p(X2,Y2,Z2).p(X3,Y3,Z3)).nl.
                L1 = sqrt((X2-X1)*(X2-X1)+(Y2-Y1)*(Y2-Y1)+(Z2-Z1)*(Z2-Z1)).
                L2 = sqrt((X1-X3)*(X1-X3)+(Y1-Y3)*(Y1-Y3)+(Z1-Z3)*(Z1-Z3)).
                L3 = sqrt((X3-X2)*(X3-X2)+(Y3-Y2)*(Y3-Y2)+(Z3-Z2)*(Z3-Z2)).
                midpoint(Xm,Ym,Zm).
                L4 = sqrt((X1-Xm)*(X1-Xm)+(Y1-Ym)*(Y1-Ym)+(Z1-Zm)*(Z1-
                Zm)).
                L5 = sqrt((X2-Xm)*(X2-Xm)+(Y2-Ym)*(Y2-Ym)+(Z2-Zm)*(Z2-
                Zm)).
                L6 = sqrt((X3-Xm)*(X3-Xm)+(Y3-Ym)*(Y3-Ym)+(Z3-Zm)*(Z3-
                Zm)).
                C1 = (L3*L3-L2*L2+L1*L1)/(2*L1).
                Ht = (L3*L3-C1*C1).
                checkzero(Ht.H).
                AT = L1*H/2.
                C11 = (L4*L4-L5*L5+L1*L1)/(2*L1).
                Ht1 = (L4*L4-C11*C11).
                checkzero(Ht1.H1).
                A1 = L1*H1/2.
                C12 = (L4*L4-L6*L6+L2*L2)/(2*L2).
                Ht2 = (L4*L4-C12*C12).
                checkzero(Ht2.H2).
                A2 = L2*H2/2.
                C13 = (L5*L5-L6*L6+L3*L3)/(2*L3).
                Ht3 = (L5*L5-C13*C13).
                checkzero(Ht3.H3).
                A3 = L3*H3/2.
                AT < (A1+A2+A3)+0.001.
                AT > (A1+A2+A3)-0.001.
                asserta(ptbound):
                true.

(205) checkzero(Num.Ans):-
                Num < 0 + 0.0001.
                Num > 0 - 0.0001.
                Ans = 0;
                Ans = sqrt(Num).

(206) pribackup:-
                pobbackup.!.
                pibbackup.!.

(207) pobbackup:-
                pob(Name.List),
                not(used(Name)),
                asserta(used(Name)),
                asserta(pob1(Name.List)).
                pobbackup:
                retractall(used(_)).

(208) pibbackup:-
                pib(Name.P.List).
                not(pair1(Name.P)).
                asserta(pair1(Name.P)).
                asserta(pib1(Name.P.List)).
                pibbackup:
                retractall(pair1(_,_)).

(209) secbackup:-
                sobbackup.!.
                sibbackup.!.

(210) sobbackup:-
                sob(Name.Group.List).
                not(pair(Name.Group)).
                asserta(pair(Name.Group)).
                asserta(sob1(Name.Group.List)).
                sobbackup:
                retractall(pair(_,_)).

(211) sibbackup:-
                sib(Name.Group.P.List).
                not(tri(Name.Group.P)).
                asserta(tri(Name.Group.P)).
                asserta(sib1(Name.Group.P.List)).
                sibbackup:
                retractall(tri(_,_,_)).

(212) cleanup:-
                status(Name1.List2,_).
                not(used(Name1)).
                List2 = [_|_].
                asserta(used(Name1)).
                curvelist(Name1.List1).
                cleanup1(Name1.List1.List2).
                cleanup:
                status(Name1.List2,_).
                not(used(Name1)).
                List2 = [].
                asserta(used(Name1)).
                curvelist(Name1.List1).
                cleanall(Name1.List1).
                cleanup:
                retractall(used(_)).
                true.

(213) cleanall(_,[]).
(214) cleanall(Name1,[H|T1]):-
                asserta(curvestatus(H1.nc.Name1)).
                cleanall(Name1.T1):
                true.

(215) cleanup1(_,_,[]).
(216) cleanup1(Name1.List1.List2):-
                List2 = [pair(Name2.Group)|T],
                scurvelist(Name2.Group.List3,_).
                cleanup2(Name1.Name2.Group.List1.List3).
                cleanup1(Name1.List1.T).

(217) cleanup2(_,_,_,[],_).
(218) cleanup2(Name1.Name2.Group.[H1|T1],List3):-
                member(H1,List3).
                subtract(H1.List3.NList3).
                retract(scurvelist(Name2.Group.List3.Plane)).
                asserta(scurvelist(Name2.Group.NList3.Plane)).
                sdel(Name2.Group.H1).
                asserta(curvestatus(H1.shared.Name1)).
                cleanup2(Name1.Name2.Group.T1.NList3):
                cleanup2(Name1.Name2.Group.T1.List3).

(219) groupscurve:-
                status(Name1.List2,_).
                not(used(Name1)).
                List2 = [_|_].
                asserta(used(Name1)).
                groupscurve1(Name1.List2).
                groupscurve:
                status(Name1.List2,_).
                not(used(Name1)).
                List2 = [].
                asserta(used(Name1)).
                groupscurve:
                retractall(used(_)).
                true.

(220) groupscurve1(_,[]).
(221) groupscurve1(Name1.List2):-
                List2 = [pair(Name2.Group)|T].
                scurvelist(Name2.Group.List,_).
                groupscurve2(Name1.Name2.Group.List).
                groupscurve1(Name1.T).

(222) groupscurve2(_,_,_,[]).
(223) groupscurve2(Name1.Name2.Group.[H|T]):-
                asserta(curvestatus(H.bc(Name2.Group).Name1)).
                groupscurve2(Name1.Name2.Group.T).

(224) grouppcurve:-
                status(Name1.List2,_).
                not(used(Name1)).
                List2 = [_|_].
                asserta(used(Name1)).
                curvelist(Name1.List1).
                grouppcurve1(Name1.List1.List2).
                grouppcurve:
                status(Name1.List2,_).
                not(used(Name1)).
                List2 = [].
                asserta(used(Name1)).
                grouppcurve:
                retractall(used(_)).
                true.

(225) grouppcurve1(_,_,[]).
(226) grouppcurve1(Name1.List1.[H|T]):-
                H = pair(Name2.Group).
                scurvelist1(Name2.Group.List2).
                grouppcurve2(Name1.Name2.Group.List1.List2).
                grouppcurve1(Name1.List1,T).

(227) grouppcurve2(_,_,_,[],_).
(228) grouppcurve2(Name1.Name2.Group.[H1|T1].List2):-
                not(curvestatus(H1.shared.Name1)).
                not(curvestatus(H1.fc(_,_).Name1)).
                c(H1.p(X3,Y3,Z3).p(X4,Y4,Z4)).
                Nx = X3+(X4-X3)/2.
                Ny = Y3+(Y4-Y3)/2.
                Nz = Z3+(Z4-Z3)/2.
                asserta(midpoint(Nx,Ny,Nz)).
                grouppcurve3(Name1.Name2.Group.H1.List2).
                retract(midpoint(Nx,Ny,Nz)).
```

```
                grouppcurve2(Name1,Name2,Group,T1,List2);
                grouppcurve2(Name1,Name2,Group,T1,List2).

(229) grouppcurve3(Name1,_,_,H1,[]):-
                not(curvestatus(H1,nc,Name1)),
                asserta(curvestatus(H1,nc,Name1));
                true.
(230) grouppcurve3(Name1,Name2,Group,H1,[H2|T2]):-
                grouppcurve4(H2,Group),
                not(ptbound),
                grouppcurve3(Name1,Name2,Group,H1,T2);
                ptbound,
                retractall(curvestatus(H1,_,Name1)),
                asserta(curvestatus(H1,fc(Name2,Group),Name1)),
                retractall(ptbound).

(231) grouppcurve4(Name,Group):-
                ss1(Name,Group,p(X1,Y1,Z1),p(X2,Y2,Z2),p(X3,Y3,Z3)),
                L1 = sqrt((X2-X1)*(X2-X1)+(Y2-Y1)*(Y2-Y1)+(Z2-Z1)*(Z2-Z1)),
                L2 = sqrt((X1-X3)*(X1-X3)+(Y1-Y3)*(Y1-Y3)+(Z1-Z3)*(Z1-Z3)),
                L3 = sqrt((X3-X2)*(X3-X2)+(Y3-Y2)*(Y3-Y2)+(Z3-Z2)*(Z3-Z2)),
                midpoint(Xm,Ym,Zm),
                L4 = sqrt((X1-Xm)*(X1-Xm)+(Y1-Ym)*(Y1-Ym)+(Z1-Zm)*(Z1-
                Zm)),
                L5 = sqrt((X2-Xm)*(X2-Xm)+(Y2-Ym)*(Y2-Ym)+(Z2-Zm)*(Z2-
                Zm)),
                L6 = sqrt((X3-Xm)*(X3-Xm)+(Y3-Ym)*(Y3-Ym)+(Z3-Zm)*(Z3-
                Zm)),
                C1 = (L3*L3-L2*L2+L1*L1)/(2*L1),
                Ht = (L3*L3-C1*C1),
                checkzero(Ht,H),
                AT = L1*H/2,
                C11 = (L4*L4-L5*L5+L1*L1)/(2*L1),
                Ht1 = (L4*L4-C11*C11),
                checkzero(Ht1,H1),
                A1 = L1*H1/2,
                C12 = (L4*L4-L6*L6+L2*L2)/(2*L2),
                Ht2 = (L4*L4-C12*C12),
                checkzero(Ht2,H2),
                A2 = L2*H2/2,
                C13 = (L5*L5-L6*L6+L3*L3)/(2*L3),
                Ht3 = (L5*L5-C13*C13),
                checkzero(Ht3,H3),
                A3 = L3*H3/2,
                AT < (A1+A2+A3)+0.001,
                AT > (A1+A2+A3)-0.001,
                asserta(ptbound);
                true.

(232) secshared:-
                status(Name1,PList,_),
                not(used(Name1)),
                PList = [_|_],
                asserta(used(Name1)),
                secshared1(Name1,PList,Plist),
                secshared;
                status(Name1,PList,_),
                not(used(Name1)),
                PList = [],
                asserta(used(Name1)),
                secshared;
                retractall(used(_)).

(233) secshared1(_,[],_).
(234) secshared1(Name1,[H|T],PList):-
                H = pair(Name,Group),
                scurvelist(Name,Group,List,_),
                secshared2(Name1,Name,Group,List,PList),
                secshared1(Name1,T,PList).

(235) secshared2(_,_,_,_,[]).
(236) secshared2(N,Name,Group,List,[H1|T1]):-
                H1 = pair(Name1,Group1),
                Name1 = Name,
                Group1 = Group,
                secshared2(N,Name,Group,List,T1);
                H1 = pair(Name1,Group1),
                scurvelist(Name1,Group1,List1,_),
                secshared3(N,List,List1),
                secshared2(N,Name,Group,List,T1).

(237) secshared3(_,[],_).
(238) secshared3(N,[H|T],List1):-
                member(H,List1),
                secshared4(N,H),
                secshared3(N,T,List1);
                secshared3(N,T,List1).

(239) secshared4(N,H):-
                retract(curvestatus(H,_,N)),
                secshared4(N,H);
                asserta(curvestatus(H,shared,N)).

(240) makefreesurf:-
                status(Name1,List2,_),
                not(used(Name1)),
                List2 = [_|_],
                asserta(used(Name1)),
                asserta(tempsob(Name1,[])),
                asserta(tempsib(Name1,[])),
                makefreesurf1(Name1,List2),!,
                makefreesurf;
                status(Name1,List2,_),
                not(used(Name1)),
                List2 = [],
                asserta(used(Name1)),
                makefreesurf;
                retractall(used(_)).
                true.

(241) makefreesurf1(Name1,[]):-
                pob(Name1,List1),
                makefreesurf2a(Name1,List1),!.
(242) makefreesurf1(Name1,[H|T]):-
                H = pair(Name,Group),
                sob(Name,Group,List),
                makefreesurfsob(Name1,List),!,
                makefreesurfsib(Name1,Name,Group),!,
                makefreesurf1(Name1,T).

(243) makefreesurfsob(_,[]).
(244) makefreesurfsob(Name1,[H|T]):-
                tempsob(Name1,List),
                not(member(H,List)),
                List1 = [H|List],
                retract(tempsob(Name1,List)),
                asserta(tempsob(Name1,[H|List1])),
                makefreesurfsob(Name1,T);
                makefreesurfsob(Name1,T).

(245) makefreesurfsib(Name1,Name,Group):-
                sib(Name,Group,P,List),
                not(used1(P)),
                asserta(used1(P)),
                makefreesurfsib1(Name1,List),!,
                makefreesurfsib(Name1,Name,Group);
                retractall(used1(_)).

(246) makefreesurfsib1(_,[]).
(247) makefreesurfsib1(Name1,[H|T]):-
                tempsib(Name1,List),
                not(member(H,List)),
                List1 = [H|List],
                retract(tempsib(Name1,List)),
                asserta(tempsib(Name1,[H|List1])),
                makefreesurfsib1(Name1,T);
                makefreesurfsib1(Name1,T).

(248) makefreesurf2a(Name1,[]):-
                pob(Name1,List1),
                List1 = [_|_],
                makefreesurf2b(Name1,List1),!;
                pre_checkinside(Name1),!.
(249) makefreesurf2a(Name1,[H|T1]):-
                curvestatus(H1,nc,Name1),
                checkforfreesurf(Name1,K),
                c(H1,P1,P2),
                curvedel(Name1,H1),
                asserta(templine(H1,P1,P2)),
                asserta(freesurf(Name1,K,[H1|[]])),
                makefreesurf3(Name1,K),!,
                retractall(templine(_,_,_)),
                pob(Name1,List),
                makefreesurf2a(Name1,List);
                makefreesurf2a(Name1,T1).

(250) makefreesurf2b(Name1,[]):-
                pre_checkinside(Name1),!.
(251) makefreesurf2b(Name1,[H|T1]):-
                curvestatus(H1,shared,Name1),
                sib1(Name,Group,_,SList),
                member(H1,SList),
                status(Name1,GList,_),
                checkgroup(Name,Group,GList),
                checkforfreesurf(Name1,K),
                c(H1,P1,P2),
                curvedel(Name1,H1),
                asserta(templine(H1,P1,P2)),
                asserta(freesurf(Name1,K,[H1|[]])),
                makefreesurf3(Name1,K),!,
                retractall(templine(_,_,_)),
                pob(Name1,List),
                makefreesurf2b(Name1,List);
                makefreesurf2b(Name1,T1).

(252) makefreesurf3(Name1,K):-
                makefreesurf4(Name1,K,nc),
                not(end),
                makefreesurf3(Name1,K),!;
                not(end),
                makefreesurf4(Name1,K,bc(_,_)),
                not(end),
                makefreesurf3(Name1,K),!;
                not(end),
                makefreesurf4(Name1,K,shared),
                not(end),
                makefreesurf3(Name1,K),!;
                end,
                retractall(end).

(253) makefreesurf4(Name1,K,Type):-
                templine(H1,P1,P2),
```

```
                freesurf(Name1,K,List).
                c(C,P2,P3).
                not(member(C,List)).
                checkmember(Name1,C).
                C <> H1.
                P1 = P3.
                curvestatus(C,Type,Name1).
                curvedel(Name1,C).
                retract(freesurf(Name1,K,List)).
                asserta(freesurf(Name1,K,[C|List])).
                asserta(end).!.
(254) makefreesurf4(Name1,K,Type):-
                templine(H1,P1,P2).
                freesurf(Name1,K,List).
                c(C,P3,P2).
                not(member(C,List)).
                checkmember(Name1,C).
                C <> H1.
                P1 = P3.
                curvestatus(C,Type,Name1).
                curvedel(Name1,C).
                retract(freesurf(Name1,K,List)).
                asserta(freesurf(Name1,K,[C|List])).
                asserta(end).!.
(255) makefreesurf4(Name1,K,Type):-
                templine(H1,P1,P2).
                freesurf(Name1,K,List).
                c(C,P2,P3).
                not(member(C,List)).
                checkmember(Name1,C).
                C <> H1.
                curvestatus(C,Type,Name1).
                curvedel(Name1,C).
                retract(freesurf(Name1,K,List)).
                asserta(freesurf(Name1,K,[C|List])).
                retract(templine(H1,P1,P2)).
                asserta(templine(C,P1,P3)).!.
(256) makefreesurf4(Name1,K,Type):-
                templine(H1,P1,P2).
                freesurf(Name1,K,List).
                c(C,P3,P2).
                not(member(C,List)).
                checkmember(Name1,C).
                C <> H1.
                curvestatus(C,Type,Name1).
                curvedel(Name1,C).
                retract(freesurf(Name1,K,List)).
                asserta(freesurf(Name1,K,[C|List])).
                retract(templine(H1,P1,P2)).
                asserta(templine(C,P1,P3)).!.

(257) checkgroup(_,_,[]):-
                fail.
(258) checkgroup(Name,Group,[H|T]):-
                H = pair(Name,Group):
                checkgroup(Name,Group,T).

(259) checkforfreesurf(Name1,K):-
                freesurf(Name1,N,_).
                frontstr(1,N,_,Ns).
                str_int(Ns,Ni).
                New = Ni+1.
                str_int(News,New).
                retract(lastfree(Name1,_)).
                asserta(lastfree(Name1,New)).
                concat("L",News,K):
                asserta(lastfree(Name1,1)).
                K = "L1".

(260) curvedel(Name1,C):-
                pob(Name1,List1).
                member(C,List1).
                subtract(C,List1,NList1).
                retract(pob(Name1,List1)).
                asserta(pob(Name1,NList1)):
                pib(Name1,P,List1).
                member(C,List1).
                subtract(C,List1,NList1).
                retract(pib(Name1,P,List1)).
                asserta(pib(Name1,P,NList1)):
                tempsob(Name1,List).
                member(C,List).
                subtract(C,List,Nlist).
                retract(tempsob(Name1,List)).
                asserta(tempsob(Name1,NList)):
                tempsib(Name1,List).
                member(C,List).
                subtract(C,List,Nlist).
                retract(tempsib(Name1,List)).
                asserta(tempsib(Name1,NList)).

(261) checkmember(Name1,C):-
                pob(Name1,List).
                member(C,List):
                pib(Name1,_,List).
                member(C,List):
                tempsob(Name1,List).
                member(C,List):
                tempsib(Name1,List).
                member(C,List).
```

```
(262) pre_checkinside(Name):-
                freesurf(Name,K,_).
                not(used1(K)).
                asserta(used1(K)).
                status(Name,[H|T],_).
                pre_checkinside1(Name,[H|T],K).
                pre_checkinside(Name):
                retractall(lastlist(_)).
                retractall(lastlist2(_)).
                retractall(used1(_)).

(263) pre_checkinside1(Name,[],K):-
                not(newprp).
                checkinside(Name,K).!:
                newprp.
                retractall(newprp).
                retract(prp(Name,X2,Y2,Z2)).
                Nx = 1.10*X2.
                Ny = 1.15*Y2.
                Nz = 1.20*Z2.
                asserta(prp(Name,Nx,Ny,Nz)).
                lastlist2(List).
                pre_checkinside1(Name,List,K).
(264) pre_checkinside1(Name,[H|T],K):-
                not(newprp).
                H = pair(Name1,Group).
                sob1(Name1,Group,[H1|T1]).
                checkmember1(Name,K,[H1|T1]).
                not(common).
                sob(Name1,Group,[H1|_]).
                freesurf(Name,K,List).
                asserta(counter(0)).
                checkinside1(Name,K,H1,b(Name1,Group),List).!.
                retractall(lastlist2(_)).
                asserta(lastlist2([H|T])).
                pre_checkinside1(Name,T,K):
                not(newprp).
                retractall(common).
                pre_checkinside1(Name,T,K):
                newprp.
                retractall(newprp).
                retract(prp(Name,X2,Y2,Z2)).
                Nx = 1.10*X2.
                Ny = 1.15*Y2.
                Nz = 1.20*Z2.
                asserta(prp(Name,Nx,Ny,Nz)).
                lastlist2(List).
                pre_checkinside1(Name,List,K).

(265) checkinside(Name,K):-
                not(newprp).
                pib1(Name,P,[H1|T1]).
                not(used2(P)).
                asserta(used2(P)).
                checkmember1(Name,K,[H1|T1]).
                not(common).
                pib(Name,P,[H|_]).
                freesurf(Name,K,List).
                asserta(counter(0)).
                checkinside1(Name,K,H,a(P),List).!.
                checkinside(Name,K):
                not(newprp).
                not(common).
                retractall(used2(_)):
                not(newprp).
                common.
                retractall(common).
                checkinside(Name,K):
                newprp.
                retractall(newprp).
                used2(P).
                retract(used2(P)).
                retract(prp(Name,X2,Y2,Z2)).
                Nx = 1.10*X2.
                Ny = 1.15*Y2.
                Nz = 1.20*Z2.
                asserta(prp(Name,Nx,Ny,Nz)).
                checkinside(Name,K).

(266) checkinside1(Name,K,_,PS,[]):-
                not(newprp).
                counter(Num).
                checknum(Num,Status).
                Status = "Odd".
                retractall(ipoint(_,_,_)).
                retractall(counter(_)).
                checkinside2(Name,K,PS),!:
                not(newprp).
                retractall(ipoint(_,_,_)).
                retractall(counter(_)):
                newprp.
(267) checkinside1(Name,K,H,PS,[H1|T1]):-
                not(newprp).
                c(H,p(X1,Y1,Z1),_).
                prp(Name,X2,Y2,Z2).
                c(H1,p(X3,Y3,Z3),p(X4,Y4,Z4)).
                asserta(c1(H,p(X1,Y1,Z1),p(X2,Y2,Z2))).
                asserta(c2(H1,p(X3,Y3,Z3),p(X4,Y4,Z4))).
                intersection(X1,Y1,Z1,X2,Y2,Z2,X3,Y3,Z3,X4,Y4,Z4,"Ignore").!.
                retract(c1(H,p(X1,Y1,Z1),p(X2,Y2,Z2))).
                retract(c2(H1,p(X3,Y3,Z3),p(X4,Y4,Z4))).
```

```
                retractall(end1).
                checkinside1(Name,K,H,PS,T1):
                newprp.

(268) checkinside2(Name,K,a( P)):-
                pib(Name,P,List).
                checkinside2a(Name,K,List,List).!.
(269) checkinside2(Name,K,b(Name1,Group)):-
                sob(Name1,Group,List).
                checkinside2b(Name,K,List,List).!.

(270) checkinside2a(Name,K,[],List):-
                checkinside2b(Name,K,List,List).!.
(271) checkinside2a(Name,K,[HIT],List):-
                curvestatus(H,nc,Name).
                freesurf(Name,K,FList).
                not(member(H,FList)).
                c(H,P1,P2).
                curvedel(Name,H).
                retract(freesurf(Name,K,FList)).
                asserta(freesurf(Name,K,[HIFList])).
                asserta(templine(H,P1,P2)).
                makefreesurf3(Name,K).!.
                retractall(templine(_,_,_)):
                checkinside2a(Name,K,T,List).

(272) checkinside2b(Name,K,[],List):-
                checkinside2c(Name,K,List,List).!.
(273) checkinside2b(Name,K,[HIT],List):-
                curvestatus(H,bc(_,_).Name).
                freesurf(Name,K,FList).
                not(member(H,FList)).
                c(H,P1,P2).
                curvedel(Name,H).
                retract(freesurf(Name,K,FList)).
                asserta(freesurf(Name,K,[HIFList])).
                asserta(templine(H,P1,P2)).
                makefreesurf3(Name,K).!.
                retractall(templine(_,_,_)):
                checkinside2b(Name,K,T,List).

(274) checkinside2c(_,_,[],_).
(275) checkinside2c(Name,K,[HIT],List):-
                curvestatus(H,shared,Name).
                freesurf(Name,K,FList).
                not(member(H,FList)).
                c(H,P1,P2).
                curvedel(Name,H).
                retract(freesurf(Name,K,FList)).
                asserta(freesurf(Name,K,[HIFList])).
                asserta(templine(H,P1,P2)).
                makefreesurf3(Name,K).!.
                retractall(templine(_,_,_)):
                checkinside2c(Name,K,T,List).

(276) checkmember1(_,_,[]).
(277) checkmember1(Name,K,[H1IT1]):-
                freesurf(Name,K,List).
                not(member(H1,List)).
                checkmember1(Name,K,T1):
                asserta(common).

(278) freesurfsib:-
                status(Name1,List2,_).
                not(used(Name1)).
                List2 = [_|_].
                asserta(used(Name1)).
                freesurfsib1(Name1,List2).!.
                freesurfsib:
                status(Name1,List2,_).
                not(used(Name1)).
                List2 = [].
                asserta(used(Name1)).
                freesurfsib:
                retractall(used(_)).
                true.

(279) freesurfsib1(_,[]).
(280) freesurfsib1(Name1,[HIT]):-
                H = pair(Name,Group).
                freesurfsib2(Name1,Name,Group).
                freesurfsib1(Name1,T).

(281) freesurfsib2(Name1,Name,Group):-
                sib(Name,Group,S,[HIT]).
                not(used1(S)).
                asserta(used1(S)).
                checkprefree1(Name1,H,[HIT]),
                freesurfsib2(Name1,Name,Group);
                retractall(used1(_)).

(282) checkprefree1(Name1,H,List):-
                not(common).
                freesurf(Name1,K,List1).
                not(used2(K)).
                asserta(used2(K)).
                checkprefree2(List,List1).!.
                checkprefree1(Name1,H,List):
                not(common).
                retractall(used2(_)).
                curvedel(Name1,H).
                freesurfsib3(Name1,H);
                retractall(used2(_)).
                retractall(common).

(283) checkprefree2([],_).
(284) checkprefree2([HIT],List1):-
                not(member(H,List1)).
                checkprefree2(T,List1):
                asserta(common).

(285) freesurfsib3(Name1,H):-
                c(H,P1,P2).
                lastfree(Name1,Num).
                New = Num+1.
                str_int(News,New).
                concat("L",News,K).
                asserta(freesurf(Name1,K,[H|[]])).
                asserta(tempfree(Name1,K)).
                asserta(templine(H,P1,P2)).
                makefreesurf3(Name1,K).!.
                retractall(templine(_,_,_)).

(286) insidesib:-
                tempfree(Name,Kt),
                retract(tempfree(Name,Kt)).
                insidesib1(Name,Kt).
                insidesib;
                true.

(287) insidesib1(Name,Kt):-
                not(newprp).
                pib(Name,P,PList).
                not(used(P)).
                asserta(used(P)).
                checkprefree3(Name,Kt,PList).!.
                insidesib1(Name,Kt);
                not(newprp).
                retractall(lastlist(_)).
                retractall(used(_)).
                newprp.
                retractall(newprp).
                used(P).
                retract(used(P)).
                retract(prp(Name,X2,Y2,Z2)).
                Nx = 1.10*X2.
                Ny = 1.15*Y2.
                Nz = 1.20*Z2.
                asserta(prp(Name,Nx,Ny,Nz)).
                insidesib1(Name,Kt).

(288) checkprefree3(Name,Kt,[HIT]):-
                not(newprp).
                not(common).
                freesurf(Name,K,FList).
                not(used1(K)).
                asserta(used1(K)).
                checkprefree4([HIT],FList).!.
                checkprefree3(Name,Kt,[HIT]):
                not(newprp).
                not(common).
                retractall(used1(_)).
                freesurf(Name,Kt,List).
                asserta(counter(0)).
                insidesib2(Name,Kt,List,H).!:
                retractall(common).
                retractall(used1(_)).

(289) checkprefree4([],_).
(290) checkprefree4([HIT],FList):-
                not(member(H,FList)).
                checkprefree4(T,FList):
                asserta(common).

(291) insidesib2(Name,Kt,[],H):-
                not(newprp).
                counter(Num).
                checknum(Num,Status).
                Status = "Odd".
                retractall(ipoint(_,_,_)).
                retractall(counter(_)).
                insidesib3(Name,Kt,H).!:
                not(newprp).
                retractall(ipoint(_,_,_)).
                retractall(counter(_)):
                newprp.
(292) insidesib2(Name,Kt,[H1IT1],H):-
                not(newprp).
                prp(Name,X1,Y1,Z1).
                c(H,p(X2,Y2,Z2),_).
                c(H1,p(X3,Y3,Z3),p(X4,Y4,Z4)).
                asserta(c1(H,p(X1,Y1,Z1),p(X2,Y2,Z2))).
                asserta(c2(H1,p(X3,Y3,Z3),p(X4,Y4,Z4))).
                intersection(X1,Y1,Z1,X2,Y2,Z2,X3,Y3,Z3,X4,Y4,Z4,"Ignore").!.
                retract(c2(H1,p(X3,Y3,Z3),p(X4,Y4,Z4))).
                retract(c1(H,p(X1,Y1,Z1),p(X2,Y2,Z2))).
                retractall(end1).
                insidesib2(Name,Kt,T1,H):
                newprp.

(293) insidesib3(Name,Kt,H):-
                c(H,P1,P2).
```

```
                curvedel(Name,H).
                retract(freesurf(Name,Kt,List)).
                asserta(freesurf(Name,Kt,[Hl|List])).
                asserta(templine(H,P1,P2)).
                makefreesurf3(Name,Kt).!.
                retract(templine(H,P1,P2)).

(294) overlapsurf:-
                status(Name1,List2,_).
                not(used(Name1)).
                List2 = [_|_].
                asserta(used(Name1)).
                pob(Name1,List1).
                overlapsurf1a(Name1,List1).!.
                overlapsurf:
                status(Name1,List2,_).
                not(used(Name1)).
                List2 = [].
                asserta(used(Name1)).
                overlapsurf:
                retractall(used(_)).
                true.

(295) overlapsurf1a(Name1,[]):-
                pob(Name1,List1).
                overlapsurf1b(Name1,List1).!.
(296) overlapsurf1a(Name1,[H1|T1]):-
                curvestatus(H1,fc(N,G),Name1).
                curvedel(Name1,H1).
                checkforconsurf(N,G,New).
                asserta(consurf(N,G,New,[H1|[]])).
                c(H1,P1,P2).
                asserta(templine(H1,P1,P2)).
                overlapsurf2(Name1,N,G,New).!.
                retractall(templine(_,_,_)).
                pob(Name1,NList1).
                overlapsurf1a(Name1,NList1):
                overlapsurf1a(Name1,T1).

(297) overlapsurf1b(Name1,[]):-
                tempsob(Name1,List).
                List = [_|_].
                overlapsurf4a(Name1,List).!:
                true.
(298) overlapsurf1b(Name1,[H1|T1]):-
                curvestatus(H1,shared,Name1).
                getsob(Name1,H1,N,G).
                curvedel(Name1,H1).
                checkforconsurf(N,G,New).
                asserta(consurf(N,G,New,[H1|[]])).
                c(H1,P1,P2).
                asserta(templine(H1,P1,P2)).
                overlapsurf2(Name1,N,G,New).!.
                retractall(templine(_,_,_)).
                pob(Name1,NList1).
                overlapsurf1b(Name1,NList1):
                overlapsurf1b(Name1,T1).

(299) getsob(Name,H,N,G):-
                status(Name,List,_).
                getsob1(H,N,G,List).

(300) getsob1(_,_,_,[]):-
                fail.
(301) getsob1(H,N,G,[pair(N,G)|T]):-
                sob1(N,G,List).
                member(H,List):
                getsob1(H,N,G,T).

(302) overlapsurf2(Name1,N,G,New):-
                overlapsurf3(Name1,N,G,New,fc(N,G)).
                not(end).
                overlapsurf2(Name1,N,G,New).!:
                not(end).
                overlapsurf3(Name1,N,G,New,bc(N,G)).
                not(end).
                overlapsurf2(Name1,N,G,New).!:
                not(end).
                overlapsurf3(Name1,N,G,New,shared).
                not(end).
                overlapsurf2(Name1,N,G,New).!:
                end.
                retractall(end).

(303) overlapsurf3(Name1,N,G,New,Type):-
                templine(H1,P1,P2).
                consurf(N,G,New,List).
                c(C,P2,P3).
                not(member(C,List)).
                checkmember2(Name1,C,N,G).
                C <> H1.
                P1 = P3.
                curvestatus(C,Type,Name1).
                curvedel(Name1,C).
                retract(consurf(N,G,New,List)).
                asserta(consurf(N,G,New,[C|List])).
                asserta(end).!.
(304) overlapsurf3(Name1,N,G,New,Type):-
                templine(H1,P1,P2).
                consurf(N,G,New,List).
                c(C,P3,P2).
                not(member(C,List)).
                checkmember2(Name1,C,N,G).
                C <> H1.
                P1 = P3.
                curvestatus(C,Type,Name1).
                curvedel(Name1,C).
                retract(consurf(N,G,New,List)).
                asserta(consurf(N,G,New,[C|List])).
                asserta(end).!.
(305) overlapsurf3(Name1,N,G,New,Type):-
                templine(H1,P1,P2).
                consurf(N,G,New,List).
                c(C,P2,P3).
                not(member(C,List)).
                checkmember2(Name1,C,N,G).
                C <> H1.
                curvestatus(C,Type,Name1).
                curvedel(Name1,C).
                retract(consurf(N,G,New,List)).
                asserta(consurf(N,G,New,[C|List])).
                retract(templine(H1,P1,P2)).
                asserta(templine(C,P1,P3)).!.
(306) overlapsurf3(Name1,N,G,New,Type):-
                templine(H1,P1,P2).
                consurf(N,G,New,List).
                c(C,P3,P2).
                not(member(C,List)).
                checkmember2(Name1,C,N,G).
                C <> H1.
                curvestatus(C,Type,Name1).
                curvedel(Name1,C).
                retract(consurf(N,G,New,List)).
                asserta(consurf(N,G,New,[C|List])).
                retract(templine(H1,P1,P2)).
                asserta(templine(C,P1,P3)).!.

(307) overlapsurf4a(Name1,[]):-
                tempsob(Name1,List).
                overlapsurf4b(Name1,List).!.
(308) overlapsurf4a(Name1,[H|T]):-
                curvestatus(H,bc(N,G),Name1).
                checkforconsurf(N,G,New).
                asserta(consurf(N,G,New,[H|[]])).
                curvedel(Name1,H).
                c(H,P1,P2).
                asserta(templine(H,P1,P2)).
                overlapsurf2(Name1,N,G,New).!.
                retractall(templine(_,_,_)).
                tempsob(Name1,NList).
                overlapsurf4a(Name1,NList):
                overlapsurf4a(Name1,T).

(309) overlapsurf4b(_,[]).
(310) overlapsurf4b(Name1,[H|T]):-
                curvestatus(H,shared,Name1).
                sob1(N,G,L).
                not(consurf(N,G,_,_)).
                member(H,L).
                checkforconsurf(N,G,New).
                asserta(consurf(N,G,New,[H|[]])).
                curvedel(Name1,H).
                c(H,P1,P2).
                asserta(templine(H,P1,P2)).
                overlapsurf2(Name1,N,G,New).!.
                retractall(templine(_,_,_)).
                tempsob(Name1,NList).
                overlapsurf4b(Name1,NList):
                overlapsurf4b(Name1,T).

(311) checkforconsurf(N,G,New):-
                consurf(N,G,Name,_).
                frontstr(1,Name,_,Num).
                str_int(Num,No).
                No1 = No+1,
                str_int(No1s,No1).
                concat("C",No1s,New):
                New = "C1".

(312) checkmember2(Name1,C,N,G):-
                pob(Name1,List).
                member(C,List):
                pib(Name1,_,List).
                member(C,List):
                tempsob(Name1,List).
                member(C,List).
                sob1(N,G,L).
                member(C,L):
                tempsib(Name1,List).
                member(C,List).

(313) insideconsurf:-
                status(Name1,List2,_).
                not(used(Name1)).
                List2 = [_|_].
                asserta(used(Name1)).
                insideconsurf1(Name1,List2).!.
                insideconsurf:
                status(Name1,List2,_).
                not(used(Name1)).
                List2 = [].
                asserta(used(Name1)).
```

```
                insideconsurf;
                retractall(used(_)).
                true.

(314) insideconsurf1(_,[]).
(315) insideconsurf1(Name1,[H2|T2]):-
                H2 = pair(N,G),
                insideconsurf2(Name1,N,G),!,
                insideconsurf1(Name1,T2).

(316) insideconsurf2(Name1,N,G):-
                consurf(N,G,K,_),
                not(used1(K)),
                asserta(used1(K)),
                insideconsurf3a(Name1,N,G,K),!,
                insideconsurf2(Name1,N,G);
                retractall(used1(_)),
                insideconsurf4(Name1,N,G),!.

(317) insideconsurf3a(Name1,N,G,K):-
                pib(Name1,P,PList),
                not(used2(P)),
                asserta(used2(P)),
                PList = [_|_],
                insideconsurf3b(Name1,N,G,K,P,PList),!,
                insideconsurf3a(Name1,N,G,K);
                retractall(used2(_)).

(318) insideconsurf3b(Name1,N,G,K,P,[]):-
                pib(Name1,P,PList),
                insideconsurf3c(Name1,N,G,K,P,PList).
(319) insideconsurf3b(Name1,N,G,K,P,[H|T]):-
                curvestatus(H,fc(N,G),Name1),
                c(H,P1,P2),
                curvedel(Name1,H),
                retract(consurf(N,G,K,List)),
                asserta(consurf(N,G,K,[H|List])),
                asserta(templine(H,P1,P2)),
                overlapsurf2(Name1,N,G,K),!,
                retractall(templine(_,_,_)),
                pib(Name1,P,NList),
                insideconsurf3b(Name1,N,G,K,P,NList);
                insideconsurf3b(Name1,N,G,K,P,[]).

(320) insideconsurf3c(_,_,_,_,_,_,[]).
(321) insideconsurf3c(Name1,N,G,K,P,[H|T]):-
                curvestatus(H,shared,Name1),
                sib1(N,G,_,L),
                member(H,L),
                c(H,P1,P2),
                curvedel(Name1,H),
                retract(consurf(N,G,K,List)),
                asserta(consurf(N,G,K,[H|List])),
                asserta(templine(H,P1,P2)),
                overlapsurf2(Name1,N,G,K),!,
                retractall(templine(_,_,_)),
                pib(Name1,P,NList),
                insideconsurf3c(Name1,N,G,K,P,NList);
                insideconsurf3c(Name1,N,G,K,P,[]).

(322) insideconsurf4(Name1,N,G):-
                consurf(N,G,K,_),
                not(used1(K)),
                asserta(used1(K)),
                insideconsurf5(Name1,N,G,K),!,
                insideconsurf4(Name1,N,G);
                retractall(used1(_)).

(323) insideconsurf5(Name1,N,G,K):-
                sib(N,G,S,[H|_]),
                not(used2(S)),
                asserta(used2(S)),
                curvestatus(H,bc(N,G),Name1),
                c(H,P1,P2),
                curvedel(Name1,H),
                retract(consurf(N,G,K,List)),
                asserta(consurf(N,G,K,[H|List])),
                asserta(templine(H,P1,P2)),
                overlapsurf2(Name1,N,G,K),!,
                retractall(templine(_,_,_)),
                insideconsurf5(Name1,N,G,K);
                retractall(used2(_)).

(324) subtract(H,[H|Rest],Rest).
(325) subtract(H,[Y|Rest],[Y|Rest1]):-
                subtract(H,Rest,Rest1).

(326) subtract1(H,[H|Rest],Rest).
(327) subtract1(H,[Y|Rest],[Y|Rest1]):-
                subtract1(H,Rest,Rest1).

(328) member(Name,[Name|_]).
(329) member(Name,[_|Rest]):-
                member(Name,Rest).

(330) member1(Pair,[Pair|_]).
(331) member1(Pair,[_|Rest]):-
                member1(Pair,Rest).
```

A2.4 SCRIPT GENERATION

```
(1) createcsys([],_):-
                openwrite(csys,"C:\\Mscn4w46\\solver\\bin\\csys.prg").
                writedevice(csys).
                write("\n$ File Program Run Script").nl.
                write("{Fmu}C:\\\\Mscn4w46\\\\solver\\\\bin\\\\m1.BAS<OK>").nl.
                closefile(csys).
                writedevice(screen).
(2) createcsys([H|T],List):-
                H = e(_,_,_,_,_,_,_,_,rot(_,_,_,_,_,_)).
                createcsys1(List):
                H = e(_,_,_,_,_,_,_,_,tran(_,_,_,_,_,_)).
                createcsys1(List):
                createcsys(T,List).

(3) createcsys1(List):-
                big(Num),
                X = Num*3.
                Y = Num*3.2.
                Z = Num*3.5.
                str_real(Xs,X).
                str_real(Ys,Y).
                str_real(Zs,Z).
                asserta(point3(Xs,Ys,Zs)).
                openwrite(csys,"C:\\Mscn4w46\\solver\\bin\\csys.prg").
                writedevice(csys).
                createcsys2(List),!.
                closefile(csys).
                writedevice(screen).

(4) createcsys2([]):-
                write("\n$ File Program Run Script").nl.
                write("{Fmu}C:\\\\Mscn4w46\\\\solver\\\\bin\\\\m1.BAS<OK>").nl.
(5) createcsys2([H|T]):-
                H = e(_,_,_,_,_,_,_,_,Type).
                createcsys3(Type),!.
                createcsys2(T).

(6) createcsys3(fixed).
(7) createcsys3(extload).
(8) createcsys3(rot(X1,Y1,Z1,X2,Y2,Z2)):-
                last(Old).
                New = Old + 1.
                asserta(last(New)).
                asserta(rc(New,rot(X1,Y1,Z1,X2,Y2,Z2))).
                str_real(X1s,X1).
                str_real(Y1s,Y1).
                str_real(Z1s,Z1).
                str_real(X2s,X2).
                str_real(Y2s,Y2).
                str_real(Z2s,Z2).
                point3(X3,Y3,Z3).nl.
                write("$ Model Coord Sys").nl.
                concat("{My}<A-Z><@14002><PUSH><OK><A-X>".X1s,A1).
                concat(A1."<A-Y>".A2).
                concat(A2,Y1s,A3).
                concat(A3."<A-Z>".A4).
                concat(A4,Z1s,A5).
                concat(A5."<OK><A-X>".A6).
                concat(A6,X2s,A7).
                concat(A7."<A-Y>".A8).
                concat(A8,Y2s,A9).
                concat(A9."<A-Z>".A10).
                concat(A10,Z2s,A11).
                concat(A11,"<OK><A-X>".A12).
                concat(A12,X3,A13).
                concat(A13,"<A-Y>".A14).
                concat(A14,Y3,A15).
                concat(A15,"<A-Z>".A16).
                concat(A16,Z3,A17).
                concat(A17,"<OK><Esc>".A18).
                write(A18).nl.
(9) createcsys3(tran(X1,Y1,Z1,X2,Y2,Z2)):-
                last(Old).
                New = Old + 1.
                asserta(last(New)).
                asserta(rc(New,tran(X1,Y1,Z1,X2,Y2,Z2))).
                str_real(X1s,X1).
                str_real(Y1s,Y1).
                str_real(Z1s,Z1).
                str_real(X2s,X2).
                str_real(Y2s,Y2).
                str_real(Z2s,Z2).
                point3(X3,Y3,Z3).nl.
                write("$ Model Coord Sys").nl.
                concat("{My}<A-Z><OK><A-X>".X1s,A1).
                concat(A1,"<A-Y>".A2).
                concat(A2,Y1s,A3).
                concat(A3."<A-Z>".A4).
                concat(A4,Z1s,A5).
                concat(A5."<OK><A-X>".A6).
                concat(A6,X2s,A7).
                concat(A7."<A-Y>".A8).
```

```
                concat(A8.Y2s.A9).
                concat(A9."<A-Z>".A10).
                concat(A10.Z2s.A11).
                concat(A11."<OK><A-X>".A12).
                concat(A12.X3.A13).
                concat(A13."<A-Y>".A14).
                concat(A14.Y3.A15).
                concat(A15."<A-Z>".A16).
                concat(A16.Z3.A17).
                concat(A17."<OK><Esc>".A18).
                write(A18).nl.

(10) senddata(Name):-
                openwrite(nastran,"C:\\Mscn4w46\\solver\\bin\\m1.bas").
                writedevice(nastran).
                entity(ID,Name,_,_,_,_,_).
                retractall(used(_)).
                retractall(used1(_)).
                write("Sub Main").nl.nl.
                predimpoint.!.nl.
                asserta(last(0)).
                predimcurve.!.nl.
                asserta(last(0)).
                dimfreesurface.!.
                asserta(last(0)).
                dimconsurface.!.
                dimmatprop.!.nl.
                dimmesh.!.nl.
                dimsolidmesh.!.
                asserta(lastl(0)).
                predimload(Name,ID).!.
                asserta(lastc(0)).
                predimconstraint(Name,ID).!.
                startpoint.!.nl.
                startcurve.!.nl.
                startfreesurface.!.
                startconsurface.!.
                startmatprop(Name).!.nl.
                asserta(last(0)).
                asserta(avlength(0.0)).
                curvelength.!.
                startmeshsurf(Name).!.nl.
                startsolidmesh.!.
                startload(ID).!.
                startconstraint(ID).!.
                write("program=esp_FileProgramRun(\34c:\\mscn4w46\\solver\\bin\\E
                ndofdat.prg\34)").nl.nl.
                write("End Sub").
                writedevice(screen).
                closefile(nastran).
                system("c:\\mscn4w46\\mscn4w46",0,_).

(11) predimpoint:-
                freesurf(N,N1,List).
                asserta(pair1(N,N1)).
                dimpoint(List):
                consurf(N,G,N1,List).
                asserta(tri(N,G,N1)).
                dimpoint(List).

(12) dimpoint([H|T]):-
                c(H,P1,P2).
                asserta(point("p1",P1)).
                asserta(point("p2",P2)).
                write("Dim p1 as esp_Coord").nl.
                write("Dim p2 as esp_Coord").nl.
                dimpoint1(T).

(13) dimpoint1([]):-
                freesurf(N,N1,List).
                not(pair1(N,N1)).
                asserta(pair1(N,N1)).
                dimpoint1(List):
                consurf(N,G,N1,List).
                not(tri(N,G,N1)).
                asserta(tri(N,G,N1)).
                dimpoint1(List):
                retractall(pair1(_,_)).
                retractall(tri(_,_,_)).
                true.
(14) dimpoint1([H|T]):-
                c(H,P1,P2).
                not(point(_,P1)).
                not(point(_,P2)).
                point(Last,_).
                frontstr(1,Last,_,Nos).
                str_real(Nos,Nor).
                Nor1 = Nor+1.
                Nor2 = Nor1+1.
                str_real(Nos1,Nor1).
                concat("p",Nos1,NewP1).
                str_real(Nos2,Nor2).
                concat("p",Nos2,NewP2).
                concat("Dim ",NewP1,Sub1).
                concat(Sub1," as esp_Coord",TotP1).
                concat("Dim ",NewP2,Sub2).
                concat(Sub2," as esp_Coord",TotP2).
                asserta(point(NewP1,P1)).
                asserta(point(NewP2,P2)).
                write(TotP1).nl.
                write(TotP2).nl.
                dimpoint1(T).
                dimpoint1(T):
                c(H,P1,_).
                not(point(_,P1)).
                point(Last,_).
                frontstr(1,Last,_,Nos).
                str_real(Nos,Nor).
                Nor1 = Nor+1.
                str_real(Nos1,Nor1).
                concat("p",Nos1,NewP1).
                concat("Dim ",NewP1,Sub1).
                concat(Sub1," as esp_Coord",TotP1).
                asserta(point(NewP1,P1)).
                write(TotP1).nl.
                dimpoint1(T):
                c(H,_,P2).
                not(point(_,P2)).
                point(Last,_).
                frontstr(1,Last,_,Nos).
                str_real(Nos,Nor).
                Nor2 = Nor+1.
                str_real(Nos2,Nor2).
                concat("p",Nos2,NewP2).
                concat("Dim ",NewP2,Sub2).
                concat(Sub2," as esp_Coord",TotP2).
                asserta(point(NewP2,P2)).
                write(TotP2).nl.
                dimpoint1(T):
                dimpoint1(T).

(15) predimcurve:-
                freesurf(N,N1,List).
                asserta(pair1(N,N1)).
                dimcurve(List):
                consurf(N,G,N1,List).
                asserta(tri(N,G,N1)).
                dimcurve(List).

(16) dimcurve([]):-
                freesurf(N,N1,List).
                not(pair1(N,N1)).
                asserta(pair1(N,N1)).
                dimcurve(List):
                consurf(N,G,N1,List).
                not(tri(N,G,N1)).
                asserta(tri(N,G,N1)).
                dimcurve(List):
                retractall(last(_)).
                retractall(pair1(_,_)).
                retractall(tri(_,_,_)).
                true.
(17) dimcurve([H|T]):-
                c(H,_,_).
                not(curve(H,_)).
                last(Num).
                New = Num+1.
                retract(last(Num)).
                asserta(last(New)).
                str_int(News,New).
                concat("c",News,Name).
                asserta(curve(H,Name)).
                concat("Dim ",Name,Dim1).
                concat(Dim1," as long",Dim1t).
                write(Dim1t).nl.
                dimcurve(T):
                dimcurve(T).

(18) dimfreesurface:-
                freesurf(N,N1,_).
                not(pair1(N,N1)).
                asserta(pair1(N,N1)).
                last(Num).
                Num1 = Num+1.
                retract(last(Num)).
                asserta(last(Num1)).
                str_int(Nums,Num1).
                concat("fs",Nums,New).
                asserta(fsurf(N,N1,New)).
                concat("Dim ",New,Sub1).
                concat(Sub1," as long",Tot1).
                concat("listcurveID",New,Nid).
                concat("Dim ",Nid,Sub2).
                concat(Sub2," as long",Tot2).
                concat("listcurve",New,Nlc).
                concat("Dim ",Nlc,Sub3).
                concat(Sub3," as long",Tot3).
                write(Tot1).nl.
                write(Tot2).nl.
                write(Tot3).nl.
                dimfreesurface:
                pair1(_,_).
                retractall(pair1(_,_)).
                retractall(last(_)).
                true.nl:
                true.

(19) dimconsurface:-
                consurf(N,G,N1,_).
                not(tri(N,G,N1)).
                asserta(tri(N,G,N1)).
                last(Num).
                Num1 = Num+1.
```

```
                retract(last(Num)).
                asserta(last(Num1)).
                str_int(Nums,Num1).
                concat("cs".Nums.New).
                asserta(csurf(N.G.N1.New)).
                concat("Dim ".New.Sub1).
                concat(Sub1." as long".Tot1).
                concat("listcurveID".New.Nid).
                concat("Dim ".Nid.Sub2).
                concat(Sub2." as long".Tot2).
                concat("listcurve".New.Nlc).
                concat("Dim ".Nlc.Sub3).
                concat(Sub3." as long".Tot3).
                write(Tot1).nl.
                write(Tot2).nl.
                write(Tot3).nl.
                dimconsurface:
                tri(_,_,_).
                retractall(tri(_,_,_)).
                retractall(last(_)).
                true.nl;
                true.

(20) dimmatprop:-
                write("Dim material1 as long").nl.
                write("Dim mat as esp_Matl_Iso").nl.
                write("Dim property1 as long").nl.
                write("Dim property as esp_Property").nl.

(21) dimmesh:-
                write("Dim msize as long").nl.
                write("Dim attrib as long").nl.
                write("Dim size as double").nl.
                write("Dim listsurfacemID as long").nl.
                write("Dim listsurfacem as long").nl.
                write("Dim mesh as long").nl.

(22) dimsolidmesh:-
                solid(_).
                X = dlg_GetStr("Solid Options","Would you like to create a solid
                (y/n)?","n").
                dimsolidmesh1(X);
                X = dlg_GetStr("Shell Options","Please enter the plate
                thickness","0.1").
                asserta(shell(X)).

(23) dimsolidmesh1("y"):-
                asserta(shell("0.1")).
                asserta(makesolid).
                write("Dim property2 as long").nl.
                write("Dim listsurfacesmID as long").nl.
                write("Dim listsurfacesm as long").nl.
                write("Dim solidmesh as long").nl.nl.
(24) dimsolidmesh1("n"):-
                X = dlg_GetStr("Shell Options","Please enter the plate
                thickness","0.1").
                asserta(shell(X)).

(25) predimload(Name.ID):-
                load1(ID.Type.Lx.Ly.Lz.X.Y.Z).
                lastl(N).
                N1 = N + 1.
                retract(load1(ID.Type.Lx.Ly.Lz.X.Y.Z)).
                retract(lastl(N)).
                asserta(lastl(N1)).
                asserta(load2(ID.N1.Type.Lx.Ly.Lz.X.Y.Z)).
                predimload(Name.ID):
                dimload(Name.ID).

(26) dimload(Name.ID):-
                connected(Name.List).
                List = [_|_].
                dimload1(ID.List).!.
                dimloadend;
                dimload2(ID).!.
                dimloadend.

(27) dimload1(ID.[]):-
                dimload2(ID).!.
(28) dimload1(ID.[HIT]):-
                H = e(_,_,_,_,_,_,_,_,extload).
                dimload1a(ID.H).
                dimload1(ID.T):
                dimload1(ID.T).

(29) dimload1a(_,_):-
                dlg_note("This section is not working yet!").

(30) dimload2(ID):-
                load2(ID,_,_,_,_,_,_,_,_).
                dimload2a(ID):
                true.

(31) dimload2a(ID):-
                load2(ID.Num.Type,_,_,_,_,_,_).
                str_int(Nums.Num).
                not(used(Nums)).
                dimload2b(Nums.Type).
                dimload2a(ID):
                retractall(used(_)).

(32) dimload2b(Nums,'p'):-
                concat("listnodelID".Nums.A).
                concat("Dim ".A.A1).
                concat(A1." as long".A2).
                write(A2).nl.
                concat("listnodel".Nums.B).
                concat("Dim ".B.B1).
                concat(B1." as long".B2).
                write(B2).nl.
                concat("load".Nums.C).
                concat("Dim ".C.C1).
                concat(C1." as long".C2).
                write(C2).nl.
                concat("coordl".Nums.E).
                concat("Dim ".E.E1).
                concat(E1." as esp_Coord".E2).
                write(E2).nl.
                asserta(used(Nums)).
                asserta(end).
(33) dimload2b(Nums,'c'):-
                concat("listcurvelID".Nums.A).
                concat("Dim ".A.A1).
                concat(A1." as long".A2).
                write(A2).nl.
                concat("listcurvel".Nums.B).
                concat("Dim ".B.B1).
                concat(B1." as long".B2).
                write(B2).nl.
                concat("loadcurve".Nums.D).
                concat("Dim ".D.D1).
                concat(D1." as long".D2).
                write(D2).nl.
                concat("load".Nums.C).
                concat("Dim ".C.C1).
                concat(C1." as long".C2).
                write(C2).nl.
                concat("coordl".Nums.E).
                concat("Dim ".E.E1).
                concat(E1." as esp_Coord".E2).
                write(E2).nl.
                concat("cidl".Nums.F).
                concat("Dim ".F.F1).
                concat(F1." as long".F2).
                write(F2).nl.
                asserta(used(Nums)).
                asserta(end).
(34) dimload2b(Nums,'s'):-
                concat("listsurfacelID".Nums.A).
                concat("Dim ".A.A1).
                concat(A1." as long".A2).
                write(A2).nl.
                concat("listsurfacel".Nums.B).
                concat("Dim ".B.B1).
                concat(B1." as long".B2).
                write(B2).nl.
                concat("loadsurface".Nums.D).
                concat("Dim ".D.D1).
                concat(D1." as long".D2).
                write(D2).nl.
                concat("load".Nums.C).
                concat("Dim ".C.C1).
                concat(C1." as long".C2).
                write(C2).nl.
                concat("coordl".Nums.E).
                concat("Dim ".E.E1).
                concat(E1." as esp_Coord".E2).
                write(E2).nl.
                concat("sidl".Nums.F).
                concat("Dim ".F.F1).
                concat(F1." as long".F2).
                write(F2).nl.
                asserta(used(Nums)).
                asserta(end).

(35) dimloadend:-
                end.
                retractall(end).
                write("Dim lset as long").nl.
                write("Dim listnodelID as long").nl.
                write("Dim listnode as long").nl.
                write("Dim num as long").nl.
                write("Dim nid as long").nl.
                write("Dim dist as double").nl.
                write("Dim k as long").nl.
                write("Dim pos as long").nl.
                write("Dim dx as double").nl.
                write("Dim dy as double").nl.
                write("Dim dz as double").nl.
                write("Dim d as double").nl.
                write("Dim lmag as esp_Load_Value").nl.
                write("Dim dv as esp_Load_Dir").nl.
                write("Dim coord as esp_Coord").nl.nl:
                true.

(36) predimconstraint(Name.ID):-
                con1(ID.T1.T2.X.Y.Z).
                lastc(N).
                N1 = N + 1.
                retract(con1(ID.T1.T2.X.Y.Z)).
                retract(lastc(N)).
                asserta(lastc(N1)).
```

```
                asserta(con2(ID,N1,T1,T2,X,Y,Z)).
                predimconstraint(Name,ID);
                dimconstraint(Name,ID).

(37) dimconstraint(Name,ID):-
                connected(Name,List).
                List = [_|_],
                dimconstraint1,
                dimconstraint2(ID),
                dimconstraintend;
                dimconstraint2(ID),
                dimconstraintend.

(38) dimconstraint1:-
                consurf(N,G,N1,_),
                not(tri(N,G,N1)),
                asserta(tri(N,G,N1)),
                group(G,Type),
                lastc(Old),
                New = Old + 1,
                retract(lastc(Old)),
                asserta(lastc(New)),
                str_int(Num,New),
                asserta(csurf1(N,G,N1,Num)),
                dimconstraint1a(Num,Type),!,
                dimconstraint1;
                retractall(tri(_,_,_)).

(39) dimconstraint1a(Num,fixed):-
                concat("listsurfacecID",Num,A),
                concat("Dim ",A,A1),
                concat(A1," as long",A2),
                write(A2),nl,
                concat("listsurfacec",Num,B),
                concat("Dim ",B,B1),
                concat(B1," as long",B2),
                write(B2),nl,
                concat("constraint",Num,C),
                concat("Dim ",C,C1),
                concat(C1," as long",C2),
                write(C2),nl,
                asserta(end).
(40) dimconstraint1a(Num,rot(_,_,_,_,_,_)):-
                concat("listsurfacecID",Num,A),
                concat("Dim ",A,A1),
                concat(A1," as long",A2),
                write(A2),nl,
                concat("listsurfacec",Num,B),
                concat("Dim ",B,B1),
                concat(B1," as long",B2),
                write(B2),nl,
                concat("constraint",Num,C),
                concat("Dim ",C,C1),
                concat(C1," as long",C2),
                write(C2),nl,
                concat("bc",Num,D),
                concat("Dim ",D,D1),
                concat(D1," as esp_BC",D2),
                write(D2),nl,
                asserta(end).
(41) dimconstraint1a(Num,tran(_,_,_,_,_,_)):-
                concat("listsurfacecID",Num,A),
                concat("Dim ",A,A1),
                concat(A1," as long",A2),
                write(A2),nl,
                concat("listsurfacec",Num,B),
                concat("Dim ",B,B1),
                concat(B1," as long",B2),
                write(B2),nl,
                concat("constraint",Num,C),
                concat("Dim ",C,C1),
                concat(C1," as long",C2),
                write(C2),nl,
                concat("bc",Num,D),
                concat("Dim ",D,D1),
                concat(D1," as esp_BC",D2),
                write(D2),nl,
                asserta(end).
(42) dimconstraint1a(_,extload).

(43) dimconstraint2(ID):-
                con2(ID,_,_,_,_,_,_).
                dimconstraint2a(ID);
                true.

(44) dimconstraint2a(ID):-
                con2(ID,Num,Type,_,_,_,_),
                str_int(Nums,Num),
                not(used(Nums)),
                dimconstraint2b(Nums,Type),
                dimconstraint2a(ID);
                retractall(used(_)).

(45) dimconstraint2b(Nums,'p'):-
                concat("listpointcID",Nums,A),
                concat("Dim ",A,A1),
                concat(A1," as long",A2),
                write(A2),nl,
                concat("listpointc",Nums,B),
                concat("Dim ",B,B1),
                concat(B1," as long",B2),
                write(B2),nl,
                concat("constraint",Nums,C),
                concat("Dim ",C,C1),
                concat(C1," as long",C2),
                write(C2),nl,
                concat("pc",Nums,D),
                concat("Dim ",D,D1),
                concat(D1," as long",D2),
                write(D2),nl,
                concat("attach",Nums,F),
                concat("Dim ",F,F1),
                concat(F1," as long",F2),
                write(F2),nl,
                concat("pon",Nums,G),
                concat("Dim ",G,G1),
                concat(G1," as long",G2),
                write(G2),nl,
                concat("pidc",Nums,H),
                concat("Dim ",H,H1),
                concat(H1," as long",H2),
                write(H2),nl,
                concat("coordc",Nums,E),
                concat("Dim ",E,E1),
                concat(E1," as esp_Coord",E2),
                write(E2),nl,
                asserta(used(Nums)),
                asserta(end).
(46) dimconstraint2b(Nums,'c'):-
                concat("listcurvecID",Nums,A),
                concat("Dim ",A,A1),
                concat(A1," as long",A2),
                write(A2),nl,
                concat("listcurvec",Nums,B),
                concat("Dim ",B,B1),
                concat(B1," as long",B2),
                write(B2),nl,
                concat("constraint",Nums,C),
                concat("Dim ",C,C1),
                concat(C1," as long",C2),
                write(C2),nl,
                concat("concurve",Nums,D),
                concat("Dim ",D,D1),
                concat(D1," as long",D2),
                write(D2),nl,
                concat("coordc",Nums,E),
                concat("Dim ",E,E1),
                concat(E1," as esp_Coord",E2),
                write(E2),nl,
                concat("cidc",Nums,F),
                concat("Dim ",F,F1),
                concat(F1," as long",F2),
                write(F2),nl,
                asserta(used(Nums)),
                asserta(end).
(47) dimconstraint2b(Nums,'s'):-
                concat("listsurfacecID",Nums,A),
                concat("Dim ",A,A1),
                concat(A1," as long",A2),
                write(A2),nl,
                concat("listsurfacec",Nums,B),
                concat("Dim ",B,B1),
                concat(B1," as long",B2),
                write(B2),nl,
                concat("constraint",Nums,C),
                concat("Dim ",C,C1),
                concat(C1," as long",C2),
                write(C2),nl,
                concat("consurface",Nums,D),
                concat("Dim ",D,D1),
                concat(D1," as long",D2),
                write(D2),nl,
                concat("coordc",Nums,E),
                concat("Dim ",E,E1),
                concat(E1," as esp_Coord",E2),
                write(E2),nl,
                concat("sidc",Nums,F),
                concat("Dim ",F,F1),
                concat(F1," as long",F2),
                write(F2),nl,
                asserta(used(Nums)),
                asserta(end).

(48) dimconstraintend:-
                end,
                retractall(end),
                write("Dim cset as long").nl,nl;
                true.

(49) startpoint:-
                point(Name,p(X,Y,Z)),
                not(used(Name)),
                str_real(X1,X),
                str_real(Y1,Y),
                str_real(Z1,Z),
                concat(Name,".x = ",Sub1),
                concat(Sub1,X1,Tot1),
                concat(Name,".y = ",Sub2),
                concat(Sub2,Y1,Tot2),
                concat(Name,".z = ",Sub3),
                concat(Sub3,Z1,Tot3),
                write(Tot1,": ",Tot2,": ",Tot3),nl.
```

```
        asserta(used(Name)).
        startpoint:
        retractall(used(_)).
        true.

(50) startcurve:-
        curve(H,Name).
        not(used(H)).
        c(H,P1,P2).
        point(Name1,P1).
        point(Name2,P2).
        concat(" = esp_LineEndpoints(color,layer,",Name1,Sub1).
        concat(Sub1,",",Sub2).
        concat(Sub2,Name2,Sub3).
        concat(Sub3,")",Sub4).
        concat(Name,Sub4,Tot).
        write(Tot),nl.
        asserta(used(H)).
        startcurve:
        retractall(used(_)).
        true.

(51) startfreesurface:-
        freesurf(N,N1,List).
        not(pair1(N,N1)).
        asserta(pair1(N,N1)).
        fsurf(N,N1,Name).
        concat("listcurveID",Name,Lid).
        concat(Lid," = esp_ListNextAvailableID",Tot).
        write(Tot).nl.
        concat("listcurve",Name,Lc).
        startfreesurface1(Name,Lc,Lid,List).
        startfreesurface:
        pair1(_,_).
        retractall(pair1(_,_)).
        true,nl:
        true.

(52) startfreesurface1(Name,_,Lid,[]):-
        concat(Name," = esp_SurfBoundary(",S1).
        concat(S1,Lid,S2).
        concat(S2,")",Tot).
        write(Tot),nl.
(53) startfreesurface1(Name,Lc,Lid,[HIT]):-
        curve(H,HNew).
        concat(" = esp_ListAdd(",Lid,Sub1).
        concat(Sub1,",",Sub2).
        concat(Sub2,HNew,Sub3).
        concat(Sub3,")",Sub4).
        concat(Lc,Sub4,Tot).
        write(Tot),nl.
        startfreesurface1(Name,Lc,Lid,T).

(54) startconsurface:-
        consurf(N,G,N1,List).
        not(tri(N,G,N1)).
        asserta(tri(N,G,N1)).
        csurf(N,G,N1,Name).
        concat("listcurveID",Name,Lid).
        concat(Lid," = esp_ListNextAvailableID",Tot).
        write(Tot),nl.
        concat("listcurve",Name,Lc).
        startconsurface1(Name,Lc,Lid,List).
        startconsurface:
        tri(_,_,_).
        retractall(tri(_,_,_)).
        true,nl:
        true.

(55) startconsurface1(Name,_,Lid,[]):-
        concat(Name," = esp_SurfBoundary(",S1).
        concat(S1,Lid,S2).
        concat(S2,")",Tot).
        write(Tot),nl.
(56) startconsurface1(Name,Lc,Lid,[HIT]):-
        curve(H,HNew).
        concat(" = esp_ListAdd(",Lid,Sub1).
        concat(Sub1,",",Sub2).
        concat(Sub2,HNew,Sub3).
        concat(Sub3,")",Sub4).
        concat(Lc,Sub4,Tot).
        write(Tot),nl.
        startconsurface1(Name,Lc,Lid,T).

(57) startmatprop(Name):-
        entity(_,Name,_,E,G,Nu,_).
        shell(X),
        concat("mat.E = ",E,E1).
        concat("mat.G = ",G,G1).
        concat("mat.Nu = ",Nu,Nu1).
        write(E1),nl.
        write(G1),nl.
        write(Nu1),nl.
        write("material1=esp_MatlCreateIsotropic(1,\34mat1\34,color,layer,ma
        t)"),nl,nl.
        concat("property,val1 = ",X,X1).
        write(X1),nl.
        write("property1=esp_PropCreate(1,\34prop1\34,17,1,color,layer,flags,
        property)"),nl.

(58) curvelength:-
        curve(H,_).
        not(used(H)).
        asserta(used(H)).
        c(H,p(X1,Y1,Z1),p(X2,Y2,Z2)).
        L = sqrt((X2-X1)*(X2-X1)+(Y2-Y1)*(Y2-Y1)+(Z2-Z1)*(Z2-Z1)).
        retract(avlength(Old)).
        New = Old + L.
        asserta(avlength(New)).
        retract(last(Num)).
        Next = Num + 1.
        asserta(last(Next)).
        curvelength:
        retract(avlength(Len)).
        last(Num).
        Av = (Len/Num)/2.
        asserta(avlength(Av)).
        retractall(used(_)).
        retractall(last(_)).

(59) startmeshsurf(Name):-
        entity(_,Name,_,_,_,_,Mrs).
        avlength(Len).
        str_real(Mrs,Mr).
        Size = Len/Mr.
        str_real(Sizes,Size).
        concat("msize = esp_MSizeDefault(",Sizes,M1).
        concat(M1,",",M2).
        concat(M2,"1)",M3).
        write(M3).nl.
        write("listsurfacemID = esp_ListNextAvailableID").nl.
        startmeshsurf1,!.

(60) startmeshsurf1:-
        fsurf(_,_,Name1).
        not(used(Name1)).
        asserta(used(Name1)).
        concat("attrib = esp_MAttrSurf(",Name1,M1).
        concat(M1,",",M2).
        concat(M2,"property1)",M3).
        write(M3).nl.
        concat("listsurfacem = esp_ListAdd(listsurfacemID,",Name1,M4).
        concat(M4,")",M5).
        write(M5).nl.
        startmeshsurf1:
        csurf(_,_,_,Name1).
        not(used(Name1)).
        asserta(used(Name1)).
        concat("attrib = esp_MAttrSurf(",Name1,M1).
        concat(M1,",",M2).
        concat(M2,"property1)",M3).
        write(M3).nl.
        concat("listsurfacem = esp_ListAdd(listsurfacemID,",Name1,M4).
        concat(M4,")",M5).
        write(M5).nl.
        startmeshsurf1:
        retractall(used(_)).
        write("mesh = esp_MeshSurface(listsurfacemID,1)").nl.

(61) startsolidmesh:-
        makesolid.
        write("property2=esp_PropCreate(2,\34prop2\34,25,1,color,layer,flags,
        empty)").nl.
        write("listsurfacesmID = esp_ListNextAvailableID").nl.
        startsolidmesh1:
        true.

(62) startsolidmesh1:-
        fsurf(_,_,Name1).
        not(used(Name1)).
        asserta(used(Name1)).
        concat("listsurfacesm = esp_ListAdd(listsurfacesmID,",Name1,M4).
        concat(M4,")",M5).
        write(M5).nl.
        startsolidmesh1:
        csurf(_,_,_,Name1).
        not(used(Name1)).
        asserta(used(Name1)).
        concat("listsurfacesm = esp_ListAdd(listsurfacesmID,",Name1,M4).
        concat(M4,")",M5).
        write(M5).nl.
        startsolidmesh1:
        retractall(used(_)).
        write("solidmesh=esp_MeshSolidFromSurf(listsurfacesmID,property2)
        ").nl,nl.

(63) startload(ID):-
        load2(ID,_,_,_,_,_,_,_,_,_).
        write("lset = esp_LoadCreateSet(1,\34load1\34)").nl.
        write("listnodeID = esp_ListNextAvailableID").nl.
        write("listnode = esp_ListSelectAll(listnodeID,7)").nl.
        write("num = esp_ListNumber(listnodeID)").nl,nl.
        startload1(ID),!:
        true.

(64) startload1(ID):-
        load2(ID,Num,Type,Lx,Ly,Lz,X,Y,Z).
        str_int(Nums,Num).
        not(used(Nums)).
        concat("coord",Nums,C).
        str_real(Xs,X).
        str_real(Ys,Y).
```

```
                str_real(Zs.Z).
                concat(C.".x".C1).
                concat(C1." = ".D1).
                concat(D1.Xs.C2).
                write(C2).nl.
                concat(C.".y".C3).
                concat(C3." = ".D2).
                concat(D2.Ys.C4).
                write(C4).nl.
                concat(C.".z".C5).
                concat(C5." = ".D3).
                concat(D3.Zs.C6).
                write(C6).nl.nl.
                write("nid = 0").nl.
                write("dist = 1000").nl.nl.
                write("For k = 1 To num").nl.
                write("\tpos = esp_CoordOnNode(k.coord)").nl.
                concat("\tdx = ".C1.F1).
                concat(F1."-coord.x".F2).
                write(F2).nl.
                concat("\tdy  = ".C3.G1).
                concat(G1."-coord.y".G2).
                write(G2).nl.
                concat("\tdz = ".C5.H1).
                concat(H1."-coord.z".H2).
                write(H2).nl.
                write("\td = Sqr(dx*dx+dy*dy+dz*dz)").nl.
                write("\tIf d < dist Then").nl.
                write("\t\tdist = d").nl.
                write("\t\tnid = k").nl.
                write("\tEnd If").nl.
                write("Next k").nl.nl.
                startload2(Nums.Type.Lx.Ly.Lz).!.
                startload1(ID):
                retractall(used(_)).

(65) startload2(Nums.'p'.Lx.Ly.Lz):-
                concat("listnodeID".Nums.A).
                concat(A." = esp_ListNextAvailableID".A1).
                write(A1).nl.
                concat("listnodel".Nums.B).
                concat(B." = esp_ListAdd(".B1).
                concat(B1.A.B2).
                concat(B2.".nid)".B3).
                write(B3).nl.
                str_real(Lxs.Lx).
                str_real(Lys.Ly).
                str_real(Lzs.Lz).
                concat("lmag.x = ".Lxs.P1).
                write(P1).nl.
                concat("lmag.y = ".Lys.P2).
                write(P2).nl.
                concat("lmag.z = ".Lzs.P3).
                write(P3).nl.
                concat("load".Nums.Q1).
                concat(Q1." = esp_LoadNodal(1.".Q2).
                concat(Q2.A.Q3).
                concat(Q3.".0.lmag.0.dv)".Q4).
                write(Q4).nl.nl.
                asserta(used(Nums)).
(66) startload2(Nums.'c'.Lx.Ly.Lz):-
                concat("loadcurve".Nums.R).
                concat("cidl".Nums.S).
                concat(R." = esp_ListCurvebyNode(nid.".R1).
                concat(R1.S.R2).
                concat(R2.")".R3).
                write(R3).nl.
                concat("listcurvelID".Nums.A).
                concat(A." = esp_ListNextAvailableID".A1).
                write(A1).nl.
                concat("listcurvel".Nums.B).
                concat(B." = esp_ListAdd(".B1).
                concat(B1.A.B2).
                concat(B2.",".B3).
                concat("cidl".Nums.C).
                concat(B3.C.B4).
                concat(B4.")".B5).
                write(B5).nl.
                str_real(Lxs.Lx).
                str_real(Lys.Ly).
                str_real(Lzs.Lz).
                concat("lmag.x = ".Lxs.P1).
                write(P1).nl.
                concat("lmag.y = ".Lys.P2).
                write(P2).nl.
                concat("lmag.z = ".Lzs.P3).
                write(P3).nl.
                concat("load".Nums.Q1).
                concat(Q1." = esp_LoadCurve(1.".Q2).
                concat(Q2.A.Q3).
                concat(Q3.".0.lmag.0.dv)".Q4).
                write(Q4).nl.nl.
                asserta(used(Nums)).
(67) startload2(Nums.'s'.Lx.Ly.Lz):-
                concat("loadsurface".Nums.R).
                concat("sidl".Nums.S).
                concat(R." = esp_ListSurfbyNode(nid.".R1).
                concat(R1.S.R2).
                concat(R2.")".R3).
                write(R3).nl.
                concat("listsurfacelID".Nums.A).
                concat(A." = esp_ListNextAvailableID".A1).
                write(A1).nl.
                concat("listsurfacel".Nums.B).
                concat(B." = esp_ListAdd(".B1).
                concat(B1.A.B2).
                concat(B2.",".B3).
                concat("sidl".Nums.C).
                concat(B3.C.B4).
                concat(B4.")".B5).
                write(B5).nl.
                str_real(Lxs.Lx).
                str_real(Lys.Ly).
                str_real(Lzs.Lz).
                concat("lmag.x = ".Lxs.P1).
                write(P1).nl.
                concat("lmag.y = ".Lys.P2).
                write(P2).nl.
                concat("lmag.z = ".Lzs.P3).
                write(P3).nl.
                concat("load".Nums.Q1).
                concat(Q1." = esp_LoadSurface(1.".Q2).
                concat(Q2.A.Q3).
                concat(Q3.".0.lmag.0.dv)".Q4).
                write(Q4).nl.nl.
                asserta(used(Nums)).

(68) startconstraint(ID):-
                consurf(_,_,_,_).
                write("cset = esp_BCCreateSet(1,\34Constraint1\34)").nl.nl.
                startconstraint1.!.
                startconstraint2(ID).!;
                con2(ID,_,_,_,_,_,_).
                write("cset = esp_BCCreateSet(1,\34Constraint1\34)").nl.nl.
                startconstraint1.!.
                startconstraint2(ID).!:
                true.

(69) startconstraint1:-
                consurf(N.G.N1,_).
                not(tri(N.G.N1)).
                asserta(tri(N.G.N1)).
                group(G.Type).
                csurf(N.G.N1.Name).
                csurf1(N.G.N1.Num).
                startconstraint1a(Num.Name.Type).
                startconstraint1;
                retractall(tri(_,_,_)).

(70) startconstraint1a(_,_,extload).
(71) startconstraint1a(Num.Name.fixed):-
                concat("listsurfacecID".Num.A1).
                concat(A1." = esp_ListNextAvailableID".A2).
                write(A2).nl.
                concat("listsurfacec".Num.B1).
                concat(B1." = esp_ListAdd(".B2).
                concat(B2.A1.B3).
                concat(B3.",".B4).
                concat(B4.Name.B5).
                concat(B5.")".B6).
                write(B6).nl.
                concat("constraint".Num.C1).
                concat(C1." = esp_BCSurface(".C2).
                concat(C2.A1.C3).
                concat(C3.",1)".C4).
                write(C4).nl.nl.
(72) startconstraint1a(Num.Name.Type):-
                Type = rot(_,_,_,_,_,_).
                concat("listsurfacecID".Num.A1).
                concat(A1." = esp_ListNextAvailableID".A2).
                write(A2).nl.
                concat("listsurfacec".Num.B1).
                concat(B1." = esp_ListFillByEntity(".B2).
                concat(B2.A1.B3).
                concat(B3.",7,".B4).
                concat(B4.Name.B5).
                concat(B5.",5)".B6).
                write(B6).nl.
                concat("bc".Num.C1).
                concat(C1.",tx = 1".C2).
                concat(C1.",tz = 1".C3).
                concat(C1.",ry = 1".C4).
                write(C2).nl.
                write(C3).nl.
                write(C4).nl.
                rc(N.Type).
                str_int(Ns.N).
                concat("constraint".Num.D1).
                concat(D1." = esp_BCNode(".D2).
                concat(D2.A1.D3).
                concat(D3.",".D4).
                concat(D4.Ns.D5).
                concat(D5.",".D6).
                concat(D6.C1.D7).
                concat(D7.")".D8).
                write(D8).nl.nl.
(73) startconstraint1a(Num.Name.Type):-
                Type = tran(_,_,_,_,_,_).
                concat("listsurfacecID".Num.A1).
                concat(A1." = esp_ListNextAvailableID".A2).
                write(A2).nl.
                concat("listsurfacec".Num.B1).
```

```
            concat(B1," = esp_ListFillByEntity(",B2).
            concat(B2,A1,B3).
            concat(B3,",7,",B4).
            concat(B4,Name,B5).
            concat(B5,",5)",B6).
            write(B6).nl.
            concat("bc",Num,C1).
            concat(C1,",tx = 1",C2).
            concat(C1,",ty = 1",C3).
            concat(C1,",rx = 1",C4).
            concat(C1,",ry = 1",C5).
            write(C2).nl.
            write(C3).nl.
            write(C4).nl.
            write(C5).nl.
            rc(N,Type).
            str_int(Ns,N).
            concat("constraint",Num,D1).
            concat(D1," = esp_BCNode(",D2).
            concat(D2,A1,D3).
            concat(D3,",",D4).
            concat(D4,Ns,D5).
            concat(D5,",",D6).
            concat(D6,C1,D7).
            concat(D7,")",D8).
            write(D8).nl.nl.

(74) startconstraint2(ID):-
            con2(ID,Num,T1,T2,X,Y,Z).
            str_int(Nums,Num).
            not(used(Nums)).
            concat("coordc",Nums,C).
            str_real(Xs,X).
            str_real(Ys,Y).
            str_real(Zs,Z).
            concat(C,",x",C1).
            concat(C1," = ",D1).
            concat(D1,Xs,C2).
            write(C2).nl.
            concat(C,",y",C3).
            concat(C3," = ",D2).
            concat(D2,Ys,C4).
            write(C4).nl.
            concat(C,",z",C5).
            concat(C5," = ",D3).
            concat(D3,Zs,C6).
            write(C6).nl.nl.
            write("nid = 0").nl.
            write("dist = 1000").nl.nl.
            write("For k = 1 To num").nl.
            write("\tpos = esp_CoordOnNode(k,coord)").nl.
            concat("\tdx = ",C1,F1).
            concat(F1,"-coord.x",F2).
            write(F2).nl.
            concat("\tdy = ",C3,G1).
            concat(G1,"-coord.y",G2).
            write(G2).nl.
            concat("\tdz = ",C5,H1).
            concat(H1,"-coord.z",H2).
            write(H2).nl.
            write("\td = Sqr(dx*dx+dy*dy+dz*dz)").nl.
            write("\tIf d < dist Then").nl.
            write("\t\tdist = d").nl.
            write("\t\tnid = k").nl.
            write("\tEnd If").nl.
            write("Next k").nl.nl.
            startconstraint2a(Nums,T1,T2,Xs,Ys,Zs).!.
            startconstraint2(ID);
            retractall(used(_)).

(75) startconstraint2a(Num,'p',T2,X,Y,Z):-
            concat("pc",Num,P).
            concat(P," = esp_PointCreate(color,layer,",P1).
            concat(P1,X,P2).
            concat(P2,",",P3).
            concat(P3,Y,P4).
            concat(P4,",",P5).
            concat(P5,Z,P6).
            concat(P6,")",P7).
            write(P7).nl.
            concat("attach",Num,Q).
            concat(Q," = esp_NodeAttach(nid,3,",Q1).
            concat(Q1,P,Q2).
            concat(Q2,")",Q3).
            write(Q3).nl.
            concat("pidc",Num,S).
            concat("pon",Num,R).
            concat(R," = esp_ListPointbyNode(nid,",R1).
            concat(R1,S,R2).
            concat(R2,")",R3).
            write(R3).nl.
            concat("listnodecID",Num,A).
            concat(A," = esp_ListNextAvailableID",A1).
            write(A1).nl.
            concat("listnodec",Num,B).
            concat(B," = esp_ListAdd(",B1).
            concat(B1,A,B2).
            concat(B2,",",B3).
            concat(B3,S,B4).
            concat(B4,")",B5).
            write(B5).nl.
            contype(T2,Type).!.
            concat("constraint",Num,C).
            concat(C," = esp_BCPoint(",C1).
            concat(C1,A,C2).
            concat(C2,",",C3).
            concat(C3,Type,C4).
            concat(C4,")",C5).
            write(C5).nl.nl.
            asserta(used(Num)).
(76) startconstraint2a(Num,'c',T2,_,_,_):-
            concat("concurve",Num,R).
            concat("cidc",Num,S).
            concat(R," = esp_ListCurvebyNode(nid,",R1).
            concat(R1,S,R2).
            concat(R2,")",R3).
            write(R3).nl.
            concat("listcurvecID",Num,A).
            concat(A," = esp_ListNextAvailableID",A1).
            write(A1).nl.
            concat("listcurvec",Num,B).
            concat(B," = esp_ListAdd(",B1).
            concat(B1,A,B2).
            concat(B2,",",B3).
            concat(B3,S,B4).
            concat(B4,")",B5).
            write(B5).nl.
            contype(T2,Type).!.
            concat("constraint",Num,C).
            concat(C," = esp_BCCurve(",C1).
            concat(C1,A,C2).
            concat(C2,",",C3).
            concat(C3,Type,C4).
            concat(C4,")",C5).
            write(C5).nl.nl.
            asserta(used(Num)).
(77) startconstraint2a(Num,'s',T2,_,_,_):-
            concat("consurface",Num,R).
            concat("sidc",Num,S).
            concat(R," = esp_ListSurfbyNode(nid,",R1).
            concat(R1,S,R2).
            concat(R2,")",R3).
            write(R3).nl.
            concat("listsurfacecID",Num,A).
            concat(A," = esp_ListNextAvailableID",A1).
            write(A1).nl.
            concat("listsurfacec",Num,B).
            concat(B," = esp_ListAdd(",B1).
            concat(B1,A,B2).
            concat(B2,",",B3).
            concat(B3,S,B4).
            concat(B4,")",B5).
            write(B5).nl.
            contype(T2,Type).!.
            concat("constraint",Num,C).
            concat(C," = esp_BCSurface(",C1).
            concat(C1,A,C2).
            concat(C2,",",C3).
            concat(C3,Type,C4).
            concat(C4,")",C5).
            write(C5).nl.nl.
            asserta(used(Num)).

(78) contype('f',"1").
(79) contype('t',"2").
(80)entype('r',"3").
```

A2.5 OUTPUT DATA RETRIEVAL

```
(1) v_gauge(Name):-
            exist.!.
            v_next(Name).!.
            deletefile("C:\\Mscn4w46\\solver\\bin\\m1.bas").
            deletefile("C:\\Mscn4w46\\solver\\bin\\m1.dat").
            deletefile("C:\\Mscn4w46\\solver\\bin\\m1.f04").
            deletefile("C:\\Mscn4w46\\solver\\bin\\m1.f06").
            deletefile("C:\\Mscn4w46\\solver\\bin\\m1.log").
            deletefile("C:\\Mscn4w46\\solver\\bin\\m1.op2").

(2) exist:-
            existfile("C:\\Mscn4w46\\solver\\bin\\m1.f06").
            copyfile("C:\\Mscn4w46\\solver\\bin\\m1.f06","C:\\Mscn4w46\\solver\\
            m1.f06").
            file_str("C:\\Mscn4w46\\solver\\m1.f06",File).
            searchstring(File,"1                    * * * END OF JOB * *
            *",_).
            deletefile("C:\\Mscn4w46\\solver\\m1.f06");
            exist.

(3) v_next(Name):-
            disp_gauge(Name).!.
            stress_gauge.!.
```

```
(4) disp_gauge(Name):-
        vg(Name,Data),
        fronttoken(Data,X,Rest),
        checktext1(X,Rest,Xa,Resta),
        fronttoken(Resta,Y,Rest2),
        checktext1(Y,Rest2,Ya,Rest2a),
        fronttoken(Rest2a,Z,Rest3),
        checktext1(Z,Rest3,Za,_),
        str_real(Xa,Xr),
        str_real(Ya,Yr),
        str_real(Za,Zr),
        asserta(vnode(Xr,Yr,Zr)),
        openread(dat,"C:\\Msen4w46\\solver\\bin\\m1.dat"),
        readdevice(dat),
        asserta(last(0)),
        read2,!,
        retractall(last(_)),
        close(file(dat)),
        openread(006,"C:\\Msen4w46\\solver\\bin\\m1.006"),
        readdevice(006),
        read4,!,
        decider,!,
        close(file(006)),
        dlg_note("No Virtual Gauge has been specified!").

(5) read2:-
        not(eof(dat)),
        readln(Line),
        fronttoken(Line,Tok,Rest),
        compare(Tok,Rest),!.

(6) compare(Tok,Rest):-
        Tok <> "GRID",
        read2;
        compare1(Tok,Rest),!.

(7) read3:-
        not(eof(dat)),
        readln(Line),
        fronttoken(Line,Tok,Rest),
        Tok = "GRID",
        compare1(Tok,Rest);
        true.

(8) compare1(Tok,Rest):-
        fronttoken(Rest,ID,Rest1),
        str_int(ID,IDI),
        fronttoken(Rest1,_,Rest2),
        fronttoken(Rest2,X,Rest3),
        checktext1(X,Rest3,Xa,Rest3a),
        fronttoken(Rest3a,Xb,_),
        checktext2(Xb,Rest3a,_,Rest3b),
        fronttoken(Rest3b,Y,Rest4),
        checktext1(Y,Rest4,Ya,Rest4a),
        fronttoken(Rest4a,Yb,_),
        checktext2(Yb,Rest4a,_,Rest4b),
        fronttoken(Rest4b,Z,Rest5),
        checktext1(Z,Rest5,Za,_),
        str_real(Xa,Xar),
        str_real(Ya,Yar),
        str_real(Za,Zar),
        vnode(X1,Y1,Z1),
        Dx = Xar - X1,
        Dy = Yar - Y1,
        Dz = Zar - Z1,
        Dist = sqrt(Dx*Dx + Dy*Dy + Dz*Dz),
        last(Num),
        New = Num + 1,
        retract(last(Num)),
        asserta(last(New)),
        asserta(node(IDI,Dist,Xar,Yar,Zar)),
        compare1a(IDI,Dist,New);
        true.

(9) compare1a(IDI,Dist,New):-
        Dist < 0.0001,
        Dist > -0.0001,
        asserta(match(IDI));
        New <= 4,
        asserta(close(IDI,Dist)),
        read3;
        close(ID1,D1),
        close(ID2,D2),
        ID2 <> ID1,
        D2 >= D1,
        close(ID3,D3),
        ID3 <> ID2, ID3 <> ID1,
        D3 >= D2,
        close(ID4,D4),
        ID4 <> ID3, ID4 <> ID2, ID4 <> ID1,
        D4 >= D3,
        Dist < D4,
        retract(close(ID4,D4)),
        asserta(close(IDI,Dist)),
        read3;
        read3.

(10) checktext1(A,Rem1,B,Rem2):-
        A = "-",
        fronttoken(Rem1,B1,Rem2),
        concat(A,B1,B);
        B = A,
        Rem2 = Rem1.

(11) checktext2(A,Rem1,B,Rem2):-
        A = ".",
        fronttoken(Rem1,B,Rem2);
        B = A,
        Rem2 = Rem1.

(12) read4:-
        not(eof(f06)),
        readln(Line),
        Line <> "                                   DISPLACEMENT VEC
        TOR",
        read4;
        readln(_),
        readln(_).

(13) decider:-
        match(ID),
        readln(Line),
        findnode1(ID,Line);
        close(ID,Dist),
        asserta(used4(ID)),
        asserta(total(0.0)),
        sortnode(ID,Dist),!,
        sortnode1,!,
        retractall(used4(_)),
        close1(NID,Ndist),
        asserta(used4(NID)),
        readln(Line),
        asserta(disp(0.0,0.0,0.0,0.0)),
        findnode2(NID,Ndist,Line).

(14) findnode1(ID,Line):-
        searchstring(Line,"1",Pos),
        Pos = 1,
        findnode1a(ID);
        findnode1b(ID,Line).

(15) findnode1a(ID):-
        readln(_),readln(_),readln(_),readln(_),readln(_),readln(_),readln(Line1),
        findnode1a_next(ID,Line1).

(16) findnode1a_next(ID,Line1):-
        fronttoken(Line1,"POINT",_),
        readln(Line2),
        findnode1(ID,Line2);
        true.

(17) findnode1b(ID,Line1):-
        fronttoken(Line1,Node,_),
        str_int(Node,Nodei),
        ID <> Nodei,
        readln(Line2),
        findnode1(ID,Line2);
        fronttoken(Line1,Node,Rest),
        str_int(Node,Nodei),
        ID = Nodei,
        fronttoken(Rest,_,Rest1),
        fronttoken(Rest1,H,T),
        checktext1(H,T,X,T1),
        fronttoken(T1,H1,T2),
        checktext1(H1,T2,Y,T3),
        fronttoken(T3,H2,T4),
        checktext1(H2,T4,Z,_),
        str_real(X,Xr),
        str_real(Y,Yr),
        str_real(Z,Zr),
        R = sqrt(Xr*Xr + Yr*Yr + Zr*Zr),
        str_real(Rs,R),
        concat("\n'X' Displacement = ",X,X1),
        concat(X1,"\n'Y' Displacement = ",X2),
        concat(X2,Y,Y1),
        concat(Y1,"\n'Z' Displacement = ",Y2),
        concat(Y2,Z,Z1),
        concat(Z1,"\n'R' Displacement = ",Z2),
        concat(Z2,Rs,Z3),
        asserta(vgd(Z3)).

(18) sortnode(ID,Dist):-
        close(ID2,_),
        not(used4(ID2)),
        ID > ID2,
        asserta(used4(ID2)),
        sortnode(ID,Dist);
        close(ID2,Dist2),
        not(used4(ID2)),
        ID < ID2,
        asserta(used4(ID2)),
        sortnode(ID2,Dist2);
        asserta(close1(ID,Dist)),
        total(Old),
        New = Old + (1/Dist),
        retract(total(Old)),
        asserta(total(New)),
        retract(close(ID,Dist)),
        retractall(used4(_)),
        close(ID2,Dist2),
        asserta(used4(ID2)).
```

```
        sortnode(ID2.Dist2);
        true.

(19) sortnode1:-
        total(Tot),
        close1(ID1.D1),
        close1(ID2.D2),
        ID1 <> ID2,
        close1(ID3.D3),
        ID3 <> ID1, ID3 <> ID2,
        close1(ID4.D4),
        ID4 <> ID1, ID4 <> ID2, ID4 <> ID3,
        DN1 = 1/(D1/Tot),
        DN2 = 1/(D2/Tot),
        DN3 = 1/(D3/Tot),
        DN4 = 1/(D4/Tot),
        NewTot = DN1+DN2+DN3+DN4,
        retractall(close1(_,_)),
        retract(total(_)),
        asserta(close1(ID4,DN4)),
        asserta(close1(ID3,DN3)),
        asserta(close1(ID2,DN2)),
        asserta(close1(ID1,DN1)),
        asserta(total(NewTot)).

(20) findnode2(ID,Dist,Line):-
        searchstring(Line,"1",Pos),
        Pos = 1,
        findnode2a(ID,Dist);
        findnode2b(ID,Dist,Line).

(21) findnode2a(ID,Dist):-
        readln(_),readln(_),readln(_),readln(_),readln(_),readln(Line1),
        findnode2a_next(ID,Dist,Line1).

(22) findnode2a_next(ID,Dist,Line1):-
        fronttoken(Line1,"POINT",_),
        readln(Line2),
        findnode2(ID,Dist,Line2);
        true.

(23) findnode2b(ID,Dist,Line1):-
        fronttoken(Line1,Node,_),
        str_int(Node,Node1),
        ID <> Node1,
        readln(Line2),
        findnode2(ID,Dist,Line2);
        fronttoken(Line1,Node,Rest),
        str_int(Node,Node1),
        ID = Node1,
        fronttoken(Rest,_,Rest1),
        fronttoken(Rest1,H,T),
        checktext1(H,T,X,T1),
        fronttoken(T1,H1,T2),
        checktext1(H1,T2,Y,T3),
        fronttoken(T3,H2,T4),
        checktext1(H2,T4,Z,_),
        total(Tot),
        K  = Dist/Tot,
        str_real(X,Xr),
        str_real(Y,Yr),
        str_real(Z,Zr),
        X1 = K * Xr,
        Y1 = K * Yr,
        Z1 = K * Zr,
        disp(Xo,Yo,Zo),
        Xn = Xo + X1,
        Yn = Yo + Y1,
        Zn = Zo + Z1,
        retract(disp(Xo,Yo,Zo)),
        asserta(disp(Xn,Yn,Zn)),
        before_findnode2.

(24) before_findnode2:-
        close1(ID1,Dist1),
        not(used4(ID1)),
        asserta(used4(ID1)),
        readln(Line2),
        findnode2(ID1,Dist1,Line2);
        disp(X,Y,Z),
        str_real(Xs,X),
        str_real(Ys,Y),
        str_real(Zs,Z),
        R = sqrt(X*X + Y*Y + Z*Z),
        str_real(Rs,R),
        concat("\n'X' Displacement = ",Xs,X1),
        concat(X1,"\n'Y' Displacement = ",X2),
        concat(X2,Ys,Y1),
        concat(Y1,"\n'Z' Displacement = ",Y2),
        concat(Y2,Zs,Z1),
        concat(Z1,"\n'R' Displacement = ",Z2),
        concat(Z2,Rs,Z3),
        asserta(vgd(Z3)).

(25) stress_gauge:-
        not(makesolid),
        openread(dat,"C:\\Mscn4w46\\solver\\bin\\m1.dat"),
        readdevice(dat),
        retractall(used4(_)),
        findelement,!,
        closefile(dat),
        decider1,!,
        openread(f06,"C:\\Mscn4w46\\solver\\bin\\m1.f06"),
        readdevice(f06),
        findstress,!,
        closefile(f06),
        print_stress,!;
        makesolid,
        openread(dat,"C:\\Mscn4w46\\solver\\bin\\m1.dat"),
        readdevice(dat),
        retractall(used4(_)),
        findsolelem,!,
        retractall(counter(_)),
        closefile(dat),
        decider1,!,
        openread(f06,"C:\\Mscn4w46\\solver\\bin\\m1.f06"),
        readdevice(f06),
        findsolstress,!,
        closefile(f06),
        print_stress,!.

(26) findelement:-
        readln(Line),
        findelement_next(Line).

(27) findelement_next(Line):-
        fronttoken(Line,"CQUAD4",Rest),
        findelement1(Rest);
        fronttoken(Line,"CTRIA3",Rest),
        findelement1(Rest);
        findelement.

(28) findelement1(Rest):-
        match(IDn),
        asserta(last(0)),
        asserta(counter(0)),
        fronttoken(Rest,IDe,Rest1),
        fronttoken(Rest1,_,Rest2),
        findelement1a(IDn,IDe,Rest2,[]),!;
        retractall(last(_)),
        retractall(counter(_)),
        asserta(counter(0)),
        asserta(last(0)),
        fronttoken(Rest,IDe,Rest1),
        fronttoken(Rest1,_,Rest2),
        close1(N1,_),
        close1(N2,_),
        N2 <> N1,
        findelement1b(N1,N2,IDe,Rest2,[]),!.

(29) findelement1a(IDn,IDe,Rest,List):-
        fronttoken(Rest,N1,R1),
        findelement1a(IDn,IDe,R1,[N1|List]);
        last(Num),
        str_int(IDe,IDei),
        IDei = Num + 1,
        retract(last(Num)),
        asserta(last(IDei)),
        checkelement(IDn,IDei,List),!;
        true.

(30) checkelement(IDn,_,[]):-
        readln(Line),
        fronttoken(Line,_,Rest),
        fronttoken(Rest,IDe1,Rest1),
        fronttoken(Rest1,_,Rest2),
        findelement1a(IDn,IDe1,Rest2,[]).
(31) checkelement(IDn,IDe,[NfT]):-
        str_int(N,Ni),
        IDn = Ni,
        write_element(IDn,IDe),!;
        checkelement(IDn,IDe,T).

(32) write_element(IDn,IDe):-
        asserta(element(IDe)),
        counter(Num),
        New = Num + 1,
        New <= 4,
        retract(counter(Num)),
        asserta(counter(New)),
        readln(Line),
        fronttoken(Line,_,Rest),
        fronttoken(Rest,IDe1,Rest1),
        fronttoken(Rest1,_,Rest2),
        findelement1a(IDn,IDe1,Rest2,[]),!;
        true.

(33) findelement1b(N1,N2,E,Rest,List):-
        fronttoken(Rest,IDn,Rest1),
        findelement1b(N1,N2,E,Rest1,[IDn|List]);
        last(Num),
        str_int(E,Ei),
        Ei = Num + 1,
        retract(last(Num)),
        asserta(last(Ei)),
        checkelement2(N1,N2,E,List,List),!;
        true.

(34) checkelement2(N1,N2,_,[],_):-
        readln(Line),
        fronttoken(Line,_,Rest),
        fronttoken(Rest,E,Rest1),
```

```
                fronttoken(Rest1,_,Rest2).
                findelement1b(N1,N2,E,Rest2,[]):
                true.
(35) checkelement2(N1,N2,E,[N|T],List):-
                str_int(N,Ni).
                N1 = Ni.
                checkelement3(N1,N2,E,List,List).!:
                checkelement2(N1,N2,E,T,List).

(36) checkelement3(N1,N2,_,[],List):-
                readln(Line).
                fronttoken(Line,_,Rest).
                fronttoken(Rest,E,Rest1).
                fronttoken(Rest1,_,Rest2).
                findelement1b(N1,N2,E,Rest2,[]):
                true.
(37) checkelement3(N1,N2,E,[N|T],List):-
                str_int(N,Ni).
                N2 = Ni.
                counter(Num).
                New = Num + 1.
                str_int(E,Ei).
                asserta(tempelem(Ei,List)).
                retract(counter(Num)).
                asserta(counter(New)).
                terminator(N1,N2).!:
                checkelement3(N1,N2,E,T,List).

(38) terminator(N1,N2):-
                counter(Num).
                Num = 2;
                readln(Line).
                fronttoken(Line,_,Rest).
                fronttoken(Rest,E1,Rest1).
                fronttoken(Rest1,_,Rest2).
                findelement1b(N1,N2,E1,Rest2,[]).

(39) decider1:-
                element(E).
                asserta(used4(E)).
                order(E).!.
                retractall(used4(_)):
                next_order.!.

(40) order(E):-
                element(NE).
                not(used4(NE)).
                not(ordered1(NE)).
                E > NE.
                asserta(used4(NE)).
                order(E):
                element(NE).
                not(used4(NE)).
                not(ordered1(NE)).
                E < NE.
                retractall(used4(_)).
                asserta(used4(NE)).
                order(NE):
                asserta(ordered1(E)).
                retract(used4(_)).
                element(NE).
                not(ordered1(NE)).
                asserta(used4(NE)).
                order(NE):
                true.

(41) next_order:-
                tempelem(E1,N1).
                tempelem(E2,_).
                E1 <> E2.
                assertz(ordered2(E1,0.0,0.0,0.0,0.0)).
                order2(E1,N1):
                tempelem(E1,_).
                asserta(element2(E1)).

(42) order2(_,[]):-
                tempelem(E,L).
                not(ordered2(E,_,_,_,_)).
                assertz(ordered2(E,0.0,0.0,0.0,0.0)).
                order2(E,L):
                finddist.
(43) order2(E,[H|T]):-
                str_int(H,Hi).
                node(Hi,_,X,Y,Z).
                ordered2(E,X1,Y1,Z1).
                Xn = X + X1.
                Yn = Y + Y1.
                Zn = Z + Z1.
                retract(ordered2(E,X1,Y1,Z1)).
                asserta(ordered2(E,Xn,Yn,Zn)).
                order2(E,T).

(44) finddist:-
                vnode(X,Y,Z).
                ordered2(E,X1,Y1,Z1).
                not(used4(E)).
                asserta(used4(E)).
                Xn = X - X1.
                Yn = Y - Y1.
                Zn = Z - Z1.
                A = sqrt(Xn*Xn + Yn*Yn + Zn*Zn).
                asserta(temp(E,A)).
                finddist:
                retractall(used4(_)).
                temp(E,Dist).
                asserta(used4(E)).
                findsmall2(E,Dist).!.

(45) findsmall2(E1,D1):-
                temp(E2,D2).
                not(used4(E2)).
                D1 <= D2.
                asserta(used4(E2)).
                findsmall2(E1,D1):
                temp(E2,D2).
                not(used4(E2)).
                D1 > D2.
                retractall(used4(_)).
                asserta(used4(E2)).
                findsmall2(E2,D2);
                retractall(used4(_)).
                asserta(element2(E1)).

(46) findstress:-
                not(eof(f06)).
                readln(Line).
                Line <>"                STRESSES IN QUADRILATER
                AL ELEMENTS (QUAD4)     OPTION = BILIN ".
                findstress:
                retractall(last(_)).
                readln(_).readln(_).readln(_).readln(Line).
                stress_decider(Line).!.

(47) stress_decider(Line):-
                ordered1(E).
                asserta(used4(E)).
                stress1(E,Line).!:
                element2(E).
                stress1(E,Line).!.

(48) stress1(E,Line):-
                fronttoken(Line,_,Rest).
                fronttoken(Rest,E1,_).
                str_int(E1,E1i).
                E <> E1i.
                stress1a(E):
                fronttoken(Line,_,Rest).
                fronttoken(Rest,E1,_).
                str_int(E1,E1i).
                E = E1i.
                pre_stress2(E,Line).!.

(49) stress1a(E):-
                readln(_).readln(_).readln(_).readln(_).readln(_).readln(_).readln(_).readln(_).rea
                dln(_).
                readln(_).readln(_).readln(_).readln(_).readln(_).readln(_).readln(Line).
                stress1b(E,Line).

(50) stress1b(E,Line):-
                searchstring(Line,"0",Pos).
                Pos = 1.
                stress1(E,Line).!:
                searchstring(Line,"1",Pos).
                Pos = 1.
                readln(_).readln(_).readln(_).readln(_).readln(_).readln(_).readln(_).readln(_).rea
                dln(Line1).
                stress1(E,Line1).!.

(51) pre_stress2(E,Line):-
                frontstr(120,Line,_,Ans).
                str_real(Ans,Ansr).
                stress2(E,Ansr).

(52) stress2(E,Ans):-
                not(ordered1(_)).
                asserta(element4(E,Ans)).!:
                ordered1(E1).
                not(used4(E1)).
                asserta(element3(E,Ans)).
                asserta(used4(E1)).
                pre_stress1(E1).!:
                asserta(element3(E,Ans)).
                retractall(used4(_)).
                element3(E1,Stress).
                asserta(used4(E1)).
                get_stress(E1,Stress).!.

(53) pre_stress1(E1):-
                readln(_).readln(_).readln(_).readln(_).readln(_).readln(_).readln(_).readln(_).rea
                dln(_).
                readln(_).readln(_).readln(_).readln(_).readln(_).readln(_).readln(Line).
                fronttoken(Line,_,Rest2).
                fronttoken(Rest2,En,_).
                Last(Num).
                str_int(En,Eni).
                Eni = Num + 1.
                stress1(E1,Rest2).!:
                readln(_).readln(_).readln(_).readln(_).readln(_).readln(_).readln(_).readln(_).rea
                dln(Line).
                fronttoken(Line,_,Rest2).
                fronttoken(Rest2,En,_).
                Last(Num).
```

```
                str_int(En,Eni).
                Eni = Num + 1.
                stress1(E1,Rest2),!.

(54) get_stress(E1,S1):-
                element3(E2,S2).
                not(used4(E2)).
                S1 > S2.
                asserta(used4(E2)).
                get_stress(E1,S1):
                element3(E2,S2).
                not(used4(E2)).
                S1 < S2.
                retractall(used4(_)).
                asserta(used4(E2)).
                get_stress(E2,S2):
                asserta(element4(E1,S1)).

(55) print_stress:-
                vgd(Disp).
                element4(_,S).
                str_real(Ss,S).
                concat("\nVon Mises Stress = ",Ss,A).
                concat(Disp,A,Gauge).
                vnode(Xg,Yg,Zg).
                str_real(Xgs,Xg).
                str_real(Ygs,Yg).
                str_real(Zgs,Zg).
                concat(Xgs," ",A1).
                concat(A1,Ygs,B1).
                concat(B1," ",C1).
                concat(C1,Zgs,Pos).
                concat("Virtual Gauge at Position: ",Pos,Title).
                dlg_note(Title,Gauge).

(56) findsolelem:-
                readln(Line).
                fronttoken(Line,"CTETRA",Rest).
                findsolelem1(Rest):
                findsolelem.

(57) findsolelem1(Rest1):-
                match(N).
                fronttoken(Rest1,E,R).
                fronttoken(R,_,R1).
                write(E),nl.
                str_int(E,Ei).
                NEi = Ei - 1.
                asserta(last(NEi)).
                readln(Line).
                fronttoken(Line,_,R2).
                fronttoken(R2,_,R3).
                fronttoken(R3,_,R4).
                write(R4),nl.
                before_findsolelem1a(N,E,R1,R4,[],0),!:
                fronttoken(Rest1,E,R).
                fronttoken(R,_,R1).
                str_int(E,Ei).
                NEi = Ei - 1.
                asserta(last(NEi)).
                asserta(counter(0)).
                readln(Line).
                fronttoken(Line,_,R2).
                fronttoken(R2,_,R3).
                fronttoken(R3,_,R4).
                close1(N1,_).
                close1(N2,_).
                N2 <> N1.
                before_findsolelem1b(N1,N2,E,R1,R4,[],0),!.

(58) before_findsolelem1a(N,E,R1,R4,List,Num):-
                New = Num + 1.
                New <= 6.
                fronttoken(R1,N1,NR1).
                before_findsolelem1a(N,E,NR1,R4,[N1|List],New);
                findsolelem1a(N,E,R4,List).

(59) findsolelem1a(N,E,R,List):-
                fronttoken(R,N1,R1).
                findsolelem1a(N,E,R1,[N1|List]):
                last(Num).
                str_int(E,Ei).
                Ei = Num + 1.
                retract(last(Num)).
                asserta(last(Ei)).
                checksolelem(N,Ei,List),!:
                true.

(60) checksolelem(N,_,[]):-
                readln(Line1).
                fronttoken(Line1,_,R).
                fronttoken(R,E,R1).
                fronttoken(R1,_,R2).
                readln(Line2).
                fronttoken(Line2,_,R3).
                fronttoken(R3,_,R4).
                fronttoken(R4,_,R5).
                before_findsolelem1a(N,E,R2,R5,[],0).
(61) checksolelem(N,E,[N1|T1]):-
                str_int(N1,N1i).
                N = N1i.
                write_solelem(N,E),!:
                checksolelem(N,E,T1).

(62) write_solelem(N,E):-
                asserta(element(E)).
                readln(Line1).
                fronttoken(Line1,_,R).
                fronttoken(R,E1,R1).
                fronttoken(R1,_,R2).
                readln(Line2).
                fronttoken(Line2,_,R3).
                fronttoken(R3,_,R4).
                fronttoken(R4,_,R5).
                before_findsolelem1a(N,E1,R2,R5,[],0),!:
                true.

(63) before_findsolelem1b(N1,N2,E,R1,R4,List,Num):-
                New = Num + 1.
                New <= 6.
                fronttoken(R1,N,NR1).
                before_findsolelem1b(N1,N2,E,NR1,R4,[N1|List],New);
                findsolelem1b(N1,N2,E,R4,List).

(64) findsolelem1b(N1,N2,E,Rest,List):-
                fronttoken(Rest,N,Rest1).
                findsolelem1b(N1,N2,E,Rest1,[N1|List]):
                last(Num).
                str_int(E,Ei).
                Ei = Num + 1.
                retract(last(Num)).
                asserta(last(Ei)).
                checksolelem2(N1,N2,E,List,List),!:
                true.

(65) checksolelem2(N1,N2,_,[],_):-
                readln(Line1).
                fronttoken(Line1,_,R).
                fronttoken(R,E,R1).
                fronttoken(R1,_,R2).
                readln(Line2).
                fronttoken(Line2,_,R3).
                fronttoken(R3,_,R4).
                fronttoken(R4,_,R5).
                before_findsolelem1b(N1,N2,E,R2,R5,[],0):
                true.
(66) checksolelem2(N1,N2,E,[N1|T],List):-
                str_int(N,Ni).
                N1 = Ni.
                checksolelem3(N1,N2,E,List,List),!:
                checksolelem2(N1,N2,E,T,List).

(67) checksolelem3(N1,N2,_,[],List):-
                readln(Line1).
                fronttoken(Line1,_,R).
                fronttoken(R,E,R1).
                fronttoken(R1,_,R2).
                readln(Line2).
                fronttoken(Line2,_,R3).
                fronttoken(R3,_,R4).
                fronttoken(R4,_,R5).
                before_findsolelem1b(N1,N2,E,R2,R5,[],0):
                true.
(68) checksolelem3(N1,N2,E,[N1|T],List):-
                str_int(N,Ni).
                N2 = Ni.
                counter(Num).
                New = Num + 1.
                str_int(E,Ei).
                asserta(tempelem(Ei,List)).
                retract(counter(Num)).
                asserta(counter(New)).
                solterminator(N1,N2),!:
                checksolelem3(N1,N2,E,T,List).

(69) solterminator(N1,N2):-
                counter(Num).
                Num = 2;
                readln(Line1).
                fronttoken(Line1,_,R).
                fronttoken(R,E,R1).
                fronttoken(R1,_,R2).
                readln(Line2).
                fronttoken(Line2,_,R3).
                fronttoken(R3,_,R4).
                fronttoken(R4,_,R5).
                before_findsolelem1b(N1,N2,E,R2,R5,[],0).

(70) findsolstress:-
                not(eof(f06)).
                readln(Line).
                Line <>"          STRESSES IN TETRAHEDRON
                SOLID ELEMENTS (CTETRA)".
                findsolstress;
                readln(_),readln(_),readln(Line).
                fronttoken(Line,_,Rest).
                solstress_decider(Rest),!.

(71) solstress_decider(Rest):-
                ordered1(E).
                asserta(used4(E)).
                solstress1(E,Rest),!:
```

```
        element2(E).
        solstress1(E,Rest),!.

(72) solstress1(E,Rest1):-
        fronttoken(Rest1,E1,_),
        str_int(E1,E1i),
        E <> E1i,
        solstress1a(E):
        fronttoken(Rest1,E1,_),
        str_int(E1,E1i),
        E = E1i,
        pre_solstress2(E),!.

(73) solstress1a(E):-
        readln(_),readln(_),readln(_),readln(_),readln(_),readln(_),readln(_),rea
        dln(_),
        readln(_),readln(_),readln(_),readln(_),readln(_),readln(_),readln(_),rea
        dln(Line),
        fronttoken(Line,_,R),
        fronttoken(R,_,R1),
        fronttoken(R1,_,R2),
        fronttoken(R2,"GRID",_),
        solstress1(E,R):
        readln(_),readln(_),readln(_),readln(_),readln(_),readln(_),readln(Line),
        fronttoken(Line,_,R),
        fronttoken(R,_,R1),
        fronttoken(R1,_,R2),
        fronttoken(R2,"GRID",_),
        solstress1(E,R):
        true.

(74) pre_solstress2(E):-
        readln(Line),
        searchstring(Line,"CENTER",Pos),
        Pos = 18,
        solstress2(E,Line):
        readln(_),readln(_),readln(_),readln(_),readln(_),readln(_),readln(Line),
        searchstring(Line,"CENTER",Pos),
        Pos = 18,
        solstress2(E,Line).

(75) solstress2(E,Line):-
        not(ordered1(_)),
        frontstr(117,Line,_,Ans),
        str_real(Ans,Ansr),
        asserta(element4(E,Ansr)),!:
        ordered1(E1),
        not(used4(E1)),
        frontstr(117,Line,_,Ans),
        str_real(Ans,Ansr),
        asserta(element3(E,Ansr)),
        asserta(used4(E1)),
        solstress1a(E1),!:
        frontstr(117,Line,_,Ans),
        str_real(Ans,Ansr),
        asserta(element3(E,Ansr)),
        retractall(used4(_)),
        element3(E1,Stress),
        asserta(used4(E1)),
        get_solstress(E1,Stress),!.

(76) get_solstress(E1,S1):-
        element3(E2,S2),
        not(used4(E2)),
        S1 > S2,
        asserta(used4(E2)),
        get_solstress(E1,S1):
        element3(E2,S2),
        not(used4(E2)),
        S1 < S2,
        retractall(used4(_)),
        asserta(used4(E2)),
        get_solstress(E2,S2):
        asserta(element4(E1,S1)).
```