THE INTEGRATION OF CAD AND MOTION ANALYSIS INTO A SINGLE INTUITIVE SOFTWARE PACKAGE

NITIN J DAYA

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

D. JONSON	*Prof. M. WALKER*	06/05/2002.
MSc Eng (Natal)	*MSc Eng (Natal), PhD (Natal)*	
SUPERVISOR	CO-SUPERVISOR	DATE

DURBAN, SOUTH AFRICA

JANUARY 2002

ABSTRACT

Existing CAD systems cannot satisfy all the requirements of 'real' design. Many designers would like to have more powerful and capable CAD systems. 'Real' design is a complex activity involving different types of problems thus a CAD system must be a general purpose system so that it can support all aspects of design including that of motion analysis.

The work here addresses the development of a mechanical design system based on the integration of a CAD system with a motion analysis (MA) code. The basis for this integration involves the use of a common product data model and a mechanism for automating the exchange of information between design and analysis phases using predicate logic.

Various geometric modelling techniques related to both CAD and MA are discussed, as are the aspects that relate to their application in the design environment. The requirements for motion analysis and muitibody systems are also presented in detail.

The use of artificial intelligence and predicate logic to interface the CAD and MA code is discussed. This interface will retrieve the relevant information from a centralised database (the product data model), transfer the data to the motion analysis package, and finally retrieve the necessary results from the analysis package for inspection by the designer. An example is presented to demonstrate the integration.

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.

Nitin J Daya

January 2002

ACKNOWLEDGEMENTS

I would like to extend a special thanks to my supervisors, Mr. David Jonson and Prof. Mark Walker (co-supervisor) for their guidance and encouragement to make this study a success. I would also like to thank Jason de Beer for his valuable assistance with the research.

LIST OF FIGURES

LIST OF TABLES

CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Background

People have always designed things. One of the most basic characteristics of human beings is that they make a wide range of tools and other artefacts to suite their own purposes. As those purposes change, and as people reflect on the currently available artefacts, so refinements are made to the artefacts, and sometimes completely new kinds of artefacts are conceived and made. The world is therefore full of tools, utensils, machines, buildings, furniture, clothes, and many other things that human beings apparently need or want in order to make their lives better.

The purpose of design is to derive from a set of specifications a description of an artefact sufficient for its realisation. Feasible designs not only satisfy the specifications, but take into account other constraints in the design problem arising from the medium in which the design is to be executed (e.g., the strength and properties of materials), the physical environment in which the design is to be operated (e.g., kinematic and static laws of equilibrium) and from such factors as the cost and the capabilities of the manufacturing technology available.

A definition of engineering design, can be described as the organised, thoughtful development and testing of characteristics of new objects that have a particular configuration or perform some desired function(s) that meets our aims without violating any specified limitations.

1.2 The Design Process

The design process is the organisation and management of people and the information they develop in the evolution of a product. The simplest way of explaining the design process is to look at three stages as discussed by Dym and Little [1]. In the first stage, *generation*, the designer proposes various concepts that are generated, by means unspecified. In the second stage, *evaluation*, the design is tested against the design goals, constraints, and criteria that have been set forth by the client, the user(s), and the designer. In the last stage, *communication*, the design is communicated to the manufacturers or fabricators.

The most essential design activity, therefore, is the production of a final description of the artefact. This has to be in a form that is understandable to those who will make the artefact. For this reason, the most widely-used form of communication is the drawing. These drawings will range from rather general descriptions (such as plans, elevations and general arrangement drawings) that give an 'overview' of the artefact, to the most specific (such as sections and details) that give precise instructions on how the artefact is to be made. Drawings are very good at conveying an understanding of what the final artefact has to be like, and that understanding is essential to the person who has to make the artefact.

Nowadays it is not always a person who makes the artefact, machines that have no direct human operator make some artefacts. These machines might be fairly sophisticated robots, or just simpler numerically-controlled tools such as lathes or milling machines. In these cases, therefore, the final specification of a design prior to manufacture might not be in the form of drawings but in the form a string of digits stored on a disk, or in computer software that controls the machine's actions. It is therefore possible to have a design process in which no final communication drawings are made, but the ultimate purpose of the design process remains the communication of proposals for a new artefact.

1.3 Computer-aided Design (CAD)

Geometric modelling on the computer has become the fastest-changing area of engineering design as discussed by Dieter [2], and Ullman [3]. When computer-aided design (CAD) was introduced in the late 1960s, it essentially provided an electronic drafting board for drawing in two dimensions. Through the 1970s CAD systems were improved to provide three-dimensional wire frame and surface models. By the mid – 1980s nearly all CAD products had true solid modelling capabilities which is still evident today in software like Solid Edge and Mechanical Desktop. More recently, tools for kinematic, stress, and thermal analysis have been seamlessly linked to the solid model so that analysis can be done along with modelling. In the beginning CAD required mainframe or midi computers to support the software. Today, with the enhanced capabilities of personal computers, solid modelling can be run on desktop

machines. From its initiation, CAD has promised five important benefits to the engineering design process as stated by Dieter [2].

- Automation: this was done in order to increase the productivity of designers and engineers.
- The ability to use three dimensions in the design process, hence increasing the quality of the design.
- The use of solid modelling to create a geometric database, which can be used further down the design for analysis and simulation, thereby minimising the costly testing of prototypes.
- Electronic transfer of the design to the manufacturing process (CAD/CAM) for rapid prototyping to generate 3D models of parts.
- A paperless design process, where digital databases are used instead of drawings. This process reduces costs and speeds up communications with customers and suppliers.

Most engineering organisations have already benefited from the first two advantages. The transfer of information for analysis and simulation, and computerised manufacture is a reality but has not yet been achieved by many organisations as discussed by Halpern [4]. Solid modelling is nowadays widely used in the CAD process. Solid models have the ability to distinguish between the inside and outside of an object, and are able to support the calculation of mass properties such as weight and moment of inertia.

The great advances in the speed and power of computers, and software, has greatly enhanced the engineer's ability to model designs. In design, progress has been made in one's ability to increase productivity in drafting, constructing solid models, performing

static and dynamic analysis in solids and with finite-element analysis, and even creating virtual prototypes from a CAD file, which can be transferred to the manufacturing process. CAD has changed the way design is carried out and continues to do so with the advancement of computers.

1.4 Virtual Prototyping

Prototyping is an essential but costly step in the engineering design process. Prototypes represent important features of a product that engineers must evaluate and improve. Prototypes help engineers choose design alternatives and to analyse the manufacturing process. In today's engineering environment, prototypes can take a number of forms. In the segment of design engineers, virtual prototyping, (i.e. visualising and testing computer-aided design (CAD) models on a computer before they are physically created) is becoming an increasingly popular way to refine design assumptions and improve new products as shown by Lee [5]. Running a computer model through iterative dynamic simulations before making a physical or rapid prototype accomplishes virtual prototyping.

A physical prototype can require a lot of manual tooling, skilled hand assembly, delicate testing instrumentation, and time spent interpreting prototype data. Engineers and designers must then incorporate what was learned by revising the design, making a new prototype, and repeating the entire process. This process is thus time consuming and generally more than one prototype has to be manufactured. Testing equipment is usually tied up for days or weeks at a time and sometimes may not even be nearly as informative as they need to be. Virtual prototyping, which performs all the above steps

on a personal computer, runs more variations than a rapid prototyping system and permits the design of tests not feasible in the laboratory. With the aid of virtual prototyping, smaller components can be analysed which ordinarily would be difficult to make or instrument (measure). Forces that are difficult or impossible to simulate (such as zero gravity) can be also applied to the design. Modifications and refinement can be easily accomplished with the CAD model more rapidly than with conventional design tests.

Rapid prototyping (RP) systems produce relatively quick, life-size models that engineers can touch, feel, and hold in their hands. The physical models are especially helpful to see how a part will look or how subassemblies may fit together to produce a finished product. However before creating a rapid prototype, engineers should first consider improving a design by analysing the kinematic and dynamic properties of the design, through the use of computer software programs designed to apply the laws of physics to their computer-generated model. Kinematic and dynamic analysis can be thought of as another form of rapid prototyping. It is certainly faster to model a design, construct a simulation, and animate a design, than creating a prototype.

Computers have changed and improved the designer's capabilities in several ways. Organising and the handling of time-consuming operations, means the designer is free to concentrate on more complex design tasks. The designer is able to analyse more complex problems in the design. And through computer-based information systems the designer can share more information sooner down in the design process, like the manufacturing process or tool designers.

The following study describes the integration of a CAD and motion analysis software for the purpose of virtual prototyping. Chapter 2 introduces computer-aided engineering (CAE) and computer-aided design (CAD). A brief look at enhancing CAD systems as well as geometric modelling, solid modelling and hierarchal systems are reviewed. Chapter 3 introduces engineering dynamics. Here mechanism analysis, motion simulation on PC's, multibody systems and virtual prototyping are discussed. Chapter 4 looks at artificial intelligence (AI) in engineering. Logic and object-oriented programming are discussed and how they are used to perform software integration. Various integration methods are also looked into in this chapter.

Chapter 5 discusses how the integration was carried out and the method used. A brief explanation of the program structure developed is also explained. Lastly a generated example explains the application. Chapter 6 then concludes the work.

CHAPTER 2

COMPUTER AIDED ENGINEERING - *MODELLING*

2.1 Introduction

In the 1970's, computer based design tools, which were originally used in the aerospace industry, were adopted by civil and mechanical engineers. A generic set of tools began to emerge at this time. The development of these tools unified various aspects of the efforts in aerospace, civil and mechanical engineering as shown in the work carried out by the University of Southern California [6]. This thus led to the distinct field of Computer-aided Engineering (CAE).

Computer-aided Engineering tools are usually orientated to modelling, simulation, visualisation, optimisation, automated design, documentation, manufacturing and information management as discussed by Lee [5], Ohsuga [7], Thilmany [8], and Bussler [9]. Development in Computer-aided Engineering involves computational mechanics simulation [10], utilising the finite element [11], finite difference and boundary element methods [12]. The use of computational mechanics allowed the simulation of complicated systems, involving solid mechanics (incorporating motion analysis), fluid mechanics, thermal analysis and other areas. The development of these simulation tools was followed by the display of models and results through visualisation methods as discussed by Lee [5] and Thilmany [8].

As the model size and complexity increased, modelling procedures, based on surface and solid representations were also developed. In addition there was also a requirement for meshing procedures to link the geometric and computational mechanics entities. This technology as used by commercial software like MSC Nastran for Windows, ADAMS, and Solid Edge etc., has automated the modelling process. The automation of the design process led to the development of optimisation tools. These tools could be used, for example, to optimise the shape of a solid component subject to stress and buckling constraints used in software such as Pro/Mechanica [13].

Finite element and Motion analysis software have been greatly used for carrying out analysis on virtual prototypes to further automate the design process. Further design process automation has also come through the use of artificial intelligence (AI), as discussed by Holzhauer and Grosse [14] and Ohsuga [7]. Holzhauer and Grosse [14] have used AI techniques to intelligently control finite element thermal analyses. These techniques are used to implement component level decomposition of the physical domain, and to integrate domain information into the analysis process to increase computational efficiency. The use of AI by Ohsuga is discussed later on in the chapter in more detail. Developments in the fields of manufacturing and systems management have also been made to speed up and make Computer-aided Engineering more flexible, '*user friendly*' as seen in Rasdorf [15].

2.2 Computer-aided Design (CAD) Systems

With recent technological advancements, industries are encouraged to design and produce new products of a higher quality. Products are required to have improved

performance, reliability and a lower price while their design and manufacture process is to be quicker and require less manpower. Thus the need for more powerful CAD systems than those currently used is growing. Various studies have been carried out on implementing and improving CAD systems. In Peak *et al* [16] the description of a multi-representation architecture (MRA), a design analysis integration strategy that views computer aided design – computer aided engineering (CAD/CAE) integration as an information intensive mapping between design models and analysis models is investigated. The MRA system involves two distinct phases: tool creation and tool usage. A designer, developer and an analyst are required in the creation phase. The designer then completes the task in the usage phase.

Ohsuge [7] presents an implementation of intelligent CAD systems based on a system called KAUS (Knowledge Acquisition and Utilisation System), which has the capability for processing information that has not been achieved by conventional CAD systems. The conditions that have not been met by conventional CAD systems are mostly implicit but are crucial for 'real world' design. Ohsuge [7] also discusses design analysis in the 'real world' and the features that must be taken into consideration in CAD system design. The following are the most important considerations:

- The human designer should keep control of the design process, but the CAD system must be involved in the process as far (much) as possible.
- The design activity is a process that satisfies given requirements. However requirements are not always fixed and change in the midst of design. A typical CAD system must be able to adapt to such situations.
- Design alternatives may not be restricted in advance without any definite reason.

- The CAD system must be able to deal with complex data and further be able to manipulate the data flexibly to adapt to the dynamic environment of design.
- The CAD system must be a general-purpose system so that it can support all aspects of design.

Thus, to summarise: flexibility, adaptability, expandability, practicability and generality are the conditions that must be met by CAD systems.

2.2.1 CAD system enhancement

There are number of ways to enhance CAD systems such that they can undertake the workload involved in the design process. Ohsuga [7] recognises two possibilities, which are expected to be the most effective to enhance CAD systems:

1. "To computerise as many individual operations involved in a design process." For example, in analysing some aspects, computational methods may have been established and specific programs are available, such as the structural analysis program. For analysing other aspects however, we may need to use databases for model evaluations. Where no analysis method is available, the only option may be to depend on the engineer's knowledge and expertise [7]. Thus the information and extent of knowledge about the object available differs depending on the situation, and also on the showing that computers need the capability to perform as many different operations as possible.

2. "To integrate the different operations involved in the design process into a single process." In order to achieve this goal, systems must be able to transfer information automatically from one operation to another. As each operation in the design process is defined independently the CAD system must be able to keep information on what operation should follow and how to transfer information between different operations. Peak *et al.* [16], Gabbert and Wehner [11], Remondi *et al.* [17], Ohsuga [7] and Hardell [10], have all carried out various studies in the field of integrating CAD and analysis systems. Although the above researchers used different methods, their main goal was to close the gap between CAD and CAE integration thus making the design process a more versatile system.

2.3 Geometric Modelling in CAD systems

Geometric modelling is a basic engineering tool and is the cornerstone of computer-aided design (CAD). Geometric modelling serves as the backbone of engineering design and shadows the design process. Currently geometric modelling is done by either using the geometric constructs (i.e., line, circle, polygon, etc.) supported by the CAD system or by writing a program that generates the geometry of interest automatically as illustrated by Lakmazaheri [18]. While both interactive and automated geometric modelling are more efficient than manual drafting they still have limitations. Interactive geometric modelling is time-consuming. The user has to spend a great deal of time constructing and modifying a drawing. In most cases the user has to develop expertise in the specific commands required in the CAD systems. A limitation of automated geometric modelling is that it requires the development of application

programs (written in programming languages such as C or C++) that can be executed in the CAD environment. The development and maintenance of such programs is expensive and requires programming expertise as discussed by Lakmazaheri [18] and, used by software like Pro/Mechanica.

Various researchers have studied other methods of geometric modelling and shape representation. Damski and Gero [19] discuss a logic-based framework to represent graphical shapes in two dimensions. In the research carried out here, two fundamental concerns when considering a shape are stated: depiction and semantics. Depiction is concerned with the representation of a shape, whilst semantics is concern with it's meaning within a defined context. A development of a logic-based representation for shapes using halfplanes is used as a basis for Damski and Gero's [19] research. Unlike the definition of the halfplane in geometry, where two halfplanes are divided by a line, and the points on that line do not belong to any halfplane, here there is no line. Only a conceptual border divides two sets of points where each set of points define one halfplane. According to a given set of rules, all shapes expressed by halfplanes are then rewritten as a WFF (Well Formed Formula) in the first-order logic. This representation is then used to map shapes into halfplanes, and from them into predicates to be manipulated according to first order logic principals. The halfplane concept has also been used by Giraud [20] but with a different formalisation technique. Here the shape is represented symbolically without any reference to a particular coordinate system or any other numerical reference.

The combination of shape algebras and symbolic logic has been shown to be a powerful tool in the specification of design systems. Chase [21] uses this method by constructing

a model of shape and spatial relations from first principles of geometry, topology and logic. The model developed provides a generalised parameterisation scheme for shape and spatial relations of any dimension and description. The research carried out here concentrates predominately on the knowledge to be modelled and not directly on implementation.

The representations of shapes or geometric models are essentially computational structures that capture the spatial aspects of the object of interest for an application. Modelling is increasingly being used in the design process for various applications. Both real (i.e., physical) and virtual objects are often products of modelling. Virtual objects often correspond to physical objects that are being designed but have not yet been built. Graphic models contain the shape or geometry of the objects, i.e., their geometric models, but they often require additional information such as colour, texture, physical properties (mass, E, G, etc.) and so on. Three-dimensional solid modelling has been able to provide a complete geometric and mathematical description of the part geometry to meet most of the solid modelling requirements.

2.3.1 Solid Modelling

Solid modelling involves the creation and manipulation of complete, unambiguous mathematical representations of 3-D objects. The usual purpose of a solid model is to provide information needed to perform calculations associated with the process of geometric design: visualisation of modelled solids and calculation of shape related properties such as volume, mass, moments of inertia, surface area, and convex hulls.

Solid modelling greatly enhances the engineering design environment by providing a virtual model shop for creating and testing computer-generated prototypes as discussed by Ganter and Storti [22].

Most virtual prototyping and CAD software increasingly make use of solid models. CAD software such as Solid Edge, Pro-Engineer and Solid Works, have all used solid models. These models are then integrated with analysis software to help enable more design alternatives in a shorter period of time. Solid modelling essentially falls into various categories. Research carried out in these categories has been done in order to enhance solid modelling.

Ganter and Storti [22] discuss a renewed method for geometric design by using a method of implicit solid modelling (ISM). The common feature essential to all solid modelling methods is creation of an oriented two-dimensional surface which partitions space into regions occupied by the interior and exterior of the solid. The natural mathematical description of such a spatial partitioning consists of an implicit function $F(x,y,z)$ of the coordinates along with a cut-off value of the function, f_o, associated with the surface. The interior and exterior of the solid are associated with the set of points where $F(x,y,z) < f_o$, and $F(x,y,z) > f_o$, respectively. The term ISM is used to refer to the methods used to define functions which describe "primitive" solids and combine "primitive" defining functions to create the implicit functions which correspond to composite solids. ISM represents a solid as a set of points where an implicit global defining function takes on a value less than a given threshold value. ISM is the natural formulation for mathematical modelling of solids and provides an effective application of solid modelling.

Constructive Solid Geometry (CSG) within an interactive modelling environment provides a simple and intuitive approach to solid modelling. With CSG the solid is constructed in building-block fashion by combining primitive shapes like a cube, cylinder, cone, or sphere. These primitives may be scaled, rotated, and translated in space. They are then combined using the Boolean operations of union, and intersection. The union operation combines the two primitive shapes. The shape resulting from a difference operation resembles the original shape with the area of overlap removed. Wiegand [23] has researched the CSG method and presents an algorithm that directly renders an arbitrary CSG tree that is suitable for use in interactive modelling applications. The purpose of this research by Wiegand [23] is to enhance solid models in an interactive modelling environment.

2.3.2 Hierarchical Systems

Complex design frameworks/models are by their very nature hierarchical in structure. This is due in part to the decomposition at multiple levels of detail of the description of a geometric object. It is both intuitive and advantageous for a CAD environment to have a hierarchical structure to represent objects in the design framework. In CAD models, assemblies consisting of individual parts and subassemblies can be represented by a hierarchical tree structure as shown in Figure 2.1.

Another method of representing assemblies is by representing its individual component functions. The functions describe operations involving change, connect, separate, store,

etc. Implementation of these functions are generally focused on mechanical systems and yields various design models. "Functional decomposition" can be defined as partitioning a given complex functional design requirement into more manageable functions such that it is easier to match design concepts with these functions and arrive at a solution. The functional decomposition of a CAD system can then be represented in a hierarchical structure.

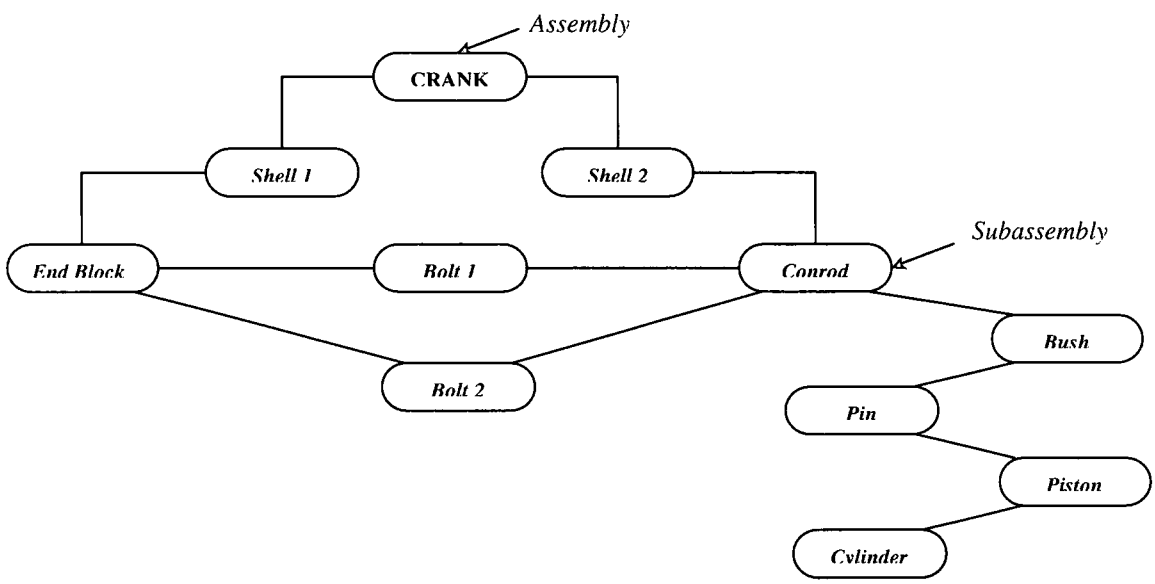


Figure 2.1: Hierarchical structure of Piston-crank Mechanism

Three methods of drawing up a tree structure can be used [24]:

1. *Bottom-up* i.e., from components and subassemblies whose detailed design is already available like that shown in Figure 2.1.
2. *Top-down* i.e., by hierarchical decomposition of functions and sub-functions.
3. By a combination of bottom-up and top-down approaches.

The bottom-up approach is used when the detailed designs of the constituent parts of the assembly are available. Using the bottom-up approach, the solution process is controlled by local interaction of the structural components acting as independent objects in virtual reality, and where the system model is created spontaneously, by interaction of the component models as discussed by Jankovic *et al* [25]. Top-down approach is used for conceptual synthesis of a new system, where the components are unknown at the beginning. The top-down approach requires the knowledge of the entire state-space of the model in order to program the solution which often involves simplification and consequent inaccuracy, whilst its computation intensity does not allow user interaction in the real time as shown by Shah and Tadepalli [24].

Many researchers have used graph structures to represent a models assembly hierarchy. Eastman [26] used graphs in which the components were represented by nodes and transformation matrices attached to arcs. Position changes can then be made from parent to child nodes. Wesley [27] used a graph structure similar to Eastman's wherein the nodes also store positional relationships between objects and material properties. A computer program AUTOPASS was developed using a world (global) model which models the above relationships between parts and sub-assemblies in the data structure [24]. Shah and Tadepalli [24] designed a system that not only captured the assembly hierarchy but also relations between local regions of mating parts (mating features). A hierarchical tree structure is an ideal manner of representing any assembly. A method of applying artificial intelligence (AI) (logic) to assist assembly modelling has also been studied. Lakmazaheri and Edwards [18] use predicate logic and the hierarchical representation to represent components and attributes. The geometric relationships among the attributes are also defined using this system.

2.4 Summary

The advancements made in the field of computer-aided engineering has revolutionised the engineering design process. The great advances in the speed of computers and software has greatly enhanced the engineer's ability to model designs. Geometric modelling on the computer has become the fastest-changing area of engineering design. The use of solid modelling tools has resulted in increased productivity and improved product quality.

Virtual prototyping/engineering is making the traditional design process more intelligent and user friendly. However, today's solid-modelling technology brings us one step closer to realising virtual prototyping (VP), fulfilment of this goal requires more technical leaps like more time saving and automation in the design process.

CHAPTER 3

ENGINEERING MECHANICS - *DYNAMICS*

3.1 Introduction

Engineers are responsible for the design construction and testing of most devices used today. They must have an understanding of physics of these devices and must also be familiar with the use of mathematical models to predict its behaviour. In order to analyse and predict the behaviour of physical systems, the study of mechanics is carried out. " Mechanics may be defined as that science which describes and predicts the conditions of rest or motion of bodies under the action of forces" [28].

Mechanics is divided into three main parts:
- mechanics of rigid bodies,
- mechanics of deformable bodies &
- mechanics of fluids.

The mechanics of rigid bodies is further subdivided in statics and dynamics.
Statics is predominately concerned with the study of objects in equilibrium (rest), and dynamics concerns the study of objects in motion. In civil engineering more attention is paid to statics, while dynamics is a predominant part of mechanical engineering. As

motion analysis is concerned with dynamics, the study of dynamics will be looked into more detail. The study of dynamics is further divided into 2 parts:

1. *Kinematics* – which is the study of the geometry of motion; kinematics is used to relate displacement, velocity, acceleration and time, without reference to the cause of the motion.
2. *Kinetics* – which is the study of the relation existing between the forces acting on a body, the mass of the body, and the motion of the body; kinetics is used to predict the motion caused by given forces or to determine the forces required to produce a given motion.

3.2 Dynamics

As mentioned before dynamics has two main branches, kinematics and kinetics. Kinematics is an important subject and forms the basis of dynamics. To understand kinetics requires an understanding of kinematics, since it would be impossible to relate the forces to the motion if the motion could not be described in the first place as discussed by Pytel and Kiusulaas [29].

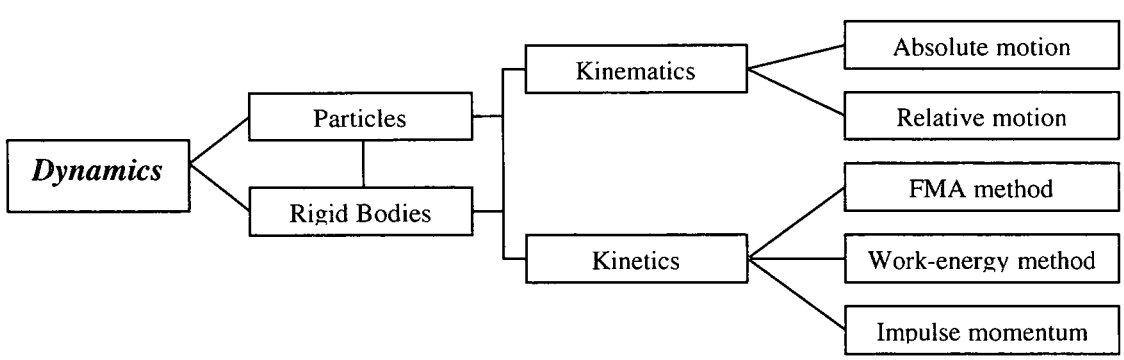


Figure 3.1: Branches of Dynamics

With reference to Figure 3.1, it can be seen that kinematics is subdivided into two categories: absolute motion and relative motion. Absolute motion describes the motion as observed from a fixed reference frame. Relative motion, is motion that is observed from a reference frame that is itself moving [29]. From Figure 3.1 the three methods of kinetic analysis is also shown.

1. The Force-mass-acceleration (FMA) method. This method uses Newton-Euler laws to relate forces that act on a body to its accelerations. The accelerations can then be integrated to determine the velocities and positions of the body as functions of time.
2. The work-energy method, which relates to the work done by the applied forces to the change in kinetic energy of the body.
3. The impulse-momentum method, which relates the impulses of the applied forces to the change in momentum of the body. This method is useful in problems involving colliding bodies.

The above is a basic introduction to engineering mechanics. It explains the different types of mechanics and the categories they fall into to carry out a motion analysis.

3.3 Mechanism Analysis

"Mechanism analysis may be defined as a systematic analysis of a mechanism based on principals of kinematics, or the study of motion of machine components without regard to the forces that cause the motion" [30].

All bodies deform (i.e., change shape) when they are subjected to forces when deformations are so small, that they have no effect on the motion of the body as a whole, the body is the considered to be rigid. A rigid body is therefore, a body for which the deformations can be neglected with no loss of accuracy in the analysis of motion. Most analysis is carried out using rigid bodies. Well-known examples of analysis codes used for simulation are ADAMS and DADS [10].

A mechanism is a combination of rigid bodies (a multibody system) so connected that the motion of one will produce a definite and predictable motion of others, according to a physical law. Alternatively, a mechanism is considered to be a kinematic chain in which one of the rigid bodies is fixed. Mechanism analysis therefore serves as a necessary prerequisite for the proper sizing of machine components, so that they can withstand the loads and stresses to which they will be subjected. "Multibody dynamics can be defined as the discipline describing the dynamic behaviour of mechanical systems consisting of interconnected bodies" [31].

A large majority of mechanisms exhibit motion such that all the links move in parallel planes. This type of motion is called two-dimensional, plane or planer motion. Planar rigid body motion, consists of rotation about axes perpendicular to the plane of motion and translation, where all points in the body move along parallel straight or planar curvilinear paths and all lines embedded in the body remain parallel to their original orientation [32]. Spatial mechanisms allow movement in three dimensions. Combinations of rotation around up to three nonparallel axes and translations in up to three directions are possible depending on the constraints imposed by the joints between links (e.g. spherical, helical, cylindrical, etc.) see [32].

3.3.1 Motion Simulation on PC's

Commercial software programs such as ADAMS® (Automatic Dynamic Analysis of Mechanical Systems), DADS® (Dynamic Analysis and Design Systems), IMP® (Integrated Mechanisms Program), MECHANICA®, and Working Model® perform dynamic simulation of mechanical systems. Many of these general – purpose programs are widely used in industry for virtual prototyping and have user – friendly graphical interfaces that guide the user through various stages of model building [32].

The models (geometric models) used in motion analysis codes have physical properties that include mass, static and kinetic friction, elasticity, moment of inertia and electrostatic charge. Constraints that are available in most motion analysis code include pulleys, joints, dampers, ropes, inflexible rods, springs, actuators and motors to join masses. Uses can also constrain masses with pin, slot, keyed slot, and rigid joints, and specify forces acting on the model. Calculating equations of motion through numerical integration carries out the basic operation of motion analysis. Users define the time steps in the simulation to determine accuracy. Large time steps are acceptable for slowly moving objects, but may throw off accuracy. "Different codes provide several different numerical methods that trade off accuracy versus speed of calculation" [33].

A motion simulation package evaluates moving assemblies by supporting driven kinematic and force-based dynamic simulations, in both 2D and full 3D. engineers can evaluate velocity, part angular acceleration, part mass properties, point position, point velocity, point acceleration, net force, point-to-point separation, separation speed, separation speed change, mechanism redundancies, kinetic energy and point-to-point

true angles by animating actual CAD geometry rather than manually interpreting mathematically relationships. Motion simulation improves product quality and design excellence while reducing development time, prototypes and costs. This is shown by the different commercial packages available like ADAMS, DADS, MSC Nastran 4D and Pro/Mechanica a member of the PTC I-Series. The above packages are ideal for performing virtual simulation, however expertise in the specific package is required which also means that it is time consuming.

Research in integrating design and analysis software has been studied to overcome these disadvantages hence speeding up the design process. Hardell [10] has studied an integrated system for computer-aided design and analysis of multibody systems. Remondini *et al.* [17] also investigates the integration of mechanical analysis within a design process.

3.3.2 Multibody Systems

When carrying out multibody analysis the geometric models of a system is of great importance. Hardell [10] has divided the model of multibody systems into three classes, all of which have importance for the purpose of design.

1. Simple models – These might either represent a simple system or the gross behaviour of a complete system.
2. Models consisting of a small number of interconnected bodies, similar to a linkage system.

3. Models involving more complete assemblies such as engines, complex robots and so on.

Depending on the end requirement any system can be represented as simple, moderate or very complex [10]. Multibody system analysis codes like ADAMS and DADS, are suitable for simulation of the behaviour of rigid body systems of all three classes described above.

The geometric models of a multibody system are mostly carried out using 3D solid models (rigid bodies) with mass and inertia properties. The solid modelling capability makes it possible to define complete and accurate properties of the components of the multibody system. The solid models contain all the rigid body data needed for motion analysis and all the surface data needed for visualisation [10]. Global properties like volume, surface area, moments of inertia and centre of gravity are computed by evaluation of triple integrals such as:

$$\phi_{Solid} = \int_{Solid} f(p)dv \tag{1}$$

where ϕ is the property required, f($\mathbf{p}$) is a vector function describing is a vector function describing ϕ, and integration is performed over the entire volume of the solid [34]. The integral in (1) is used to calculate the properties of the primitives in the Constructive Solid Geometry (CSG as discussed in detail in Chapter 2) scheme as presented in equations (2) and (3), see [34].

$$\phi_{Solid1-Solid2} = \phi_{Solid1} - \phi_{Solid1 \cap Solid2} \tag{2}$$

$$\phi_{Solid1 \cup Solid2} = \phi_{Solid1} + \phi_{Solid2} - \phi_{Solid1 \cap Solid2} \tag{3}$$

The global properties can also be calculated using boundary representations where surface integrals are adopted. Geometry can also be imported from a variety of popular CAD programs using standard formats such as IGES and STEP.

System constraints are other central aspects when modelling multibody systems. The constraints in multibody systems are categorised as follows [32]:

a. Idealised joints that have a physical counterpart. Joint types include revolute (hinge) joint, translation (sliding) joint, cylindrical joint, spherical joint, constant velocity joint, screw joint, planar joint, universal joint, and fixed joint. Complex joints indirectly connect parts by coupling simple joints. They include gears and couplers to model belts and pulleys or chains and sprockets.
b. Joint primitives that place a restriction on relative motion, such as the restriction that one part always moves parallel to another part.
c. Motion generators that drive the model. These supply whatever force is required to make the part satisfy the motion. Motion can be defined as acceleration, displacement, or velocity over time. There are two types of motion: *Joint motion* – Prescribes translation or rotation motion on a translation, revolute, or cylindrical joint. *Point motion* – Prescribes motion between two parts.
d. Contacts that specify how bodies react when they come in contact with another, when the model is in motion. An example could be how a pin moves in a slot.

The forces applied to a multibody system are essential to complete a model for simulation. Forces define loads and compliances on parts. The following are the types of forces used [32]:

- *Applied forces*: They define loads and compliances on parts so that they move in certain ways.
- *Flexible forces*: Flexible connectors resist motion. Examples include beams, bushings, translation spring – dampers, and torsion springs.
- *Special forces*: these include tire and gravity forces.

Once a multibody system (model) is completely defined, a motion analysis software like ADAMS would set the initial conditions and formulate appropriate equations of motion based on the laws of Newtonian mechanics to predict how objects in the model move based on the set of forces and constraints acting on them [32]. Depending on the type and sophistication of the software different types of simulations can be carried out.

3.3.3 Virtual Prototyping

The use of motion analysis has shown that it is not only used to verify the soundness of a initial design, but the feedback received from a good motion analysis simulation gives users important information that can lead to modifying the parameters and improving the design in ways that might otherwise have been overlooked [5]. In achieving this motion analysis can be thought of as a form of prototyping. 'Kinematic analysis allows you to do things you cant do in model testing" [5]. With the use of motion analysis the

engineer can program a simulation to apply forces that will deliberately overload and destroy the prototype. However if a real prototype is used, only one destructive test can be carried out. With motion analysis this can be performed over and over.

Motion analysis software like ADAMS enables engineers to "build" models of not just parts but entire mechanical systems and then simulate their full motion behaviour and optimise their design by producing virtual prototypes. The use of virtual prototypes enables the designer to shorten overall product cycles, reduce the number of hardware prototypes and experiment with more design alternatives [8].

3.4 Summary

Software tools used in the design of components of multibody systems are well developed and widely used [10, 18-12]. This is also the case for the analysis of the behaviour of multibody systems. The use of virtual prototyping in the design process has improved the process by achieving better development times and reducing costs. However the process is still complicated.

Typically, the designer must be experienced in CAD packages, as well as motion analysis package to enable him/her to successfully model and then analyse the proposed design. Integrating the CAD and motion analysis pacakages could significantly optimise the efficiency of this process.

CHAPTER 4

ARTIFICIAL INTELLIGENCE IN ENGINEERING

4.1 Introduction

The use of Artificial Intelligence (AI) in engineering has grown in the past few years. In computer-aided design AI has been used to advance the technology and make CAD systems more intelligent to achieve the users goals. AI technology has made computers more intelligent in solving problems and to cut down the gap between requirements and the computers' capability as show by researchers like Ohsuga [35]. Many people/institutions have introduced AI into CAD system design. With the vast use of AI it is clear that AI offers powerful means to build advanced CAD systems as proven by Gero [36]. From an engineering viewpoint, AI can be defined as generating representations and problems otherwise solved by humans [37]. Related sub fields of AI include: logic, neural networks, object - oriented programming, formal languages, and so on.

The artificial intelligence approach to the development of computer programs is inherently different form the so called traditional approach, when a problem has been identified, the programmer has to design a suitable algorithm for solving the problem, to write the algorithm by means of a suitable programming language and to finally load

the program into the computer for its execution: the computer acts solely as an executor of the program received in input as discussed by Tasso [38]. Using the artificial intelligence approach as shown by Tasso [38], the goal is to provide the computer with problem solving capabilities. The computer in this case will be able to receive the description of a problem at hand.

Knowledge representation is another fundamental corner stone of artificial intelligence: its languages (called knowledge representation languages) are, adequate for coding all the knowledge relevant for a given application. A knowledge representation language is characterised by a syntax, specifying how the symbols of the language can be correctly used and combined together, and a semantics, which describes their meaning in terms of domain knowledge. A knowledge representation language is characterised by a set of reasoning algorithms, i.e. algorithmic procedures, which specify how the specific knowledge representation language can be manipulated in order to automatically perform problem solving, and reasoning activities, which resemble human problem solving behaviour [38]. The most common knowledge languages are: logic formalisms, semantic network frames and production rules.

Among the artificial intelligence techniques and expert systems, knowledge-based systems (KBSs) also play a fundamental role in the field of artificial intelligence. A knowledge-based system (KBS) is a software system capable of supporting the explicit representation of knowledge in some specific competence domain and of exploiting it through appropriate reasoning mechanisms in order to provide high-level problem-solving performance. Therefore a KBS is a specific, dedicated, computer-based problem-solver, able to face complex problems, which, if solved by humans, would

require advanced reasoning capabilities, such as deduction, abduction, hypothetical reasoning, model-based reasoning, analogical reasoning, learning, etc. The special-purpose modules of a KBS as discussed by Tasso [38] include:

- A software interface, which connects the KBS to external software systems.
- An external interface, which connects the KBS to the external environment in which it operates, such as sensors, data acquisition systems, actuators, etc.
- A user interface, which is aimed at making the interaction between the user and the KBS friendly and effective. It may include advanced man-machine interaction facilities, multimedia and dialogue systems, user modelling [39] and smart help systems.
- An explanation system, which is directly connected to the knowledge-based components of the KBS and explains and justifies the behaviour of the KBS to the user, by showing the knowledge utilised, the problem-solving strategies used, and illustrating the main reasoning steps.

More and more, KBSs are not developed in isolation, but they are variously integrated or embedded with traditional software systems. In engineering, these other systems may concern CAD/CAM packages, libraries of specific processing routines, production management and more generally, information systems, and so on. Tools for the development of knowledge-based systems are frequently categorised into three types as stated by Blount and Clark [40]:

1. languages (such as LISP and PROLOG);
2. expert systems 'toolkits' or 'environments';

3. expert system 'shells'.

Languages are the most flexible method programming. Shells are a cost-effective and quick method of constructing a system. Toolkits have flexibility required and have features available to enable rapid construction of systems, but they require relatively expensive hardware and software. Tasso and Oliveira [41], discuss the development of knowledge-based systems for engineering. Here a better understanding to the design and development process and the specific techniques utilised for constructing expert systems in engineering is discussed.

Artificial Neural Networks are another AI application, which has recently been used widely to model some of the human activities in many areas of science and engineering. Neural Networks are AI tools that are able to learn and generalise from examples and experience and to produce meaningful solutions to problems, even when input data contain errors or are incomplete. Rafiq *et al* [42] has given some guidelines for designing neural networks to solve engineering problems.

4.2 Logic in AI

"Logic can be used to express statements, to validly infer additional statements and to rigorously prove (or disprove) statements" [37].

The use of logic as a specification and programming tool has become widespread over the past two decades, initially with the Prolog language and continuing with constraint logic programming languages as shown by Benhamou [43]. Among the advantages

over more traditional procedural programming methods is the ability to describe the knowledge to be encapsulated in a model without the need to prescribe data manipulation procedures [44]. Mathematical logic deals with the language for defining mathematical objects and the laws for reasoning about them. Mathematical logic provides a rigorous symbolic language for representing and a mechanical method for manipulating (reasoning about) knowledge. Several types of logic exist, examples of which are propositional logic, predicate logic, probabilistic logic, and fuzzy logic [18]. The use of predicate logic has been extensively used for geometric modelling in the CAD environment. Numerous papers may be found in the literature, dealing with the use of logic in modelling [18-21, 45-47]. These are discussed in more detail further on in the chapter.

4.2.1 Predicate Logic

Predicate logic, is a well-understood and widely used logic. It consists of a formal language for representing problems and an inference strategy for solving them. Inference is the process of deducing (new) facts from (other) existing facts. The language of predicate logic embodies a vocabulary and a set of syntax rules. The vocabulary of predicate logic consists of predicates (for representing objects) and a set of logical operators (for representing relations). The syntax rules guide the formulation of valid logical expressions. The inference strategy of predicate logic is based on theorem proving where pattern matching and substitution are used for drawing logical inferences [18].

There are two features of predicate logic that make it suitable for engineering problem solving automation as discussed by Lakmazaheri [45].

1. The declarative nature.
2. The relational nature of the language.

The declarative nature of predicate logic facilities problem modelling by reducing the modelling process to the task of describing the problem using declarative statements without much concern about how these statements are to be processed. The processing of logical expressions is handled by the problem independent inference strategy. The relational nature of predicate logic promotes non - directional computation. That is, the set of logical expressions representing a problem can be used to determine the values of the remaining attributes. The relational nature of predicate logic greatly simplifies the modelling process by permitting multiple uses of a formulation.

The use of predicate logic and its extensions (e.g. constraint logic) for automated engineering problem solving has been a research focus in recent years [7, 48-51]. Lakmazaheri [45] illustrates the use of this extended logic for 2D geometric modelling and presents an implementation of the logic using a commercial CAD environment. A logical expression usually consists of a set of predicates connected by logical operators.

"A predicate is a parameterised proposition, i.e., a proposition with variables" [37]. A predicate can be viewed as a symbolic representation of a relation, it has a name and several attributes. The predicate name part of a fact normally appears first. The things related to one another by a predicate are called its arguments. For example, *rectangle*

(30,40) is a predicate where *'rectangle'* is the predicate name and *'30'* and *'40'* are its arguments. The primary logical operators used in predicate logic are implication, conjunction, disjunction and negotiation. The implication operator is denoted by the symbol *':-'* and symbol *','* is used for denoting conjunction as used by Prolog. A rule has two parts, a head and a body. The head is the predicate that appears to the left of the implication operator, the body is the set of predicates that appear to the right of this operator (*head:-body*). The following is an example of how the rules are used in predicate logic. This example shows a rule that can be used to conclude whether a menu item is suitable for John.

John is a vegetarian and eats only what his doctor tells him to eat.

Given a menu and the preceding rule, you can conclude if John can order a particular item on the menu. To do this, you must check to see if the item on the menu matches the constraints given.

a. Is Food_on_menu a vegetable?

b. Is Food_on_menu on the doctor's list?

c. Conclusion: If both answers are yes, John can order Food_on_menu.

In Prolog (predicate logic), a relationship like this must be represented by a rule because the conclusion is based on facts. Here's one way of writing the rule:

john_can_eat(Food_on_menu):- vegetable(Food_on_menu),

on_doctor_list(Food_on_menu).

Notice here the comma after *vegetable(Food_on_menu)*. The comma introduces a conjunction of several goals, and is simply read as "and"; both

vegetable(Food_on_menu) and *on_doctor_list(Food_on_menu)* must be true, for *john_can_eat(Food_on_menu)* to be true.

Similarly, Lakmazaheri [45] describes a geometry declaratively by: (1) identifying the components and underlying relations, and (2) expressing them as logical expressions. Using deductive rules, one can define a complex geometry in terms of primitive geometric shapes. Primitive shapes with no deductive definition are represented using logical expressions called facts. Deductive reasoning, is a means of determining the truth of a conclusion from a set of hypotheses [45].

Logical Inference

First, the conclusion of interest, called the goal statement, must be represented as a logical expression. For example:

link(X1,Y,_,_,_,_), link(X2,Y,_,_,_,_),

L=X1 - X2, L>0.

The above expression states that there are two links, one located at (Z1,Y) and the other located at (X2,Y), such that L = X1 - X2 > 0. The objective is to determine whether or not the above statement is true. Each predicate in the goal is replaced with the body of the rule. The head predicate matches the goal predicate. The goal is expanded, (its predicates are replaced with the rule's body predicate), in this fashion until no further expansion is possible. This process can be illustrated graphically using a tree structure

consisting of root nodes and child nodes. The tree structure is similar to a hierarchical system discussed in Chapter 2. The solution to the goal can be obtained by processing the tree bottom-up, from the child nodes to the root node. The processing begins by finding all the nodes that match each of the child nodes of the tree and passing these facts to the parent node for further processing.

The processing of the parent node entails joining the set of values obtained from the child node(s) and eliminating the values that do not satisfy the constraints attached to the node. The processing of a goal tree in this fashion yields a possibly empty set, therefore having a false conclusion. Otherwise the conclusion is proven true. This kind of processing as implemented by Lakmazaheri [45] is used in most logic programming environments such as Prolog. Languages like Prolog do not generate the whole goal tree, rather if utilized what is known as depth-first strategy. Using this strategy a partial goal tree is generated by expanding the tree along the other branches. If it becomes obvious that a child node does not lead to a solution, then the system backtracks to the parent of that node and the goal is expanded along another branch. This process is repeated until as solution is found or all the branches are exhausted.

As stated before, predicate logic is a means of determining the truth, not enforcing logical statements; deciding whether a statement is true and determining the conditions that make the statement true. By enforcing the truth of a goal statement, a geometric mechanism can be created deductively [45].

4.2.2 Logic - Based Geometric Modelling

Most engineering systems can be conceptually decomposed into a hierarchy of components, which can be represented as a tree structure, as discussed in Chapter 2. In this structure, a parent node represents a component and a child node represents a subcomponent. Firstly, a partial rule is formed by writing each parent predicate as the rule's head and child predicated (in conjunctive form) as the rule's body. Secondly, the geometric relations that hold true among the attributes of the predicates in the rule are identified. These relations are added to the body of the rule as constraints. This method of representing a system has proven successful and is used by researchers like Lakmazaheri and Edwards [18], Lakmazaheri [45] and Wu *et al* [46].

Geometric relations can also be defined using linguistic phrases. Keywords like supports are on, and adjacent to indicate the existence of certain geometric relations among two or more objects. Linguistic phrases defining geometric relations are replaced with constraints in the translation process. Therefore, the constraint definition of such linguistic phrases must be pre-defined and specified as a part of the geometry definition. Lakmazaheri and Edwards [18] explains how to translate the linguistic definition of geometry into a logic program by using a series of sentence transformations. The sentences describing geometry can be separated into five sets [18]:

1. Sentences that define the components' hierarchy.
2. Sentences that define the components' attributes directly.
3. Sentences that define the components' attributes indirectly.
4. Sentences representing relations as constraints and linguistic phrases, and
5. Sentences that define linguistic phrases in terms of constraints.

Sentences in the first, second and third sets go through a series of transformations via a set of re-write rules.

The translation of the linguistic definition of geometry into a logic program is carried out in eight steps, see Lakmazaheri and Edwards [18].

Step 1: Using the rules the first set of transforms can be re-written into a set of sentences of form a:- b, where a is a component name and b is the name of a subcomponent. Once the duplicate sentences are removed the sentences are combined to appear in one sentence in the form 'component:-subcomponent$_1$, -subcomponent$_2$ ..., subcomponent$_n$."

Step 2: In this step the second set of transforms are rewritten into a set of sentences each defining an attribute of a component ordered lexicographically.

Step 3: Here the transforms of the third set of elements are rewritten into sentences of form $P_1 \rightarrow P_2$, where P_1 and P_2 are two component names.

Step 4: Here modifications are made to second set using the third set. For example $P_1(X)$ and $P_2(Y)$ be two elements in the second set, where X and Y are lists of attributes . If $P_1 \rightarrow P_2$ then $P_2(Y)$ changes to $P_2(Y4X)$.

Step 5: This step involves combining the first and second sets. Let $P_1(a_1,a_2,...a_n)$ be an element in the second set. Then every occurrence of P_1 in the first set is replaced with $P_1(a_1,a_2,...a_n)$.

Step 6: The fourth set contains three types of elements: relations in the form of simple constraints, relations in the form of compounds constraints, and relations as linguistic phrases. In this step, first every linguistic phrase denoting a relation is replaced with its constraint definition given in the fifth set. Then every compound constraint is replaced with a set of simple constraints.

Step 7: The component names appearing in each constraint in the fourth set are identified and the constraint is appended to every expression in the first set that contains the same names.

Step 8: By step eight each sentence in the first set has the following format:

$$\underbrace{P_1(a_1,a_2,...):-P_2(b_1,b_2,...}_{predicates},\underbrace{a_1 ofP_1 = b_1 ofP_2 + b_2 ofP_2,...}_{constraints}$$

Where P_1, P_2, ... are component names; and a_1, a_2, ..., b_1, b_2, ... are attribute names. As shown in the preceding sentence, each sentence consists of a predicate part and a constraint part. In this step, first the constraint part is processed. This processing involves replacing each component attribute with a unique variable. Hence, the previous expression becomes

$$P_1(a_1,a_2,...):-P_2(b_1,b_2,...),...,X_1 = X_2 + X_3,...$$

where X_1 is the variable representing a_1 of P_1, X_2 is the variable representing b_1 of P_1, and X_1 is the variable representing b_2 of P_2. Redundant constraints (if any) are then removed from the expression, resulting in a reduced set of constraints.

Using the above rules a complex geometry can be defined in terms of simple components in turn can be defined in terms of simple geometric shapes. Lakmazaheri [45] also uses this approach and explains how to draw up the structure with a set of rules. An example of a retaining wall is used to further explain more detail. In addition to representing the system, Lakmazaheri [45] also shows how text objects can be easily defined and incorporated into the logic-based definition of geometry. The dimension lines and the associated text for a complex geometry can be defined hierarchically and represented as a set of rules. Most CAD systems allow the creation & manipulation of

geometric objects. Geometric operation (i.e. scaling, translation, rotation, etc.) can be defined deductively using predicate logic, see Lakmazaheri [45].

Damski & Gero [19] have also used first - order logic to represent shapes. In this work a logic - based representation for shapes using half planes as its basis is developed. The method used was to map shapes into half planes and into predicates, manipulated according to first - order logic principles. Of particular importance in the research was the representation, the consistency with which shapes can be represented, can be carried out by either straight line segments or curved line segments. This is an unusual characteristic to be able to achieve in a shape representation. The work carried out here can be used as the basis for an implementation to support early stages of design.

Logic has also been implemented in constraint modelling to effectively describe and manage assembly related geometry constraints for mechanical systems. In Wu *et al* [46] a method to model design constraints using first order predicate logic is developed. When applying first order logic to formally describe a relationship such as constraints, axioms are first developed. An axiom is a valid form. Instances of these axioms are used to describe detailed constraint relationships that are observed in an entity, such as a mechanical system. This logic based constraint formulation provides a natural way to describe geometry constraints and builds a basis for developing engineering design change management capabilities for concurrent engineering environments.

The logic definition of geometry can be used for several purposes as discussed by Lakmazaheri [45]. It can be used to:

1. Construct the entire (or parts of) geometry.
2. Modify the geometry.
3. Query the constructed geometry (to extract explicitly or implicitly defined data).
4. Reason about the geometry (to deduce implicitly defined relations).

The declarative nature of predicate logic facilitates linguistic geometric modelling. For a certain class of problems it is possible to define a vocabulary and a grammar for describing geometric components and components hierarchy in a language that is natural to the user. The statements formulated in this fashion can be transformed into a set of rules and facts, which can then be processed via deductive reasoning.

4.3 Object - Oriented Approach

Object - oriented approaches have recently emerged as a unifying concept in development and integrating programs from various application domains [47]. The approach has already gained popularity in a number of fields in computer science, including programming languages, database applications, data modelling, artificial intelligence, knowledge representation, office automation and CAD/CAM. Applications of object - oriented approaches have taken several routes, but the most pronounced feature of all object - oriented approaches is the representation of physical and conceptual entities of the real world as encapsulated objects. The closer the objects are to the entities in the real world, the easier it is to understand and manipulate the objects and accurately model their physical or conceptual states and behaviour [47].

In an object - oriented system, all entities (physical or conceptual) are represented by objects. Each object is an instance of a class. A class describes a set of physical or conceptual objects that have similar properties, constraints and operations. Each class defines attributes, or a set of data, that define the state of the object and a set of methods (procedures) that describes the objects behaviour. Each object has specific values assigned to the attributes defined for its class. Objects that belong to the same class have the same methods, and they behave similarly. Each object encapsulates (hides) its data attributes from other objects, and only shows its behaviour [47].

Using this object - oriented approach, integrated computer-aided design (CAD) systems have been developed. Abdalla *et al* [47] has developed software using an object - oriented approach to integrate a finite elements and graphics application program. Here a data-translation facility (DTF) is developed using the C++ language, for translating data among finite element and graphics-based programs using IGES (Initial Graphic Exchange Standard) as the standard exchange format. In an object - oriented approach to software design, a program is viewed as a system of objects (data and operation), where the dynamics of the real world are modelled as objects that interact with each other via messages. The design of classes of objects and how objects interact with each other are the most important and essential parts of an object - oriented design. The following is an outline of the object - oriented design methodology used by Abdalla *et al* [47], in their DTF.

1. Identification of classes and class hierarchy.
2. Identification of communication channels and protocols.
3. Declaration of methods for classes.
4. Declaration of attributes for classes.

5. Use of aggregation for building more complex objects.

This methodology is general and applicable to diverse software design and implementation problems in engineering.

Mukunda *et al* [52], have developed a parallel finite element framework to facilitate rapid prototyping using the object - oriented techniques. Parallel computing and object - oriented technologies are integrated to achieve efficiency in both computation and software development. The framework developed here provides simple and consistent interfaces, which look similar to those of sequential finite element analysis framework. With the aid of object - oriented concepts, the extension from the sequential finite element architecture framework to a parallel finite element framework, was accomplished easily and elegantly proving further that the use of the object - oriented approach to FEM and design is a natural step.

The object - oriented technique has also been used to guide the design of information management systems that support the layout design process as studied by Vallis and Colton [53].

4.4 Integrated Systems

Interests in developing integrated design systems have grown and a great effort has also been devoted to integrating existing systems. Several approaches to integration have been developed over the years, and each approach has several advantages and

disadvantages. These approaches include the following as discussed by Abdalla and Yoon [47]:

1. *Direct translation approach*: Here each application programs' data are translated from its format to the specific format required by another program.
2. *Neutral file approach*: All application programs read from and write in a file that is in a standard neutral format.
3. *Central database approach (CDB)*: In this case the data common among all applications are kept in a central repository. Application programs then access the CBD through the database management system (DBMS).
4. *Distributed database approach*: Here data is kept in local application databases. Information common among two or more applications is shared through a knowledge-based system.

These and other approaches have been used by various researchers to make CAD systems more powerful and effective for users. Hardell [10], uses one of the above approaches to develop an integrated system for computer aided design and analysis software of multibody systems. The integration of software can be of two kinds:

- Open integration which usually means communication between different software using files, and
- Closed integration, which means that the transfer of data is performed without user interaction and control.

Hardell [10] has developed an integrated system using the open system with a product database. The integrated system is centred around a product database, the structure of which was developed using a commercial database management system (DBMS), see [54] Bertino *et al.* The structured query language (SQL) of the DBMS allows advanced filtering of the data relevant for an application before the data is shipped to the application program. There were two interface codes developed by Hardell [10] for the interaction of his system.

Firstly, a program was produced that reads the database dump created by the CAD program and writes multibody system data to the product database. This program uses the specialised subroutines delivered together with the CAD software to read the database dump, and also embedded SQL (ESQL) to manipulate the product database. Secondly, a program is developed that reads multibody system data from the product database, manipulates the data and writes the analysis code input. Here the physical description of the product database is transferred into mathematical expressions and subroutine calls.

Peak *et al* [16] have also studied integrating engineering design and analysis. The integration here is carried out as an information-intensive mapping between design models and analysis models using a multi-representation architecture (MRA) strategy. The integration strategy presented by Peak *et al* [16] has the following characteristics. The MRA:

- Addresses the information intensive nature of CAD-CAE integration.
- Breaks the design-analysis integration gap into smaller sub-problems.

- Flexibly supports different design and analysis methods and tools.
- Is based on modular, re-usable information building blocks.
- Defines a methodology for creating specialised, highly automated analysis tools to support produced design.

Four representations:
1. Solution Method Model (SMM)
2. Analysis Building Block (ABB)
3. Product Model (PM)
4. Product Model-based Analysis Model (PBAM)

compose the MRA and together make it a flexible, extendible architecture.

Both Hardell and Peak *et al* saw the use of a product data model. Gabbert and Wehner [11] also make use of the product data model as a pool for CAD-FEA data. Product data models are semantic models of a product. They can be found in the shape of internal data models, found in CAE applications. They cover the product geometry as well as functional, technological and other features. The product data model used by Gabbert and Wehner [11] can be referred to as a centralised database. Design, manufacturing, management and numerical analysis are just some of the data that can be integrated into the product data model. Wu *et al* [46] have also made use of the product data model. In their work, development is made so that the data model includes constraint relationships. Here the geometry constraints are stored in the global database at various objective levels. These constraints can be easily translated into a predicate form.

4.5 Summary

Artificial Intelligence (AI) methodologies will eventually become a natural and integral component of the set of computer based tools of engineers. These tools will then significantly elevate the role of computers in engineering from the present day emphasis on calculation to the much broader area of reasoning.

The main gains from integrated design and analysis systems, are the possibilities of using calculated component data like mass and moment of inertia from CAD in the simulation models, the automatic formulation of input files for the analysis cool and finally the visualisation of simulation results using the surface data of the solid models. Virtual prototyping relies on the complete integration of design and analysis phases so that the design configuration may be immediately evaluated. Virtual prototyping is becoming more apparent in today's engineering design process, assisting the designer to improve the quality of a design.

CHAPTER 5

APPLICATION STUDIES – SOFTWARE INTEGRATION

5.1. Introduction

The following chapter describes the method used to integrate CAD and motion analysis software. The basis for this integration involves the use of a common product data model and a mechanism for automating the exchange of information between design and analysis phases using predicate logic. Numerous design and analysis packages are available. A study has been undertaken to find a compatible pair of design and analysis packages.

Emphasis was placed on the program structure developed to automatically construct a motion analysis model (multibody system) and to analyse this system obtaining the necessary results for use in finite element analysis and other design requirements.

The main objectives of this program was to achieve the following:

- Establish a communication link with analysis software.
- Communicate with a product data model (database), i.e. to add or remove information from the database.
- To automatically model (geometrically) bodies of a multibody system.

- To automatically generate commands / script for an analysis code.

The aim of this approach involves the generation of specialised interfaces between the CAD and analysis software to allow them to operate in an automatic or integrated mode. The integrated mode offers the advantage of allowing the user to work in a single design environment whilst automating the translation process. This approach has the potential of increasing productivity and reducing the chance of error during the translation process, as shown in the work carried out by Jonson *et al* [55].

5.2. Software tools

5.2.1 Software tools for the design of multibody systems

There are currently many CAD software products available as. These are capable of representing the geometrical characteristics of a particular product's design. The software packages incorporate various features depending on the level of sophistication demanded by the designer. Originally, most CAD packages were primarily used as a substitute for the drafting board. Presently more CAD packages are advanced modular systems that are capable of facilitating a variety of operations ranging from design development via solid modelling through to finite element mesh generation. When considering a model for analysis the model should include the following functionalities as discussed by Hardell [10]:

1. Solid modelling capability – The components of the mechanical system should be defined completely and as accurately as possible. Solid models are capable

of conveying information related to visual appearance as well as to the rigid body data required for performing motion analysis.

2. Mechanism design capabilities – The design environment should include features such as joints, springs, dampers, as well as forces and displacements applied to the components of the mechanical system.
3. A mechanism for extracting the required analysis specific data from the product data model and channelling it to the analysis code it in an acceptable format.

In terms of meeting the requirements for the design generation software in the development of the integrated system a commercial product was not used. A design generation tool, developed at Technikon Natal was used instead. This tool is based on the Open Inventor set of graphics libraries.

Open Inventor is a library of objects and methods used to create interactive 3D graphics applications. Open Inventor is a set of building blocks that enables one to write programs that take advantage of powerful graphics hardware features with minimal programming effort. The software is also equipped with full-featured solid modelling capabilities allowing realistic visual representation of a mechanical system as well as the calculation of all quantities required for the execution for the motion analysis.

The various components that form part of a multibody system generated in Open Inventor, is converted from the geometrical models into a data file. This information is contained in a text format in the form of 'facesets'. A faceset is a means of graphical representation composed of individual faces as described by Werneke [56]. A faceset can be referred to as triangular shaped surfaces. The required number of facesets is

assembled to make up a complete component. All straight edge shapes can be easily and exactly represented using facesets. Shapes that contain curved edges can only be approximated due to each face of a faceset being a straight edge. However, increasing the number of faces used to represent a curved edge can enhance the accuracy. Figure 5.1 shows a sphere represented by using facesets.

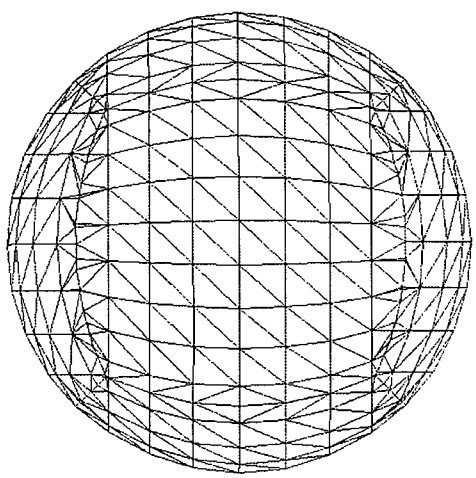
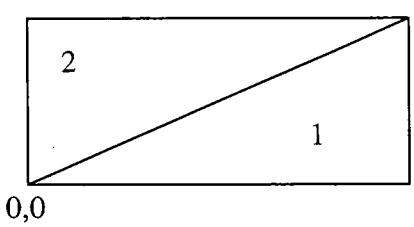


Figure 5.1. Sphere represented using facesets

Each individual faceset is represented by its x, y, and z coordinates. For example using two facesets as shown in Figure 5.2, can represent a rectangle with height=1, and length=2. Each faceset has its three points (nodes) that make up the surface. This is represented as follows: $(x_1,y_1,z_1,x_2,y_2,z_2,x_3,y_3,z_3)$ where x,y,z represents the coordinates of each of the nodes of the faceset.



Faceset 1: (0,0,0,2,1,2,0,0,0)

Faceset 2: (0,0,0,2,1,0,0,1,0)

Figure 5.2. Rectangle with facesets

Using this format the geometry is stored into the data file as mentioned previously. Although the geometric and feature data are contained within a local database accessible only to the modelling engine (Open Inventor), all data is channelled to a centralised database for storage and later retrieval. The mechanism used to extract motion analysis specific data and format it for use by the motion analysis software is based on the use of predicate logic as mentioned before, and is described further on in this chapter.

5.2.2 Software tools for the Motion Analysis of Multibody Systems

The calculation of the motion as well as the internal and external forces associated with interconnected components that constitute a multibody system forms the basic purpose of the integrated system developed here. Using numerical methods such as finite element or boundary element analysis, the complete static and dynamic behaviour of individual components subject to known loads may be accurately determined. Alternatively, in the case of a complete system, the inertial effects of a multibody system may be more important. In situations like this, the method utilised in determining the behaviour of the entire multibody system involves analysing the system as an ordered structure of interconnected rigid body components. Consequently, the output (resultant forces) from the motion analysis can be used as input (in the form of loads and constraints) into the finite element analysis.

As in the case of CAD software, there are various software tools available for simulating the response of multibody mechanical systems can be divided into two broad groups as discussed by Jonson *et al* [57].

1. Multibody system analysis codes that describe system response in terms of numerical data. These codes are capable of analysing large complex problems.
2. Multibody system analysis codes that describe system response in terms of explicit symbolic system equations. The complexity of the system equations increase with the size of the mechanical system and for this reason these codes are limited to medium sized problems.

In comparison to the CAD software, here a commercially available code was used. Initially, Working Model 2D was used to analyse the mechanical motion of a multibody system. Working Model is a software that operates with a numerical code and therefore falls into the first group mentioned above. The code is 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 user interaction. A simple example of a 2D piston-crank assembly was written out in WM Basic within Working Model, see Appendix A. WM Basic (short for Working Model Basic) is an object-oriented programming language based on Visual Basic. Using WM Basic, Working Model will automatically create a model that satisfies give parameters, thus the user does not need to be familiar with Work Model. The script generated in Appendix A was done manually to test the application. However, this process was to be automated based on predicate logic. Due to the restriction of two dimensions in Working Model, alternative software incorporating three dimensions was also considered.

ADAMS, is a full three-dimensional simulation package, and is equipped with powerful modelling and simulation capabilities. It enables users to produce virtual

prototypes realistically simulating the full-motion behaviour of complex mechanical systems on their computers and quickly analysing multiple degign variations until an optimal design is achieved. When compared to Working Model ADAMS has more advanced capabilities for multibody's simulation. ADAMS is also equipped with a command/macro feature, which allows motion analysis specific information to be inputted directly into the software in a specific format. ADAMS / View commands consist of one or more parameters, with one or more values assigned to each parameter. The smallest meaningful command is one keyword. Keyword indicate what action the command is to perform "FILE ADAMS_DATA_SET.WRITE", for instance. Parameters hold the data for the command. Parameters have both a name and a value separated by an "=" sign, "FRAME_NUMBER=4" and "COLOR=RED" for instance. A macro is a single command that you create to execute a series of ADAMS/View commands. The following are some of the tasks that can be performed using these commands.

- Automate repetitive procedures.
- Build general-purpose extensions to model in ADAMS.
- Automatically create an entire model.
- Quickly build many variations of a mechanism.

ADAMS, met the requirements set out in the CAD software, i.e. solid modelling, mechanism design capabilities and obtaining results from a completed analysis. The commands are essentially the data channel through which analysis specific product data is supplied from the product database to ADAMS. As mentioned previously this mechanism is based on the use of predicate logic.

5.2.3 The product data model

As mentioned before the basis for this integration involves the use of a common product data model. The product data model can also be referred to as a centralised product database. The product data model should be specified in such a way that it is capable of supporting all the data transactions that is required in the development of the design and analysis models. In a general sense, product data models can be viewed as semantic models and are therefore capable of encapsulating information related to aspects of product design and performance as well as to aspects of implementation such as manufacturing as shown by Jonson *et al* [55]. In essence in order to support the integration of the various software tools used in product development the product data model contained in the product database should store product geometry as well as functional, technological and other features of the product design that may impact on design development.

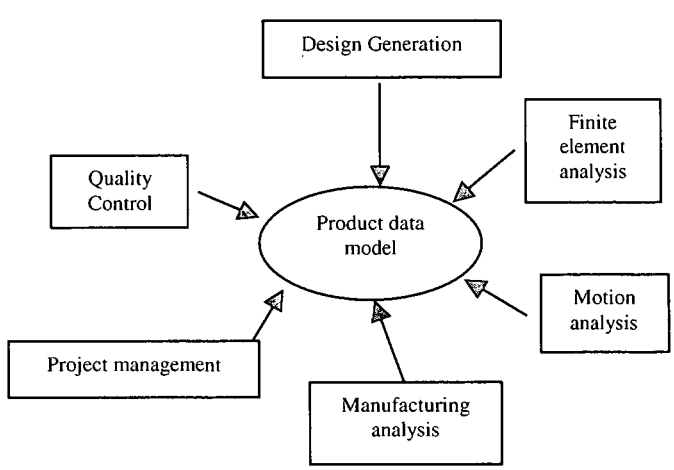


Figure 5.3. A generalised representation of the product data model

Figure 5.3 shows a general representation of various data that may be integrated within the product data model. The data model used for the integration is similar to that suggested by Wu *et al* [46] for the design and simulation of mechanical systems. A simulation based mechanical system design contains objects including mechanical systems, simulations, bodies, connectors, assemblies, parts, feature and connectivity's between bodies, between assemblies, and between parts. Using these objects Wu *et al* organised them in five object levels: environment, mechanical system assembly part, and feature. A mechanical system is composed of interconnected assemblies, including bodies and connectors. The interconnection relationship among other information is all stored in the data model.

To implement the use of the product data model, it is stored in an object-oriented format within a relational database management system. To achieve this the commercial available package Microsoft Access was the chosen DBMS. Table 5.1 illustrates the database generated in Microsoft Access, used to store the appropriate information.

ID	Name	Shape	Joint Type	Material Name	E (GPa)	G (GPa)	nu	Density	Location	Load Type	Load	Orientation	Connection 1	Connection 2
2	crank	faceset		steel	207E+9	80E+9	0.3	7850	0, 0, 0			0, 0, 0		
5	joint1	joint	revolute						80,80, 0			0, 0, 0	ground	crank
9	load1	load							80, 80, 0	rotational	10E9	0, 0, 0	crank	ground

Table 5.1. Database

From Table 5.1, the following fields of information are represented.

1. ID – An identification number given to each entity of the multibody system.
2. Name – Defines the name of the entity. The name is used for connectivity between components and to identify the various components.
3. Shape – This defines the components geometrical or constraint property.
4. Joint Type – The type of joint, revolute, translational, linear, etc.
5. Material Name – Identifies the type/name of material for a particular component.
6. E (GPa) – Represents the elastic modulus of the material.
7. G (GPa) – Represents the shear modulus of the material.
8. nu – Identifies the possions ratio.
9. Density – Represents the density of the material used in kg/m^3.
10. Location – Position of each entity with relation to the origin (0,0,0) of the model.
11. Load Type – the type of load, rotational, translational, linear, etc.
12. Load – Represents the magnitude of the load.
13. Orientation – This defines the orientation of a entity with respect to how it is originally created as a default.
14. Connection 1 – Connection of a constraint to a given entity. First connection point.
15. Connection 2 – Connection of a constraint to a given entity. Second connection point.

5.3 Interface program structure

As mentioned previously predicate logic is used as the basis for this interface. Considering that predicate logic has a declarative nature and the relational nature of the language, a suitable programming language was chosen. PROLOG (PROgramming in LOGic), a declarative language met all requirements for the application. Here a commercial package Visual Prolog, which is a programming environment for the PROLOG programming language, is used. Prolog is what is known as a declarative language. This means that given the necessary facts and rules, Prolog will use deductive reasoning to solve programming problems.

Visual Prolog provides several predicates that allow the user to access a PCs' operating system. This access to the operating system is achieved by using the system predicate. An external program can be executed with a call to the system. The call sends a command string to the operating system for execution. This command was used to link up with ADAMS.

system ("\"\34D:\\Program Files\\ADAMS 11.0\\common\\mdi.bat\34 aview ru-s i \"");

This is shown in Appendix B, page 109 under the section Start ADAMS. The open link between ADAMS and Prolog is used to transfer information to and from ADAMS.

5.3.1 Database communication link

The Visual Prolog external database system is not meant to be readily available from non-Prolog applications. For that purpose, third party SQL (Structured Query

Language) Databases is used as a standard, Database-Management System (DBMS). The chosen database Microsoft Access meant that the link was possible. SQL involves terms like: tables, rows, columns and indexes.

The interaction between the Client (Microsoft Access database) and the Server applications (in Visual Prolog) is possible after a connection has been established. A connection is identified by a Connection Handle (DBC Handle), which is returned by the predicate *sql Connect.* At the end of the interaction the connection must be closed by using the predicate *sql Disconnect.* Appendix B, page 108 shows the connection made to the database (Access Data) under the section of Database connection. The interaction between the client and the server is further divided into a number of statements. A statement is identified by a Statement Handle (STMT HANDLE) and contains a single row of information from the database. The rows of information are further subdivided into the individual fields contained in the database. The information in the database is stored as string variables. When numerical values are required for specific calculations the strings are converted to real variables using Prolog predicates.

Once the connection is made with the external database, information is then stored in a temporary internal database. Visual Prolog's internal database is composed of facts that can be added directly into and removed from the program at run time. This internal database is temporarily contained within the Prolog program and has no connection with the Access database. When the program is executed all information stored in the internal database is deleted to prevent duplication.

The objects contained in the Access database all defined by shape, constraint type or applied load. This object is stored as a fact containing material properties, location orientation and connectivity of the components in the initial database. The fact is defined by the functor entity, with its arguments shown below:

entity(id,name,shape,name,e,g,u,d,location)

Where:	id	-	is the ID number of the row of information
	name	-	name of component
	shape	-	type of shape of the component (represented as facesets, this will be explained later in the chapter)
	name	-	name of material (type)
	e	-	modulus of elasticity
	g	-	shear modulus
	u	-	poissons ratio of the material
	d	-	density of material
	location	-	position of each component from its drawing origin 0,0,0

Similar facts are defined for the constraints and loads where the material name is replaced by jtype or ltype to represent the type of joint or load respectively. The facts used are *entity 2 & 3* as shown in Appendix B, page 90 (database entity).

Information regarding the geometrical description is also required to be represented in the form of facts (shapes as above). This was done using the text file generated by Open Inventor as discussed before.

Using the text file generated and the standard predicate consult, the information is converted into a set of facts, defined by the functor s, and each of its points.

$$S(x_1,y_1,z_1, x_2,y_2,z_2, x_3,y_3,z_3)$$

Where $x_?$; $y_?$; $z_?$ represents the "x" "y" "z" coordinates of the faceset.

This implies that all information from the Microsoft Access database has been retrieved and converted into facts to create a model in the ADAMS environment.

5.3.2 Geometric Modelling representation

In order to carry out an analysis, a geometric model is required. Thus the faceset information is to be represented in a geometric model for this purpose. The faceset information, as described previously, is modified into two facts to allow for the representation to be more convenient, so that it can be converted easily for generating a complete model. The facts are defined by the functors S1and S2. See Appendix B, page 94→95. The section *changedata, changedata1 and changedata2* shows the modification of the fact $S(x_1,y_1,z_1, x_2,y_2,z_2, x_3,y_3,z_3)$ to :

$$S1(name,p(x_1,y_1,z_1),p(x_2,y_2,z_2),p(x_3,y_3,z_3))$$

$$S2(name,C_1,C_2,C_3)$$

The first fact defines the faceset in terms of its name and coordinates / node. The name allows the program to differentiate between the different facesets. Each node (p) is represented by its coordinates. The second fact also defines the faceset by its name and a list of three curves joining the three nodes to make up the faceset. No two curves or points may be coincident. In order to recognise which curves have already been created, the following logic is used when formulating the facts defined. To construct the first faceset the fact is recognised by name. Then curves are constructed from point 1 to point 2, point 2 to point 3 and lastly point 1 to point 3. This fact is then saved in the internal database. For the rest of the facesets, a new name is defined each time. If a curve exists from the 1st point to the 2nd point then the previously defined curve is used in the new fact. If a curve does not exist then a curve is constructed form the 1st to the 2nd point. The same is done for the last two curves. This is then saved in the internal database. The construction of the curves is shown in Appendix B, page 95→96 under the *section curvedata1, 2, and 3.*

All the facesets that are on common planes needed to be grouped together to form a boundary surface composed of a number of facesets, which in turn would represent the final geometry. To develop an equation describing the plane of each faceset, it was necessary to determine the equation of a vector normal to that plane. This can be achieved by the locations of three points lying on the plane. This information is available in the form of facts describing the location of the three nodes of each faceset. In figure 5.4, p_1, p_2, and p_3 represent the three nodes of a given faceset on plane xy. S and R are vectors constructed between points p_1 and p_2, and p_1 and p_3 respectively. Q is a vector perpendicular to both R and S and normal to the plane of the faceset (p_1, p_2 and p_3 points).

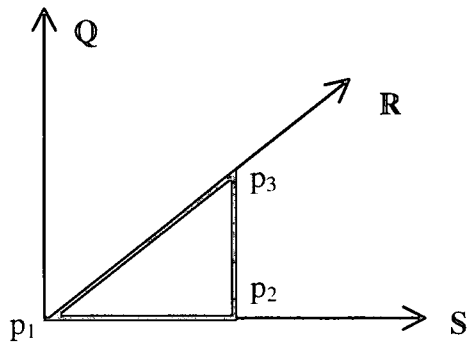


Figure 5.4. Vectors of a faceset.

Using the "cross product' method [58], the equation of the vector Q can be calculated from equations of R and S. If S and R are defined by the following vector equations:

$$S = s_1 i + s_2 j + s_3 k$$

$$R = r_1 i + r_2 j + r_3 k$$

Then the cross product of S and R can be calculated as follows:

$$R \times S = (s_1 r_3 - s_3 r_2)i - (s_1 r_3 - s_3 r_1)j + (s_1 r_2 - s_2 r_1)k$$

$$= A_i - Bj + Ck$$

Using the equation passing through (x_0, y_0, z_0) is given by:

$$A(x - x_0) + B(y - y_0) + C(z - z_0) = 0$$

Using this method, an equation describing the plane of each faceset is developed and stored as a fact in the following format:

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

Where:	Name	-	name of faceset.
	A,B,C	-	coefficient of the x, y, and z plane equations.
	D	-	summation of the coefficients by the coordinates of the origin point.

This information of each plane allows for the orientation of each faceset to be directly compared and analysed.

Once the plane equations have been defined, all facesets on common planes need to be identified and grouped. The Visual Prolog program compares the coefficient (including variable D) of the plane facts to determine which facesets lie on the same plane. Facesets are assumed to be co-planar if the following conditions are met: the planar equations are identical, the planar equations are exact opposites, and one equation is an exact multiple of the other. The logic structure used to identify and group common planes is shown in Appendix B, page 96→97, under the section *makeplane* and *makeplane2*. This can be described using the appropriate statements and rules.

The first plane fact from the internal database is selected and a new fact called "facelist" containing a list is made. A marker on the plane fact is defined to prevent it from being used again. For the other plane facts, the values from the database is retrieved and

defined by another name. The values from the first plane are compared to the new one in question. A new name is added to the facelist. Once all plane facts have been used then the command ends.

Once the facesets on common planes have been identified and stored into the database the fact is defined by the functor facelist.

Facelist(Name,List)

Where:	Name	-	is the name of the facelist with all the faceset on a specific plane.
	List	-	contains the names of the facesets.

Using the lists in the *facelist* facts, it is then determined if all the facesets in each list are adjacent to each other. The method used to achieve this is briefly described. A duplication is made from each fact in the *facelist* and stored into a temporary fact (*templist*). A new fact is then created, curvelist1, using Name and an empty list. The first faceset from the *templist* is then subtracted and placed in the empty list. Another fact, *lastlist*, defined by an empty list as its only argument is created. The *lastlist* allows the last set of adjacent facesets added to curvelist1 to be checked in term for the facesets adjacent to them. The program then determines which facesets are adjacent to the one in the empty list and is added to the list. Similarly the process is repeated until all the *templist* facts contain empty lists. The *lastlist* and empty list then combined and process is repeated until all the *templist* facts contain empty lists. The internal database now is expanded with a fact containing a list of all adjacent facesets belonging to a common

plane. The code used to determine the facesets adjacent to each other, can be seen in Appendix B, page 100, under the section *checkcurve1, 2 & 3.*

In order to complete the representation of the geometry using the facesets, a boundary curve from the set of adjacent facesets was to be extracted. The curve that makes up the boundary, will be selected if a particular curve is not shared between two facesets on the same plane. The logic used here, selects a curve form a faceset, then searches for the same curves continuation in another faceset. From curvelist1 each faceset is selected until the list is empty. The process is carried out on all three curves that make up the faceset. As shown in Appendix B, page 101, the boundary curves are inserted, into the internal database stored in the format of:

Curvelist(Name,List)

Where:	Name	-	list of boundary curves on a plane
	List	-	list of boundary curves

Using this a complete list of curves is available in order to represent a specific geometry in a wireframe model. The format of the curves can represent a model in both two and three dimensions depending on how the model is constructed in the Open Inventor graphics library (i.e. in 2-D or 3-D). Using the commands in ADAMS, the geometry can be represented to simulate virtual models.

5.3.3 ADAMS commands

In order to communicate with the ADAMS software, a list of commands is written to an ADAMS command file (*.cmd extension), with the necessary commands needed to generate, assemble and analyse a multibody system. In order to communicate with a file the *openwrite* predicate is used. If a file already exists, it is then erased. Otherwise, Visual Prolog creates a new file and makes an entry in the appropriate directory. A write device is then used, (i.e. the Symbolic File Name) in this case a view for ADAMS/view. With having the file open information in order to communicate with ADAMS is then inserted.

A typical set of commands as seen in Appendix C, page 113, would set up basic requirements to create and analyse a multibody system. Default units (length = mm, mass = kg, force = Newton, time = sec and angle = deg) are set up initially. A coordinate system is then chosen for the model (eg Cartesian orientation type). The required materials for the different components are then created. The properties for the material, which are stored in the temporary internal database as explained before, are restructured and used as required. As seen in Appendix B, page 93, under the section *nextstep1*, the *retractall* runtime fact is used. *Retractall* retracts all matching facts for a facts section predicate in the facts section that matches the given facts. The list of materials is queried by the material name so as not to duplicate any materials. As seen in Appendix C, page 113, the material properties include the young's modulus, poissons ratio and density as required by the ADAMS software. The next step was to create a default part ground. This can be seen as a global part to which all other components can be assembled. It can also be referred to al a drawing platform. Once all the above is

achieved the necessary components, joints, loads etc can be inserted. At this stage the file is closed.

The next step first carries out the conversion of the facesets to the required geometry as explained earlier in this chapter. Using this faceset technique meant that the geometry was to be created using line geometry. The original aim of this project was to construct three-dimensional wire – frame models using ADAMS. To construct these models the coordinates of the vertices were first specified. Thereafter an attempt was made to join these vertices with a single continuous polyline. It was necessary to use a single polyline to ensure that ADAMS would recognise it as a single model. However in constructing a three dimensional wire-frame with a single polyline, it is seldom possible to complete the frame without overlapping segments. Thus it is not possible to construct three-dimensional wire-frames using polylines. It is for this reason that the project was restricted to two-dimensional models.

Due to the restriction of two dimensions a flat 2-D geometry was drawn, and then extruded to a specific depth. At this point the *openappend* predicate is used to reopen the file created for ADAMS communication. This allows for more information to be inserted into the file. Using the boundary curves the coordinates of the start and end point of the curve was then obtained. To extrude a 2-D drawing it must be represented as a surface. This meant that a geometry need to start from its origin 0,0,0 and end at the same point, or even start at any point but "loop" and end back at the start point. Appendix B, page 103, shows the rules used to carry out this action under the section *startpoint*. Once one point of the curve is chosen the other point is subtracted. Then another curve with the considered second point is selected and the process is repeated

till it loops back to the first point. Defined by the material name, the component material is specified. Each component is also given an identification number, name, and a marker number. A marker is used in ADAMS to specify the location of the component. Each component has its own unique marker number. This is then used as a reference for further modification to the component. The process then searches the database via Name of entity and constructs all the other components required. No two components are repeated. Once again the file is closed once all the components are constructed.

The next section includes constraints in the multibody system. Here joints between components and specific global constraints are included into the file. The file is once again reopened using the openappend predicate. This time the fact used is:

entity2(id,name,_,Jtype,Orientation,Conn1,Conn2,location)

Where:	id	-	is the ID number of the row of information / constraint
	name	-	name of joint
	Jtype	-	type of joint – translation, rotational etc.
	Orientation	-	Orientation of the joint relative to the global/ground coordinates
	Conn1	-	Connection of the 1st component
	Conn2	-	Connection of the 2nd component
	location	-	X, Y, Z, coordinates of the joint/constraint (position)

The process used here is similar to that to create the geometry components. Appendix B, page 104, shows the code used and Appendix c, page 116, shows the result. These commands are used by ADAMS, to create the required joints and constraints. Most of the commands used are self-explanatory. As before all information is extracted from the Access database into the temporary internal database. All constraints are added by using their name so that no one constraint is repeated.

The last section required to complete the model for analysis was to incorporate the loads (forces) into the list of commands. Here as in the case of the constraints a different fact; entity3 is used. Entity3 is similar to entity2 with the difference of Ltype and force. Where Ltype represents the type of load, i.e. rotational gravitational, translational, etc, and force represents the actual load applied. As in all the other entities the loads are applied using its name entity so as not to duplicate any given load. This means that now the model is complete and can be analysed. The above section can be seen in Appendix B, on page 105 under the section *createloads*.

The last two sections of the commands list included simulating the model and the exporting of results. This task was carried out by simply inserting standard commands as shown in the list of commands in Appendix C. The commands in this section include starting a simulation, number of steps, duration and resetting the model to initial conditions. Once the simulation is completed the model is then saved in order to be able to export the results.

Since most relevant results in motion analysis, are those that occur at the constraints (joints), only these results were exported to the database. There are many formats of exporting results, included is a format that can be used by MSC Nastran. The results were exported in a spread sheet format for the various joints. Once complete a command is given to ADAMS to exit the software.

5.4 Application Example: Piston-Crank Mechanism

The following example describes how the behavioural and geometrical constraints between the components of an assembly can be presented. The mechanical system examined in this case is a simplified piston-crank assembly consisting of the following components: the crank (crank), crankshaft (shaft), the endblock (endblock), the connecting rod (conrod), and the piston (piston).

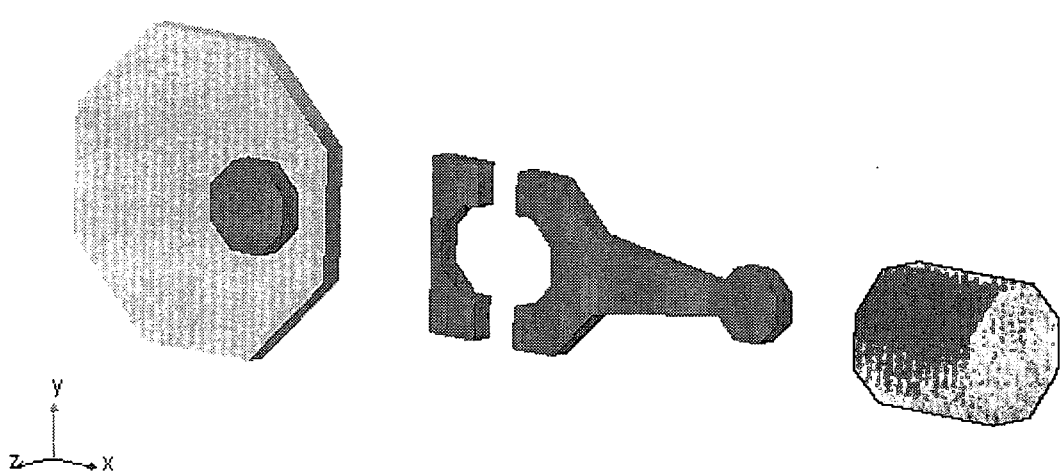


Figure 5.5. Components of the simplified piston-crank mechanism.

For the sake of simplicity (when representing the components using face-sets) circular components were modelled as octagons. A more accurate approximation would require

the non-linear features of the shape to be modelled using a larger number of edges. However, for the sake of this example, there would be no added benefit other than a more realistic visual appearance. The components, in order of appearance from left to right: crank together with the shaft, endblock, conrod and piston are shown in Figure 5.5.

One of the components is then chosen as the 'base' component. The actual choice of the 'base' component is arbitrary, as the tree-structure representing the mechanism can begin and expand from any component. The components are assembled in a hierarchical form determined by the connectivity of the individual components. The components of highest order, the 'root' component, is a component which can be geometrically constrained to the global coordinate system (The global coordinate system 'GCS' is a coordinate system to which any component can be referenced directly to provide exact positions as an alternative to the relative position between two components). Thus the root component is always constrained to both the 'GCS' and a component of lower order. A component in the lower order in this case is the default ground component (background of drawing page) as represented by ADAMS. It must be stated that the root component does not necessarily have to coincide with the base component, as this is determined by the initial selection of the base component.

The Microsoft Access database can be seen in Table 5.2. The location of each entity in the database is defined by the GCS as shown in Table 5.1. Connection 1 and Connection 2 represent the two entities to which a specific joint is connected to. Here as for all the other entities the location is defined by the GCS. The database is very

simple and self-explanatory (has been explained earlier on in the chapter in detail). The geometry of the shapes as shown is represented by facesets.

ID	Name	Shape	Joint Type	Material Name	E (GPa)	G (GPa)	nu	Density	Location	Load Type	Load	Orientation	Connection 1	Connection 2
2	Crank	faceset		steel	207E+9	80E+9	0.3	7850	18.94, 0, 0			0, 0, 0		
3	Conrod	faceset		cast_iron	96.5E+9	41E+9	0.25	7200	48.502, 50.335, 5			0, 0, 270		
4	Endblock	faceset		cast_iron	96.5E+9	41E+9	0.25	7200	36.502, 50.335, 5			0, 0, 270		
5	Piston	faceset		steel	207E+9	80E+9	0.3	7850	105.44, 18.475, 21.36			90, 90, 270		
6	Shaft	faceset		steel	207E+9	80E+9	0.3	7850	39.245, 23.085, 5			0, 0, 0		
7	Bolt1	joint	fixed						48.502, 18.71, 7.5			0, 0, 0	Endblock	Conrod
8	Bolt2	joint	fixed						48.502, 45.96, 7.5			0, 0, 0	Endblock	Conrod
9	Cshaft	joint	fixed						48.502, 32.335, 5			0, 0, 0	Crank	Shaft
10	joint1	joint	fixed						107.94, 32.335, 7.5			0, 0, 0	Conrod	Piston
11	joint4	joint	translational						107.94, 32.335, 7.5			90, 90, 0	piston	ground
12	CrankJoint	joint	revolute						32.335, 32.335, 0			0, 0, 0	crank	ground
13	load1	load							32.335, 32.335, 0	rotational	10E9	0, 0, 0	crank	ground

Table 5.1. Product database for piston crank assembly

The faceset information is stored in a file as and referred to when queried upon. The format of the crank is shown below.

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(14.94,52.61,0,21.06,52.61,0,25.39,56.94,0)

s(14.94,52.61,0,25.39,56.94,0,10.61,56.94,0)

s(25.39,56.94,0,10.61,56.94,0,10.61,63.06,0)

s(25.39,56.94,0,10.61,63.06,0,25.39,63.06,0)

s(10.61,63.06,0,25.39,63.06,0,14.94,67.39,0)

s(25.39,63.06,0,14.94,67.39,0,21.06,67.39,0)

The file is basically a text file. With the use of predicate logic as explained before the faceset information is transformed into a format represented by its X, Y, and Z coordinates. The code generated for this example to carry out the analysis in ADAMS, is shown in Appendix C. Figure 5.4 shows the complete model created in ADAMS, generated by the commands code in Appendix C. The code in Appendix C shows the various functions under the headings of *Materials, Parts, Constraints, Loads, Analyse model, Savemodel, Export results and Exit ADAMS.* The various commands used to generate the code have been explained earlier in the chapter. Figure 5.6 shows the complete assembly model created in ADAMS, generated by the commands code in Appendix C.

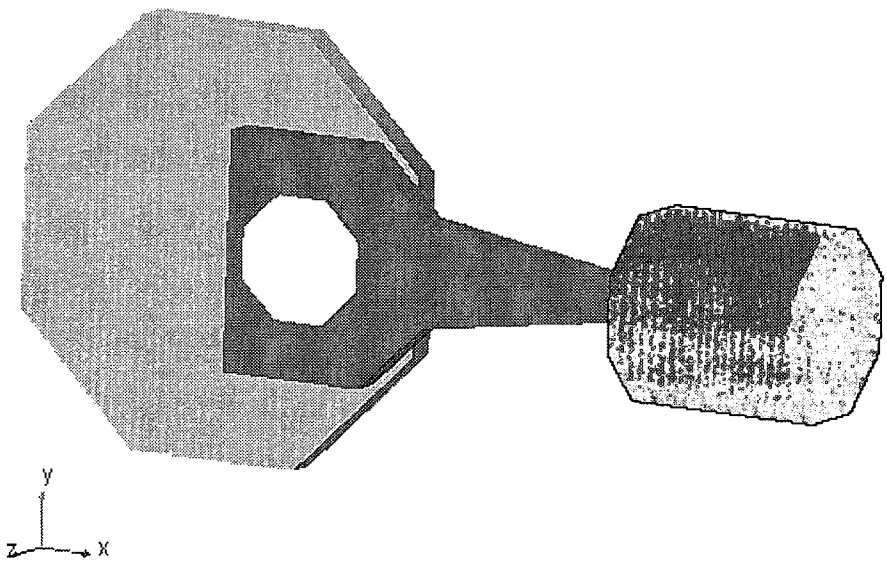


Figure 5.4. Analysis model in ADAMS

The model above includes the joints and forces used. On completing the simulation the model is saved and results are exported to a Microsoft Excel spreadsheet. This example is basic and simple in order to test out the integration code from start to end. Other smaller examples were also carried out using different geometry and loading conditions.

5.5 Summary

The development of a computer-aided integrated mechanical design system for the design and analysis of mechanical systems has been discussed in this chapter. The system is based on the combination of a CAE system based on a set of graphics libraries (Open Inventor) and commercially available motion analysis (ADAMS) code. The main aim of the system is to provide a seamless interface between the design generation and

analysis phases of the product development process. This has been achieved by considering two aspects:

1. The use of an extended product data model contained within a centralised product database. This database contains all the information to describe the geometric and functional characteristics of the design. The database is managed using a relational database management system.
2. The use of an automated mechanism to facilitate the exchange of mechanism specific data for the analysis of the mechanical system. This mechanism is based on the use of predicate logic to translate the product information into a script that may be used as input to the analysis code. User interaction is therefore not required in the construction of the analysis model.

An example was presented that served to demonstrate the application of the system to the analysis of a piston crank mechanism. The example consists of five components, six joints and one load. A model is created in ADAMS using the interface. This model is then analysed and results are then exported to a spreadsheet.

CHAPTER 6

CONCLUSION

The advancements made in the field of computer-aided engineering has revolutionised the engineering design process. The great advances in the speed of computers and software has greatly enhanced the engineer's ability to model designs. Geometric modelling on the computer has become the fastest-changing area of engineering design.

Virtual prototyping/engineering is making the traditional design process more intelligent and user friendly as discussed in the work presented here. Software tools used in the design of components of multibody systems are well developed and widely used. Artificial Intelligence (AI) methodologies will eventually become a natural and integral component of the set of computer based tools of engineers. These tools will then significantly elevate the role of computers in engineering from the present day emphasis on calculation to the much broader area of reasoning.

The main gains from integrated design and analysis systems, are the possibilities of using calculated component data like mass and moment of inertia from CAD in the simulation models, the automatic formulation of input files for the analysis cool and finally the visualisation of simulation results using the surface data of the solid models. Virtual prototyping relies on the complete integration of design and analysis phases so that the design configuration may be immediately evaluated.

The development of a computer-aided integrated mechanical design system for the design and analysis of mechanical multibody systems has been discussed in the work presented here. The research concluded with a working interface that resulted in the integration of CAD and motion analysis software.

However more work is required to streamline the system interface to achieve more sophistication and become more 'user friendly' to solve more complex designs. An integrated system is of great advantage to the designer. A virtual prototype of a multibody mechanism is obtainable, with the user not required to have an expertise in motion analysis. It is also a time saving operation, and means that more complex designs can now be undertaken.

The program developed here uses facets to generate the geometric models. This implies that constructing of the model can be carried out not only in ADAMS as done here, but in most analysis or CAD packages if required. This makes it universal and easy to implement with other applications. An example was presented that served to demonstrate the application of the system to the analysis of a simple two-dimensional piston-crank mechanism.

The system designed here is not high level of sophistication, and has been tested with a simple basic example. As mentioned previously the work carried out here was in two dimensions only.

With regards to future recommendations, 3-D modelling to further enhance the system would be considered. More complex measuring systems, i.e., placement of virtual gauges and more realistic loading conditions would also enhance the interface for more realistic virtual prototyping. The incorporation of standard components, such as bolts, bearings, pins etc. would not only reduce time but also make the system more intelligent.

REFERENCES

[1] Dym, C. L. and Little, P. (2000). *Engineering Design: A Project-Based Introduction*. John Wiley & Sons, United States of America.

[2] Dieter, J. E. (2000). *Engineering Design*. McGraw-Hill Book Company, Singapore.

[3] Ullman, D. G. (1997). *The Mechanical Design Process*. McGraw-Hill Book Company, United States of America.

[4] Halpern, M. (1997). Driving Toward Feature-Based Virtual Prototyping. *Conputer Graphics World*. September 1997.

[5] Lee, G. (1995). Virtual Prototyping on Personal Computers. *Mechanical Engineering*. Vol. 117, No. 7.

[6] Master of Engineering, Computer Aided Engineering Brochure from University of Southern California. *http://www.usc.edu/dept/civil_eng/dept/graduate/caetop1*

[7] Ohsuga, S. (1989). Toward intelligent CAD systems. *Computer – Aided Design*. **21**, 315 – 337.

[8] Thilmany, J. (1999). CAD meets CAE. *Mechanical Engineering*. October 1999.

[9] Bussler, M. (1997). How engineering will be done in the 21st Century. *Machine Design*. 23 October 1997.

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

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

[12] Manevitz, L. and Givoli, D. (1998). Automating the Finite Element Method: A Test Bed for Soft Computing. *http://citeseer.njnec.com*.

[13] Motion Simulation Pacakage. (2001). *Pro/Mechanica. http://www.ptc.com/ product/proe/sim/index.htm*

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

[15] Rasdorf, W. J. (1987). *Computers in Mechanical Engineering.* Butterworths, Boston.

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

[17] 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.

[18] Lakmazaheri, S. and Edwards, P. (1997). Linguistic Approach to 2D Geometric Modelling of Hierarchical Systems. *Journal of Computing in Civil Engineering*, July, 165 – 174.

[19] Damski, J. C. and Gero, J. S. (1996). A logic – based framework for shape representation. *Computer – Aided Design*, **28**, 169 – 181.

[20] Giraud, C. (1984). Presque half-planes: towards a general representation scheme. *Computer – Aided Design*, Vol. 16. No. 1, 17 – 24.

[21] Chase, S. C. (1997). Logic based design modelling with shape algebras. *Automation in Construction*, **6**, 311 – 322.

[22] Ganter, M. A. and Storti, D. W. (1993). Implicit Solid Modelling: A Renewed Method for Geometric Design. *Innovations in Engineering Design Education Resource Guide*, American Society of Mechanical Engineers.

[23] Wiegand, T. F. (1996). Interactive Rendering of CSG Models. *The Eurographics Association*, Vol. 15. No. 4, 249 – 261.

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

[25] Jankovic, L., Jankovic, S., Chan, A. C., and Little, G. H. (2000). Can bottom-up modelling in virtual reality replace conventional structural analysis methods?. *Proceedings of The conference on Construction Applications of Virtual Reality, 4 th ~ 5 th September 2000,* University of Teesside.

[26] Eastman, C. M. (1981). The Design of Assemblies. *SAE technical paper* #, 810197.

[27] Wesley, M. A., Lozano, P. T., and Liberman, L. I., Lavin, M. A., and Grossman, D. D. (1980). A Geometric Modelling system for automated Mechanical *Assembly. IBM J. Res. Dev*, **24**(1), 64 – 74.

[28] Beer, F. P. and Johnston, Jr. E. R. (1984). *Vector Mechanics for Engineers: Statics 4th edition*. McGraw-Hill Book Company, Japan.

[29] Pytel, A. and Kiusulaas, J. (1996). *Engineering Mechanics: Dynamics*. Harper Collins Publishers Inc, England.

[30] Barton, L. O. (1993). *Mechanism Analysis 2nd edition*. Marcel Dekker Inc, New York.

[31] Géradin, M. and Cardona, A. (2001). *Flexible Multibody Dynamics: A Finite Element Approach*. John Wiley & Sons Ltd, England.

[32] Erdman, A. G., Sandor, G. N. and Kota, S. (2001). *Mechanism Design: Analysis and Synthesis*. Prentice-Hall Inc, United States of America.

[33] MachineDesign, (2000). Making Mechanism Move. www. *machinedesign.com* / Basics of Engineering Design.

[34] Mortenson, M. E. (1985). *Geometric Modeling*, John Wiley & Sons Ltd, New York.

[35] Ohsuga, S. (1985). Introducing knowledge processing to CAD/CAM. *Finite Elements In Analysis and Design*. Vol. 1., 255 – 269.

[36] Gero, J. S. (1987). *Expert systems in computer aided design*. North-Holland. Amsterdam.

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

[38] Tasso, C. (1998). An introduction to Artificial Intelligence and to the development of Knowledge-Based systems. *Development of Knowledge-Based systems for Engineering*, SpringerWien, New York.

[39] Jameson, A., Paris, C., and Tasso, C. (1997). User Modeling. *Proceedings of the 6th Int. Conference on User Modeling*. New York.

[40] Blount, G. N. and Clark, S. (1994). Artificial Intelligence and design automation systems. *Journal of Engineering Design*. Vol. 5, No. 4.

[41] Tasso, C. and Oliveira, E. D. (1998). *Development of Knowledge-Based systems for Engineering.* Springer-Verlang, New York.

[42] Rafiq, M. Y., Bugmann, G., and Easterbrook, D. J. (1999). Guidelines for designing Neural Networks for Civil Engineering applications. *Proceedings of The Fifth International Conference on the Application of Artificial Intelligence to Civil & Structural Engineering.* Oxford, England.

[43] Benhamou, P. and Colmerauer, A. (1993). Constraint Logic Programming: Selected Research. *MIT Press.* Cambridge, MA.

[44] Kowalski, R. A. (1979). *Logic for Problem Solving.* North-Holland. New York.

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

[46] Wu, J. K.; Wang, J. H. and Feng, C. X. (1995). A logic-based mechanical system constraint model. *Engineering with Computers*, **11**, 157 – 166.

[47] 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.

[48] Yagiu, T. (1989). A predicate-logical method for modelling design objects. *Artificial Intelligence in Engineering,* **4**, 41 – 53.

[49] Lakmazaheri, S. (1998). Constraint logic programming for the analysis and partial synthesis of truss structures. *Artificial Intelligence for Engineering Design, Analysis, and Manufacturing,* **3**, 157 – 173.

[50] Tomiyama, T. and ten Hagen, P. J. W. (1990). Representing knowledge in two distinct descriptions: extensional vs. intensional. *Artificial Intelligence in Engineering,* **5**, 23 – 32.

[51] Treur, J. (1991). *A logical framework for design processes. In Intelligent CAD Systems III.* Springer-Verlang, Berlin.

[52] Mukunda, G. R.; Sotelino, E. D. and Hsieh, S. (1998). Distributed Finite Element Computations Using Object-Oriented Techniques. *Engineering with Computers*, **14**, 59 – 72.

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

[54] Bertino, E.; Negri, M.; Pelagatti, G. and Sbattella, L. (1992). Object-oriented query languages: the notion and the issued. *IEEE Trans. Knowledge Data Engr.*, **4**, 223 – 273.

[55] 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.

[56] Wernecke, J. (1994). *The Inventor Mentor.* Addison Wesley Publishing Company, United States of America.

[57] 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.

[58] Sherman, K. S. and Barcellos, A. (1992). *Calculus and Analytical Geometry. (5^{th} edition).* McGraw-Hill Book Company, New York.

APPENDIX A – WORKING MODEL 2D SCRIPT

```
Sub Main()
Dim Doc as wmdocument
dim Crank as wmbody
set doc = wm.new()
set Crank = doc.newbody ("circle")
Crank. px.value=0 :Crank. py.value=0
Crank. radius.value = 0.5
Dim Conrod as wmbody
set Conrod = doc.newbody ("polygon")
Conrod.px.value=0.25:Conrod.py.value
=0
Conrod.AddVertex 1, 0, 0.35
Conrod.AddVertex 2, 0, 0.4
Conrod.AddVertex 3, 0.22, 0.4
Conrod.AddVertex 4, 0.22, 0.1
Conrod.AddVertex 5, 1.4, 0.1
Conrod.AddVertex 6, 1.4, -0.1
Conrod.AddVertex 7, 0.22, -0.1
Conrod.AddVertex 8, 0.22, -0.4
Conrod.AddVertex 9, 0, -0.4
Conrod.AddVertex 10, 0, -0.35
Conrod.AddVertex 11, 0.15,0
Conrod.AddVertex 12, 0, 0.35
Conrod.DeleteVertex 13
Conrod.DeleteVertex 14
Conrod.DeleteVertex 15
Conrod.DeleteVertex 16
Conrod.DeleteVertex 17
Conrod.DeleteVertex 18

Dim EndBlock as wmbody
set EndBlock = doc.newbody
("polygon")
EndBlock.px.value=0.25:EndBlock.py.v
alue=0
EndBlock.AddVertex 1, 0, 0.35
EndBlock.AddVertex 2, 0, 0.4
EndBlock.AddVertex 3, -0.22, 0.4
EndBlock.AddVertex 4, -0.22, -0.4
EndBlock.AddVertex 5, 0, -0.4
EndBlock.AddVertex 6, 0, -0.35
EndBlock.AddVertex 7, -.15, 0
EndBlock.AddVertex 8, 0, 0.35
EndBlock.DeleteVertex 9
EndBlock.DeleteVertex 10
EndBlock.DeleteVertex 11
Dim Shell1 as wmbody
set Shell1 = doc.newbody ("polygon")
Shell1.px.value=0.25:Shell1.py.value=0
Shell1.AddVertex 1, 0, 0.3
Shell1.AddVertex 2, 0, 0.35
Shell1.AddVertex 3, 0.15, 0
Shell1.AddVertex 4, 0, -0.35
Shell1.AddVertex 5, 0, -0.3
Shell1.AddVertex 6, 0.1, 0
Shell1.AddVertex 7, 0, 0.3
Shell1.DeleteVertex 8
Shell1.DeleteVertex 9
Shell1.DeleteVertex 10
Dim Shell2 as wmbody
set Shell2 = doc.newbody ("polygon")
Shell2.px.value=0.25:Shell2.py.value=0
Shell2.AddVertex 1, 0, 0.3
Shell2.AddVertex 2, 0, 0.35
Shell2.AddVertex 3, -0.15, 0
Shell2.AddVertex 4, 0, -0.35
Shell2.AddVertex 5, 0, -0.3
Shell2.AddVertex 6, -0.1, 0
Shell2.AddVertex 7, 0, 0.3
Shell2.DeleteVertex 8
Shell2.DeleteVertex 9
Shell2.DeleteVertex 10
dim ConrodPosition as wmconstraint
set ConrodPosition=doc.newconstraint
("pin")
set ConrodPosition.point(2).body =
Conrod
ConrodPosition.point(2).px.value = 0
ConrodPosition.point(2).py.value = 0
set ConrodPosition.point(1).body =
Crank
ConrodPosition.point(1).px.value = 0.25
ConrodPosition.point(1).py.value = 0
dim EndBlockPosition as wmconstraint
set
EndBlockPosition=doc.newconstraint
("pin")
set EndBlockPosition.point(2).body =
EndBlock
EndBlockPosition.point(2).px.value = 0
EndBlockPosition.point(2).py.value = 0
set EndBlockPosition.point(1).body =
Crank
EndBlockPosition.point(1).px.value =
0.25
EndBlockPosition.point(1).py.value = 0
dim Shell1Position as wmconstraint
```

```
set Shell1Position=doc.newconstraint
("pin")
set Shell1Position.point(2).body =
Shell1
Shell1Position.point(2).px.value = 0
Shell1Position.point(2).py.value = 0
set Shell1Position.point(1).body =
Crank
Shell1Position.point(1).px.value = 0.25
Shell1Position.point(1).py.value = 0
dim Shell2Position as wmconstraint
set Shell2Position=doc.newconstraint
("pin")
set Shell2Position.point(2).body =
Shell2
Shell2Position.point(2).px.value = 0
Shell2Position.point(2).py.value = 0
set Shell2Position.point(1).body =
Crank
Shell2Position.point(1).px.value = 0.25
Shell2Position.point(1).py.value = 0
dim motor as wmconstraint
set motor = doc.newconstraint ("motor")
set motor.point(2).body = Crank
motor.point(2).px.value = 0
motor.point(2).py.value = 0
motor.motortype = "velocity"
motor.field.value = "200"
Dim piston as wmbody
set piston = doc.newbody ("rectangle")
piston.px.value=1.2:piston.py.value=0
piston.width.value=0.5:piston.height.val
ue=0.6
dim Pin as wmconstraint
set Pin=doc.newconstraint ("pin")
set Pin.point(1).body = piston
Pin.point(1).px.value = 0
Pin.point(1).py.value = 0
set Pin.point(2).body = Conrod
Pin.point(2).px.value = 1.2
Pin.point(2).py.value = 0
dim Cylinder as wmconstraint
set Cylinder=doc.newconstraint
("keyedhslot")
set Cylinder.point(2).body = piston
Cylinder.point(2).px.value = 0
Cylinder.point(2).py.value = 0
dim Bolt1 as wmconstraint
set Bolt1=doc.newconstraint
("squarepin")
set Bolt1.point(2).body = Conrod
Bolt1.point(2).px.value = 0
Bolt1.point(2).py.value = 0.375
set Bolt1.point(1).body = EndBlock
Bolt1.point(1).px.value = 0
Bolt1.point(1).py.value = 0.375
dim Bolt2 as wmconstraint
set Bolt2=doc.newconstraint
("squarepin")
set Bolt2.point(2).body = EndBlock
Bolt2.point(2).px.value = 0
Bolt2.point(2).py.value = -0.375
set Bolt2.point(1).body = Conrod
Bolt2.point(1).px.value = 0
Bolt2.point(1).py.value = -0.375
doc.run 500
Doc.reset
End Sub
```

APPENDIX B – VISUAL PROLOG CODE

```
/*************************************************************
                Copyright (c) Technikon Natal

 Project:  ADAMS INTERFACE
 FileName: ADAMS INTERFACE.PRO
 Purpose: No description
 Written by: Nitin Daya
 Comments:
*************************************************************/

include "adams interface.inc"
include "adams interface.con"
include "hlptopic.con"

%BEGIN_WIN Task Window
/*************************************************************
          Event handling for Task Window
*************************************************************/
domains
  id = integer
  shape = faceset(string);joint(string);load(string)
  name,jtype,ltype,mat_name,e,g,u,d,orientation,conn1,
  conn2,force,bc,
  location = string
  stringlist = string*
  clist = curve*
  curve = c(name,point,point)
  list = name*
  point = p(real,real,real)
  coord = string
  dist,x,y,z,lx,ly,lz = real

database - entity
  entity(id,name,shape,name,e,g,u,d,location)
  entity2(id,name,shape,jtype,orientation,conn1,conn2,location)
  entity3(id,name,shape,ltype,orientation,conn1,conn2,
          location,force)
  life(name)

database - surface
  s(real,real,real,real,real,real,real,real,real)
  s1(name,point,point,point)
  s2(name,list)
  c(name,point,point)
  plane(name,real,real,real,real)
  facelist(name,list)
  counter(integer)
  lastlist(list)
```

```
  templist(name,list)
  curvelist1(name,list)
  used(name)
  used1(name)
  point(name,point)
  curvelist(name,list)
  curve(name)
  solid(name,list)
  shell(string)
  curvelist2(name,integer)
  surfsolid(name)


predicates
  task_win_eh : EHANDLER
  nondeterm getdata(STMT_HANDLE)
  getcurrent(STMT_HANDLE)
  decider(STMT_HANDLE,id,string)
  nondeterm material
  nondeterm nextstep1
  nondeterm nextstep1a
  nondeterm nextstep2
  nondeterm nextstep3
  nondeterm nextstep4
  nondeterm nextstep5
  nondeterm nextstep6
  nondeterm changedata
  nondeterm changedata1
  nondeterm changedata2
  nondeterm changedata3
  nondeterm curvedata1(name,real,real,real,real,real,real)
  nondeterm curvedata2(name,real,real,real,real,real,real)
  nondeterm curvedata3(name,real,real,real,real,real,real)
  nondeterm makeplane
  nondeterm makeplane(name,real,real,real,real)
  nondeterm makeplane1(name,real,real,real,real)
  nondeterm makeplane2(name,real,real,real,real)
  nondeterm makelist
  nondeterm makelist1(name)
  nondeterm nextlist
  nondeterm nextlist1(name,name,list)
  nondeterm nextstep2(list,name)
  nondeterm checkcurve1(name,list,name,name,name)
  nondeterm checkcurve2(name,list,name,name,name)
  nondeterm checkcurve3(name,list,name,name,name)
  nondeterm subtract(name,list,list)
  nondeterm member(name,list)
  nondeterm makecurve(list,list,name)
  nondeterm makecurve1(name,list,name)
  nondeterm makecurve2(name,list,name)
  nondeterm makecurve3(name,list,name)
  nondeterm checksolid(name,list)
  nondeterm checksolid1(name,list)
  nondeterm makesolid
```

```
  nondeterm senddata(name)
  nondeterm startpoint(list)
  nondeterm startpoint1(list,list,point,point)
  nondeterm startpoint2(point)

constants

%BEGIN Task Window, CreateParms, 09:55:02-23.3.2001, Code
 automatically updated!
  task_win_Flags =
[wsf_SizeBorder,wsf_TitleBar,wsf_Close,wsf_Maximize,wsf_Minimize
,wsf_C
 lipSiblings]
  task_win_Menu  = res_menu(idr_task_menu)
  task_win_Title = "Adams Interface"
  task_win_Help  = idh_contents
%END Task Window, CreateParms

clauses
 getdata(H):-
      sql_FetchNext(H),
      getcurrent(H),!,
      getdata(H).

  getcurrent(H):-
      ID = sql_GetInteger(H,1),
      Shape = sql_GetString(H,3),
      decider(H,ID,Shape).

 decider(H,ID,"faceset"):-
      Name = sql_GetString(H,2),
      Mat_Name = sql_GetString(H,5),
      E = sql_GetString(H,6),
      G = sql_GetString(H,7),
      Nu = sql_GetString(H,8),
      D = sql_GetString(H,9),
      Location = sql_GetString(H,10),

      assertz(entity(ID,Name,faceset(Name),Mat_Name,E,G,Nu,D,
              Location),entity).

decider(H,ID,"joint"):-
      Name = sql_GetString(H,2),
      Location = sql_GetString(H,10),
      Jtype = sql_GetString(H,4),
      Orientation = sql_Getstring(H,13),
      Connection1 = sql_GetString(H,14),
      Connection2 = sql_GetString(H,15),
assertz(entity2(ID,Name,joint(Name),Jtype,Orientation,Connection
        1,Connection2,Location),entity).


 decider(H,ID,"load"):-
      Name = sql_GetString(H,2),
```

```
        Location = sql_GetString(H,10),
        Ltype = sql_GetString(H,11),
        Orientation = sql_Getstring(H,13),
        Connection1 = sql_GetString(H,14),
        Connection2 = sql_GetString(H,15),
        Force=sql_GetString(H,12),
        assertz(entity3(ID,Name,load(Name),Ltype,Orientation,
                Connection1,Connection2,Location,Force),entity).


/*****************************************************************
        THIS SECTION CREATES A SHAPE USING FACE-SETS!
*****************************************************************/


    nextstep1:-
        %not(started),
        openwrite(aview,"D:\\Nitin M-Tech\\Adams
                  Macros\\Exe\\Template1.cmd"),
        writedevice(aview),
        %SET UP NEW SESSION NEW MODEL
        write("default units length=mm mass=kg force=newton
               time=sec angle=deg  &"),nl,
        write("coordinate_system_type = cartesian orientation_type
              = body313"),nl,nl,
        write("model create  &"),nl,
        write("model_name=Test"),nl,
        write("view erase"),nl,nl,
        retractall(used(_)),
        material,!,
        write("defaults model part_name=ground"),nl,
        write("part attrib part_name=ground name_vis=off"),nl,
        write("defaults coordinate_system  &"),nl,
        write("default_coordinate_system = .Test.ground"),nl,
        write("part create rigid_body mass_properties  &"),nl,
        write("part_name = .Test.ground  &"),nl,
        write("material_type = .Test.steel"),nl,nl,
        closefile(aview),
        nextstep1a.

material:-
              entity(_,_,_,Mat_Name,E,_,Nu,D,_),
              not(used(Mat_Name)),
              asserta(used(Mat_Name)),
              write("material create  &"),nl,
        concat("material_name= .Test.",Mat_Name,Mat1),write(Mat1,
                "&"),nl,
        concat("youngs_modulus = ",E,E1),write(E1, "  &"),nl,
        concat("poissons_ratio = ",Nu,Nu1),write(Nu1, "  &"),nl,
        concat("density = ",D,D1),write(D1, "  &"),nl,nl,
        material;
        retractall(used(_)).
```

```
nextstep1a:-
      entity(_,Name,_,_,_,_,_,_,_),
         %dlg_note(Name),
            not(life(Name)),
            asserta(life(Name)),
            concat("D:\\Nitin M-Tech\\Adams Macros\\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(_)),
      retractall(used1(_)),
      curvelist(CL,CList),
      asserta(used(CL)),
      checksolid(CL,CList),!,
      senddata(Name),!,
      retractall(_,surface),
      nextstep1a;
      retractall(life(_)),
      nextstep2.

   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,!;
      write("There is no data, or the data in the file is of the
             incorrect format!").

   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.

   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,!;
    write("There is no data, or the data in the file is of the
          incorrect format!").

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.

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])).

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])).

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])).

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.

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).

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.

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.

makelist:-
    plane(Name,_,_,_,_),
    asserta(facelist("L1",[Name])),
    asserta(used(Name)),
    makelist1("L1"),!.

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);
    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.

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|[]]),!.

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|[]]);
    true.
nextlist1(Name,Name1,[H|T]):-
    s2(H,[C1,C2,C3]),
    templist(Name,List1),
    checkcurve1(Name,List1,Name1,C1,S1),
    lastlist(L1),
    nextstep2(L1,S1),
    templist(Name,List2),
    checkcurve2(Name,List2,Name1,C2,S2),
    lastlist(L2),
    nextstep2(L2,S2),
    templist(Name,List3),
    checkcurve3(Name,List3,Name1,C3,S3),
    lastlist(L3),
    nextstep2(L3,S3),
    nextlist1(Name,Name1,T).
```

```
nextstep2(L,S):-
    S <> "",
    retract(lastlist(L)),
    asserta(lastlist([S|L]));
    true.

checkcurve1(_,[],_,_,"").
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).

checkcurve2(_,[],_,_,"").
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).

checkcurve3(_,[],_,_,"").
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).
```

```
subtract(H,[H|Rest],Rest).
subtract(H,[Y|Rest],[Y|Rest1]):-
    subtract(H,Rest,Rest1).

makecurve([],_,_):-
    curvelist1(FL,List),
    not(used1(FL)),
    asserta(used1(FL)),
    makecurve(List,List,FL);
    true.
makecurve([H|T],List,FL):-
    makecurve1(H,List,FL),
    makecurve2(H,List,FL),
    makecurve3(H,List,FL),
    makecurve(T,List,FL).

makecurve1(H,[],FL):-
    s2(H,[C1,_,_]),
    curvelist(FL,List),
    not(member(C1,List)),
    retract(curvelist(FL,_)),
    asserta(curvelist(FL,[C1|List]));
    true.
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.

makecurve2(H,[],FL):-
    s2(H,[_,C2,_]),
    curvelist(FL,List),
    not(member(C2,List)),
    retract(curvelist(FL,_)),
    asserta(curvelist(FL,[C2|List]));
    true.
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.

makecurve3(H,[],FL):-
    s2(H,[_,_,C3]),
    curvelist(FL,List),
```

```
    not(member(C3,List)),
    retract(curvelist(FL,_)),
    asserta(curvelist(FL,[C3|List]));
    true.
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.

checksolid(_,[]):-
    curvelist(C,List),
    not(used(C)),
    asserta(used(C)),
    checksolid(C,List),!;
    asserta(solid("S1",[])),
    makesolid,!.
checksolid(C1,[H1|T1]):-
    not(used1(H1)),
    checksolid1(C1,[H1|T1]),!;
    checksolid(C1,T1).

checksolid1(C1,[H1|T1]):-
    curvelist(C2,List),
    not(used(C2)),
    member(H1,List),
    asserta(used1(H1)),
    checksolid(C1,T1),!;
    true.

makesolid:-
    used(S),
    retract(used(S)),
    retract(solid("S1",List)),
    asserta(solid("S1",[S|List])),
    makesolid,!;
    true.

member(Name,[Name|_]).
member(Name,[_|Rest]):-
    member(Name,Rest).

senddata(Name):-
    entity(ID,Name,_,Mat_Name,_,_,_,_,Location),
        openappend(aview,"D:\\Nitin M-Tech\\Adams
                    Macros\\Exe\\Template1.cmd"),
    writedevice(aview),
        %CREATE SIMPLE EXTRUSION FIGURE
    write("default coordinate_system  &"),nl,
    write("default_coordinate_system = .Test.ground"),nl,
```

```
      write("part create rigid_body name_and_position  &"),nl,
      concat("part_name = .Test.",Name,Na),write(Na,"  &"),nl,
           str_int(IDS,ID),
      concat("adams_id = ",IDS,ID1),write(ID1,"  &"),nl,
      concat("location = ",Location,L1),write(L1),nl,
      concat("defaults coordinate_system
default_coordinate_system = .Test.",Name,Na1),write(Na1),nl,
      write("marker create  &"),nl,
      concat("marker_name =
.Test.",Name,Na2),write(Na2,".Marker"),write(IDS,"  &"),nl,
      write(ID1, "  &"),nl,
      write("location = 0.0, 0.0, 0.0"),nl,
      write("part create rigid_body mass_properties  &"),nl,
      concat("part_name = .Test.",Name,Na3),write(Na3,"  &"),nl,
      concat("material_type =
.Test.",Mat_Name,Mat2),write(Mat2),nl,
      write("geometry create shape extrusion  &"),nl,
      concat("extrusion_name =
.Test.",Name,Na4),write(Na4,".Entity"),write(IDS,"  &"),nl,
      concat("reference_marker =
.Test.",Name,Na5),write(Na5,".Marker"),write(IDS,"  &"),nl,
      write("points_for_profile = "),
           curvelist(L,List),
      retractall(used(_)),
      asserta(used(L)),
      startpoint(List),!,
      write("length_along_z_axis = 20"),nl,nl,
      write("part attributes  &"),nl,
      concat("part_name = .Test.",Name,Na6),write(Na6,"  &"),nl,
      write("color = green"),nl,nl,nl,
      writedevice(screen),
      closefile(aview).

   startpoint([H|T]):-
      c(H,P1,P2),
      P1 = p(X1,Y1,Z1),
      P2 = p(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),
      write(X1s,",",Y1s,",",Z1s,",  &"),nl,
      write(X2s,",",Y2s,",",Z2s,",  &"),nl,
      startpoint1(T,T,P1,P2).

  startpoint1([],List,P1,P2):-
      startpoint1(List,List,P1,P2).
  startpoint1([H|T],List,P1,P2):-
      c(H,P2,P3),
      P1 = P3,
      startpoint2(P3);
      c(H,P3,P2),
```

```
      P1 = P3,
      startpoint2(P3);
      c(H,P2,P3),
      P3 = p(X3,Y3,Z3),
      str_real(X3s,X3),
      str_real(Y3s,Y3),
      str_real(Z3s,Z3),
      write(X3s,",",Y3s,",",Z3s,",  &"),nl,
      subtract(H,List,NList),
      startpoint1(T,NList,P1,P3);
      c(H,P3,P2),
      P3 = p(X3,Y3,Z3),
      str_real(X3s,X3),
      str_real(Y3s,Y3),
      str_real(Z3s,Z3),
      write(X3s,",",Y3s,",",Z3s,",  &"),nl,
      subtract(H,List,NList),
      startpoint1(T,NList,P1,P3);
      startpoint1(T,List,P1,P2).

  startpoint2(p(X,Y,Z)):-
      str_real(Xs,X),
      str_real(Ys,Y),
      str_real(Zs,Z),
      write(Xs,",",Ys,",",Zs,"  &"),nl.

/*****************************************************************
      END OF FACE-SETS!
*****************************************************************/

/*****************************************************************
      THIS SECTION CREATES JOINTS!
*****************************************************************/

nextstep2:-
      entity2(ID,Name,_,Jtype,Orientation,Conn1,Conn2,Location),
      not(used(Name)),
            asserta(used(Name)),
      openappend(aview,"D:\\Nitin M-Tech\\Adams
                  Macros\\Exe\\Template1.cmd"),
      writedevice(aview),
            %CREATE JOINTS
      write("default coordinate_system  &"),nl,
      write("default_coordinate_system = .Test"),nl,
      concat("marker create marker
=.Test.",Conn1,C1),write(C1,"."),write(Conn1,Name,"  &"),nl,
      concat("location=",Location,L2),write(L2,"  &"),nl,
      concat("orientation=",Orientation,O1),write(O1),nl,
      concat("marker create marker
=.Test.",Conn2,C2),write(C2,"."),write(Conn2,Name,"  &"),nl,
      concat("location=",Location,L2),write(L2,"  &"),nl,
      concat("orientation=",Orientation,O1),write(O1),nl,
      concat("constraint create joint ",Jtype,J1),write(J1,
               "&"),nl,
```

```
      concat("joint_name=.Test.",Name,N1),write(N1,"  &"),nl,
      str_int(IDString,ID),
      concat("adams_id = ",IDString,IDa),write(IDa,"  &"),nl,
      concat("i_marker_name =
.Test.",Conn1,C1a),write(C1a,"."),write(Conn1,Name,"  &"),nl,
      concat("j_marker_name =
.Test.",Conn2,C2a),write(C2a,"."),write(Conn2,Name),nl,
      concat("constraint attribute
             constraint_name=.Test.",Name,N2),write(N2),nl,nl,
      writedevice(screen),
      closefile(aview),
      nextstep2;
      retractall(used(_)),
      nextstep3.

/*****************************************************************
      END OF JOINTS!
*****************************************************************/

/*****************************************************************
      THIS SECTION CREATES LOADS (Forces)!
*****************************************************************/

nextstep3:-
      entity3(ID,Name,_,Ltype,Orientation,Conn1,Conn2,Location,
              Force),
      not(used(Name)),
       asserta(used(Name)),
      openappend(aview,"D:\\Nitin M-Tech\\Adams
                 Macros\\Exe\\Template1.cmd"),
      writedevice(aview),
           %CREATE LOADS
      write("default coordinate_system  &"),nl,
      write("default_coordinate_system = .Test"),nl,
      concat("marker create marker
=.Test.",Conn1,C11),write(C11,"."),write(Conn1,Name,"  &"),nl,
      concat("location=",Location,L2),write(L2,"  &"),nl,
      concat("orientation=",Orientation,O1),write(O1),nl,
      concat("marker create marker
=.Test.",Conn2,C12),write(C12,"."),write(Conn2,Name,"  &"),nl,
      concat("location=",Location,L2),write(L2,"  &"),nl,
      concat("orientation=",Orientation,O1),write(O1),nl,
      write("force create direct single_component_force  &"),nl,
      concat("single_component_force_name=.Test.",Name,N1),write
             (N1,"  &"),nl,
      str_int(IDString,ID),
      concat("adams_id = ",IDString,IDa),write(IDa,"  &"),nl,
      concat("type_of_freedom=",Ltype,L1),write(L1,"  &"),nl,
      write("action_only = on  &"),nl,
      concat("i_marker_name =
.Test.",Conn1,C1a),write(C1a,"."),write(Conn1,Name,"  &"),nl,
      concat("j_marker_name =
.Test.",Conn2,C2a),write(C2a,"."),write(Conn2,Name,"  &"),nl,
      concat("function = \34",Force,F1),write(F1,"\34"),nl,
```

```
      concat("mdi graphic_force
              object=.Test.",Name,N2),write(N2," type=1"),nl,nl,
      writedevice(screen),
      closefile(aview),
      nextstep3;
      retractall(used(_)),
      nextstep4.
/*****************************************************************
      END OF LOADS!
*****************************************************************/
```

/***

THIS SECTION STARTS THE SIMULATION AND SAVES THE MODEL (Analysis)!

***/

```
nextstep4:-
      openappend(aview,"D:\\Nitin M-Tech\\Adams
                  Macros\\Exe\\Template1.cmd"),
      writedevice(aview),
      write("var set var = .mdi.errno int=0"),nl,
      write("if cond=0"),nl,
      write("variable set variable=.gui.moag.c_simulate.
             run.tmpScriptName &"),nl,
      write("string = (eval( db_default( .system_defaults,
\34model\34 )//\34.\34//unique_name(\34SIM_SCRIPT\34) ))"),nl,
      write("simulation script create  &"),nl,
      write("type = auto_select  &"),nl,
      write("end_time = 5.0  &"),nl,
      write("number_of_steps = 50  &"),nl,
      write("sim_script_name =
             (eval(.gui.moag.c_simulate.run.tmpScriptName))"),nl,
      write("simulation single scripted  &"),nl,
      write("sim_script_name =
             (eval(.gui.moag.c_simulate.
                    run.tmpScriptName))  &"),nl,
      write("reset_before_and_after = yes"),nl,
      write("simulation script delete sim_script_name =
(eval(.gui.moag.c_simulate.run.tmpScriptName))"),nl,
      write("variable delete variable
             =.gui.moag.c_simulate.run.tmpScriptName"),nl,
      write("else"),nl,
      write("simulation single trans &"),nl,
      write("type = auto_select  &"),nl,
      write("end_time = 5.0  &"),nl,
      write("number_of_steps = 50"),nl,
      write("end"),nl,nl,
      write("var set var=.mdi.file_menu_tmp_name
             str=\34\34"),nl,
      write("if cond=(
```

```
            DB_EXISTS(\34.mdi.LAST_DATABASE_NAME\34))"),nl,
      write("if
            cond=(!str_is_space(.mdi.LAST_DATABASE_NAME))"),nl,
       write("var set var=.mdi.file_menu_tmp_name
            str=(eval(.mdi.LAST_DATABASE_NAME))"),nl,
      write("end"),nl,
      write("end"),nl,
      write("if cond=( str_is_space( .mdi.file_menu_tmp_name
           ))"),nl,
      write("if cond=( DB_DEFAULT(.system_defaults, \34model\34)
            != (none) )"),nl,
       write("var set var=.mdi.file_menu_tmp_name
            str=\34Test.bin\34"),nl,
      write("end"),nl,
      write("end"),nl,
      write("if cond=( !str_is_space(.mdi.file_menu_tmp_name)
           )"),nl,
      write("if cond=( ! file_exists( .mdi.file_menu_tmp_name )
           )"),nl,
       write("if cond=1"),nl,
       write("file binary write file=\34Test.bin\34"),nl,
       write("end"),nl,
        write("else"),nl,
       write("file binary write
            file=(eval(.mdi.file_menu_tmp_name)) alert=yes"),nl,
      write("end"),nl,
      write("var set var=.mdi.LAST_DATABASE_NAME
            str=\34Test.bin\34"),nl,
      write("else"),nl,
      write("var set var = .mdi.file_menu_status &"),nl,
       write("int = (eval(alert(\34Error\34, \34No model
            exists. Create one first or use 'Save Database
            As..'\34,\34OK\34,\34\34, \34\34, 1)))"),nl,
      write("end"),nl,nl,
      writedevice(screen),
      closefile(aview),
      nextstep5.

/*****************************************************************
     END OF THE SIMULATION AND SAVEING THE MODEL (Analysis)!
*****************************************************************/

/*****************************************************************
     THIS SECTION EXPORTS RESULTS TO A FILE !
*****************************************************************/

nextstep5:-
      entity2(_,Name,_,_,_,_,_,_),
      not(used(Name)),
       asserta(used(Name)),
      openappend(aview,"D:\\Nitin M-Tech\\Adams
                  Macros\\Exe\\Template1.cmd"),
      writedevice(aview),
      write("if cond=
```

```
          (DB_EXISTS(\34.gui.file_export.spreadsheet\34))"),nl,
      write("else"),nl,
      write("int field set
field=.gui.file_export.cadexport.file_name_f string=\34\34"),nl,
      write("int field set
field=.gui.file_export.cadexport.f_part_name string=\34\34"),nl,
      write("int cont disp cont=.gui.file_export.cadexport"),nl,
      write("end"),nl,
      write("if cond=
           (DB_EXISTS(\34.gui.file_export.spreadsheet\34))"),nl,
      write("file spread_sheet write  &"),nl,
      concat("file_name = \34",Name,Na1),write(Na1,"\34  &"),nl,
      concat("result_set_name = ",Name,Na2),write(Na2),nl,
      write("end"),nl,nl,
      writedevice(screen),
      closefile(aview),
      nextstep5;
      retractall(used(_)),
      nextstep6.
/*****************************************************************
      END OF THE EXPORTING RESULTS TO A FILE!
*****************************************************************/

nextstep6:-
      % To close the active Adams file and program
      openappend(aview,"D:\\Nitin M-Tech\\Adams
Macros\\Exe\\Template1.cmd"),
      writedevice(aview),
         write("mdi exit_macro"),nl,
         write("quit conf=no"),
         writedevice(screen),
      closefile(aview).
/*****************************************************************/

%**********DATABASE                    CONNECTION*************

task_win_eh(_Win,e_Create(_),0):-!,

      tb_project_toolbar_Create(_Win),
      tb_help_line_Create(_Win),
      retractall(_,entity),
      DBH = sql_Connect("AccessData","",""),
      H = sql_Exec(DBH,"SELECT * FROM Entity"),
      getdata(H),
      sql_drop(H),
      sql_Disconnect(DBH),
ifdef use_message
      msg_Create(100),
enddef
      !.
%END Task Window, e_Create


%*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-Analyse*-*-*-*-*-*-*-*-*-*-*-*-*-*-
%BEGIN Task Window,id_Adams_Analyse
```

```
task_win_eh(_Win,e_Menu(id_ADAMS_ANALYSE_Model,_ShiftCtlAlt),0):
-!,
     nextstep1,
     !.
%End Task Window,id_Adams_Analyse
%*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-


%MARK Task Window, new events

%BEGIN Task Window, id_help_contents
  task_win_eh(_Win,e_Menu(id_help_contents,_ShiftCtlAlt),0):-!,
     vpi_ShowHelp("adams interface.hlp"),
     !.
%END Task Window, id_help_contents

%BEGIN Task Window, id_help_about
  task_win_eh(Win,e_Menu(id_help_about,_ShiftCtlAlt),0):-!,
     dlg_about_dialog_Create(Win),
     !.
%END Task Window, id_help_about

%BEGIN Task Window, id_file_exit
  task_win_eh(Win,e_Menu(id_file_exit,_ShiftCtlAlt),0):-!,
     win_Destroy(Win),
     !.
%END Task Window, id_file_exit

%BEGIN Task Window, e_Size
  task_win_eh(_Win,e_Size(_Width,_Height),0):-!,
ifdef use_tbar
     toolbar_Resize(_Win),
enddef
ifdef use_message
     msg_Resize(_Win),
enddef
     !.
%END Task Window, e_Size

/****************************************************************
************************ START ADAMS*************************/

task_win_eh(_Win,e_Menu(id_ADAMS_Start_ADAMS,_ShiftCtlAlt),0):-
!,
system ("\"\34D:\\Program Files\\ADAMS 11.0\\common\\mdi.bat\34
aview ru-s i \""),
     !.


%END_WIN Task Window

/****************************************************************
          Invoking on-line Help
****************************************************************/
```

```
  project_ShowHelpContext(HelpTopic):-
      vpi_ShowHelpContext("adams interface.hlp",HelpTopic).

/*************************************************************
                  Main Goal
*************************************************************/

goal

ifdef use_mdi
  vpi_SetAttrVal(attr_win_mdi,b_true),
enddef
ifdef ws_win
  ifdef use_3dctrl
    vpi_SetAttrVal(attr_win_3dcontrols,b_true),
  enddef
enddef
  vpi_Init(task_win_Flags,task_win_eh,task_win_Menu,"adams
interface",task_win_Title).

%BEGIN_TLB Project toolbar, 09:55:02-23.3.2001, Code
automatically updated!

/*************************************************************
      Creation of toolbar: Project toolbar
*************************************************************/

clauses

  tb_project_toolbar_Create(_Parent):-
ifdef use_tbar
      toolbar_create(tb_top,0xC0C0C0,_Parent,

      [tb_ctrl(id_file_new,pushb,idb_new_up,idb_new_dn,idb_new_u
p,"New;New file",1,1),

tb_ctrl(id_file_open,pushb,idb_open_up,idb_open_dn,idb_open_up,"
Open;Open file",1,1),

tb_ctrl(id_file_save,pushb,idb_save_up,idb_save_dn,idb_save_up,"
Save;File save",1,1),
            separator,

tb_ctrl(id_edit_undo,pushb,idb_undo_up,idb_undo_dn,idb_undo_up,"
Undo;Undo",1,1),

tb_ctrl(id_edit_redo,pushb,idb_redo_up,idb_redo_dn,idb_redo_up,"
Redo;Redo",1,1),
            separator,

tb_ctrl(id_edit_cut,pushb,idb_cut_up,idb_cut_dn,idb_cut_up,"Cut;
Cut to clipboard",1,1),
```

```
tb_ctrl(id_edit_copy,pushb,idb_copy_up,idb_copy_dn,idb_copy_up,"
Copy;Copy to clipboard",1,1),

tb_ctrl(id_edit_paste,pushb,idb_paste_up,idb_paste_dn,idb_paste_
up,"Paste;Paste from clipboard",1,1),
            separator,
            separator,

tb_ctrl(id_help_contents,pushb,idb_help_up,idb_help_down,idb_hel
p_up,"Help;Help",1,1)]),
enddef
     true.
%END_TLB Project toolbar

%BEGIN_TLB Help line, 09:55:02-23.3.2001, Code automatically
updated!
/*************************************************************
     Creation of toolbar: Help line
*************************************************************/

clauses

  tb_help_line_Create(_Parent):-
ifdef use_tbar
     toolbar_create(tb_bottom,0xC0C0C0,_Parent,

     [tb_text(idt_help_line,tb_context,452,0,4,10,0x0,"")]),
enddef
     true.
%END_TLB Help line

%BEGIN_DLG About dialog
/*************************************************************
     Creation and event handling for dialog: About dialog
*************************************************************/

constants

%BEGIN About dialog, CreateParms, 09:55:02-23.3.2001, Code
automatically updated!
  dlg_about_dialog_ResID = idd_dlg_about
  dlg_about_dialog_DlgType = wd_Modal
  dlg_about_dialog_Help = idh_contents
%END About dialog, CreateParms

predicates

  dlg_about_dialog_eh : EHANDLER

clauses

  dlg_about_dialog_Create(Parent):-
```

```
     win_CreateResDialog(Parent,dlg_about_dialog_DlgType,dlg_ab
out_dialog_ResID,dlg_about_dialog_eh,0).

%BEGIN About dialog, idc_ok _CtlInfo

dlg_about_dialog_eh(_Win,e_Control(idc_ok,_CtrlType,_CtrlWin,_Ct
rlInfo),0):-!,
     win_Destroy(_Win),
     !.
%END About dialog, idc_ok _CtlInfo
%MARK About dialog, new events

  dlg_about_dialog_eh(_,_,_):-!,fail.

%END_DLG About dialog
```

APPENDIX C – ADAMS COMMANDS

```
default units length=mm mass=kg force=newton time=sec angle=deg  &
coordinate_system_type = cartesian orientation_type = body313

model create  &
model_name=Test
view erase

! MATERIALS

material create  &
material_name= .Test.steel  &
youngs_modulus = 207E+9  &
poissons_ratio = 0.3  &
density = 7850  &

material create  &
material_name= .Test.cast_iron  &
youngs_modulus = 96.5E+9  &
poissons_ratio = 0.25  &
density = 7200  &

! PARTS

defaults model part_name=ground
part attrib part_name=ground name_vis=off
defaults coordinate_system  &
default_coordinate_system = .Test.ground
part create rigid_body mass_properties  &
part_name = .Test.ground  &
material_type = .Test.steel

default coordinate_system  &
default_coordinate_system = .Test.ground
part create rigid_body name_and_position  &
part_name = .Test.Crank  &
adams_id = 2  &
location = 18.94, 0, 0
defaults coordinate_system default_coordinate_system = .Test.Crank
marker create  &
marker_name = .Test.Crank.Marker2  &
adams_id = 2  &
location = 0.0, 0.0, 0.0
part create rigid_body mass_properties  &
part_name = .Test.Crank  &
material_type = .Test.steel
geometry create shape extrusion  &
extrusion_name = .Test.Crank.Entity2  &
reference_marker = .Test.Crank.Marker2  &
points_for_profile = 26.79,0,0,  &
45.73,18.94,0,  &
45.73,45.73,0,  &
26.79,64.67,0,  &
0,64.67,0,  &
```

```
-18.94,45.73,0,  &
-18.94,18.94,0,  &
0,0,0,  &
26.79,0,0  &
length_along_z_axis = 5

part attributes  &
part_name = .Test.Crank  &
color = green


default coordinate_system  &
default_coordinate_system = .Test.ground
part create rigid_body name_and_position  &
part_name = .Test.Shaft  &
adams_id = 6  &
location = 39.245, 23.085, 5
defaults coordinate_system default_coordinate_system = .Test.Shaft
marker create  &
marker_name = .Test.Shaft.Marker6  &
adams_id = 6  &
location = 0.0, 0.0, 0.0
part create rigid_body mass_properties  &
part_name = .Test.Shaft  &
material_type = .Test.steel
geometry create shape extrusion  &
extrusion_name = .Test.Shaft.Entity6  &
reference_marker = .Test.Shaft.Marker6  &
points_for_profile = 18.5,5.43,0,  &
13.09,0,0,  &
5.43,0,0,  &
0,5.43,0,  &
0,13.09,0,  &
5.43,18.5,0,  &
13.09,18.5,0,  &
18.5,13.09,0,  &
18.5,5.43,0  &
length_along_z_axis = 5

part attributes  &
part_name = .Test.Shaft  &
color = green


default coordinate_system  &
default_coordinate_system = .Test.ground
part create rigid_body name_and_position  &
part_name = .Test.Endblock  &
adams_id = 4  &
location = 36.502, 50.335, 5
defaults coordinate_system default_coordinate_system = .Test.Endblock
marker create  &
marker_name = .Test.Endblock.Marker4  &
adams_id = 4  &
location = 0.0, 0.0, 0.0
part create rigid_body mass_properties  &
part_name = .Test.Endblock  &
material_type = .Test.cast_iron
geometry create shape extrusion  &
extrusion_name = .Test.Endblock.Entity4  &
```

```
reference_marker = .Test.Endblock.Marker4  &
points_for_profile = 36,12,0,  &
27.25,12,0,  &
27.25,8.17,0,  &
21.82,2.76,0,  &
14.16,2.76,0,  &
8.75,8.17,0,  &
8.75,12,0,  &
0,12,0,  &
0,0,0,  &
36,0,0,  &
36,12,0  &
length_along_z_axis = 5

part attributes  &
part_name = .Test.Endblock  &
color = green


default coordinate_system  &
default_coordinate_system = .Test.ground
part create rigid_body name_and_position  &
part_name = .Test.Conrod  &
adams_id = 3  &
location = 48.502, 50.335, 5
defaults coordinate_system default_coordinate_system = .Test.Conrod
marker create  &
marker_name = .Test.Conrod.Marker3  &
adams_id = 3  &
location = 0.0, 0.0, 0.0
part create rigid_body mass_properties  &
part_name = .Test.Conrod  &
material_type = .Test.cast_iron
geometry create shape extrusion  &
extrusion_name = .Test.Conrod.Entity3  &
reference_marker = .Test.Conrod.Marker3  &
points_for_profile = 14.94,67.39,0,  &
21.06,67.39,0,  &
25.39,63.06,0,  &
25.39,56.94,0,  &
21.06,52.61,0,  &
26,20,0,  &
36,10,0,  &
36,0,0,  &
27.25,0,0,  &
27.25,3.83,0,  &
21.82,9.24,0,  &
14.16,9.24,0,  &
8.75,3.83,0,  &
8.75,0,0,  &
0,0,0,  &
0,10,0,  &
10,20,0,  &
14.94,52.61,0,  &
10.61,56.94,0,  &
10.61,63.06,0,  &
14.94,67.39,0  &
length_along_z_axis = 5

part attributes  &
```

```
part_name = .Test.Conrod  &
color = green


default coordinate_system  &
default_coordinate_system = .Test.ground
part create rigid_body name_and_position  &
part_name = .Test.Piston  &
adams_id = 5  &
location = 105.44, 18.475, 21.36
defaults coordinate_system default_coordinate_system = .Test.Piston
marker create  &
marker_name = .Test.Piston.Marker5  &
adams_id = 5  &
location = 0.0, 0.0, 0.0
part create rigid_body mass_properties  &
part_name = .Test.Piston  &
material_type = .Test.steel
geometry create shape extrusion  &
extrusion_name = .Test.Piston.Entity5  &
reference_marker = .Test.Piston.Marker5  &
points_for_profile = 27.72,8.12,0,  &
19.58,0,0,  &
8.12,0,0,  &
0,8.12,0,  &
0,19.58,0,  &
8.12,27.72,0,  &
19.58,27.72,0,  &
27.72,19.58,0,  &
27.72,8.12,0  &
length_along_z_axis = 5

part attributes  &
part_name = .Test.Piston  &
color = green

! CONSTRAINTS

default coordinate_system  &
default_coordinate_system = .Test
marker create marker =.Test.Endblock.EndblockBolt1  &
location=48.502, 18.71, 7.5  &
orientation=0, 0, 0
marker create marker =.Test.Conrod.ConrodBolt1  &
location=48.502, 18.71, 7.5  &
orientation=0, 0, 0
constraint create joint fixed  &
joint_name=.Test.Bolt1  &
adams_id = 7  &
i_marker_name = .Test.Endblock.EndblockBolt1  &
j_marker_name = .Test.Conrod.ConrodBolt1
constraint attribute constraint_name=.Test.Bolt1

default coordinate_system  &
default_coordinate_system = .Test
marker create marker =.Test.Endblock.EndblockBolt2  &
location=48.502, 45.96, 7.5  &
orientation=0, 0, 0
marker create marker =.Test.Conrod.ConrodBolt2  &
location=48.502, 45.96, 7.5  &
```

```
orientation=0, 0, 0
constraint create joint fixed  &
joint_name=.Test.Bolt2  &
adams_id = 8  &
i_marker_name = .Test.Endblock.EndblockBolt2  &
j_marker_name = .Test.Conrod.ConrodBolt2
constraint attribute constraint_name=.Test.Bolt2

default coordinate_system  &
default_coordinate_system = .Test
marker create marker =.Test.Crank.CrankCshaft  &
location=48.502, 32.335, 5  &
orientation=0, 0, 0
marker create marker =.Test.Shaft.ShaftCshaft  &
location=48.502, 32.335, 5  &
orientation=0, 0, 0
constraint create joint fixed  &
joint_name=.Test.Cshaft  &
adams_id = 9  &
i_marker_name = .Test.Crank.CrankCshaft  &
j_marker_name = .Test.Shaft.ShaftCshaft
constraint attribute constraint_name=.Test.Cshaft

default coordinate_system  &
default_coordinate_system = .Test
marker create marker =.Test.piston.pistonjoint4  &
location=107.94, 32.335, 7.5  &
orientation=90, 90, 0
marker create marker =.Test.ground.groundjoint4  &
location=107.94, 32.335, 7.5  &
orientation=90, 90, 0
constraint create joint translational  &
joint_name=.Test.joint4  &
adams_id = 11  &
i_marker_name = .Test.piston.pistonjoint4  &
j_marker_name = .Test.ground.groundjoint4
constraint attribute constraint_name=.Test.joint4

default coordinate_system  &
default_coordinate_system = .Test
marker create marker =.Test.crank.crankCrankJoint  &
location=32.335, 32.335, 0  &
orientation=0, 0, 0
marker create marker =.Test.ground.groundCrankJoint  &
location=32.335, 32.335, 0  &
orientation=0, 0, 0
constraint create joint revolute  &
joint_name=.Test.CrankJoint  &
adams_id = 12  &
i_marker_name = .Test.crank.crankCrankJoint  &
j_marker_name = .Test.ground.groundCrankJoint
constraint attribute constraint_name=.Test.CrankJoint

default coordinate_system  &
default_coordinate_system = .Test
marker create marker =.Test.Conrod.Conrodjoint1  &
location=107.94, 32.335, 7.5  &
orientation=0, 0, 0
marker create marker =.Test.Piston.Pistonjoint1  &
location=107.94, 32.335, 7.5  &
```

```
orientation=0, 0, 0
constraint create joint fixed  &
joint_name=.Test.joint1  &
adams_id = 10  &
i_marker_name = .Test.Conrod.Conrodjoint1  &
j_marker_name = .Test.Piston.Pistonjoint1
constraint attribute constraint_name=.Test.joint1

! LOADS

default coordinate_system  &
default_coordinate_system = .Test
marker create marker =.Test.crank.crankload1  &
location=32.335, 32.335, 0  &
orientation=0, 0, 0
marker create marker =.Test.ground.groundload1  &
location=32.335, 32.335, 0  &
orientation=0, 0, 0
force create direct single_component_force  &
single_component_force_name=.Test.load1  &
adams_id = 13  &
type_of_freedom=rotational  &
action_only = on  &
i_marker_name = .Test.crank.crankload1  &
j_marker_name = .Test.ground.groundload1  &
function = "10E9"
mdi graphic_force object=.Test.load1 type=1

! ANALYSE MODEL

var set var = .mdi.errno int=0
if cond=0
variable set variable=.gui.moag.c_simulate.run.tmpScriptName &
string = (eval( db_default( .system_defaults, "model"
)//"."//unique_name("SIM_SCRIPT") ))
simulation script create  &
type = auto_select  &
end_time = 5.0  &
number_of_steps = 50  &
sim_script_name = (eval(.gui.moag.c_simulate.run.tmpScriptName))
simulation single scripted  &
sim_script_name = (eval(.gui.moag.c_simulate.run.tmpScriptName))  &
reset_before_and_after = yes
simulation script delete sim_script_name =
(eval(.gui.moag.c_simulate.run.tmpScriptName))
variable delete variable =.gui.moag.c_simulate.run.tmpScriptName
else
simulation single trans &
type = auto_select  &
end_time = 5.0  &
number_of_steps = 50
end


! SAVE MODEL

var set var=.mdi.file_menu_tmp_name str=""
if cond=( DB_EXISTS(".mdi.LAST_DATABASE_NAME"))
if cond=(!str_is_space(.mdi.LAST_DATABASE_NAME))
```

```
var set var=.mdi.file_menu_tmp_name
str=(eval(.mdi.LAST_DATABASE_NAME))
end
end
if cond=( str_is_space( .mdi.file_menu_tmp_name ))
if cond=( DB_DEFAULT(.system_defaults, "model") != (none) )
var set var=.mdi.file_menu_tmp_name str="Test.bin"
end
end
if cond=( !str_is_space(.mdi.file_menu_tmp_name) )
if cond=( ! file_exists( .mdi.file_menu_tmp_name ) )
if cond=1
file binary write file="Test.bin"
end
else
file binary write file=(eval(.mdi.file_menu_tmp_name)) alert=yes
end
var set var=.mdi.LAST_DATABASE_NAME str="Test.bin"
else
var set var = .mdi.file_menu_status &
int = (eval(alert("Error", "No model exists. Create one first or use
'Save Database As..'","OK","", "", 1)))
end
```

! EXPORT RESULTS

```
if cond= (DB_EXISTS(".gui.file_export.spreadsheet"))
else
int field set field=.gui.file_export.cadexport.file_name_f string=""
int field set field=.gui.file_export.cadexport.f_part_name string=""
int cont disp cont=.gui.file_export.cadexport
end
if cond= (DB_EXISTS(".gui.file_export.spreadsheet"))
file spread_sheet write  &
file_name = "joint2"  &
result_set_name = joint2
end

if cond= (DB_EXISTS(".gui.file_export.spreadsheet"))
else
int field set field=.gui.file_export.cadexport.file_name_f string=""
int field set field=.gui.file_export.cadexport.f_part_name string=""
int cont disp cont=.gui.file_export.cadexport
end
if cond= (DB_EXISTS(".gui.file_export.spreadsheet"))
file spread_sheet write  &
file_name = "joint3"  &
result_set_name = joint3
end

if cond= (DB_EXISTS(".gui.file_export.spreadsheet"))
else
int field set field=.gui.file_export.cadexport.file_name_f string=""
int field set field=.gui.file_export.cadexport.f_part_name string=""
int cont disp cont=.gui.file_export.cadexport
end
if cond= (DB_EXISTS(".gui.file_export.spreadsheet"))
file spread_sheet write  &
file_name = "joint4"  &
```

```
result_set_name = joint4
end

if cond= (DB_EXISTS(".gui.file_export.spreadsheet"))
else
int field set field=.gui.file_export.cadexport.file_name_f string=""
int field set field=.gui.file_export.cadexport.f_part_name string=""
int cont disp cont=.gui.file_export.cadexport
end
if cond= (DB_EXISTS(".gui.file_export.spreadsheet"))
file spread_sheet write  &
file_name = "joint1"  &
result_set_name = joint1
end


! EXIT ADAMS

mdi exit_macro
quit conf=no
```