National Library of Canada

Acquisitions and Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada

Direction des acquisitions et des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Canada

UniSet: An Object-Oriented Knowledge Based Environment for Computer Integrated Manufacturing (CIM)

Thesis submitted

to the School of Graduate Studies

in partial fulfillment of the requirements for the

degree of Doctor of Philosophy in

Mechanical Engineering

by

Kyunghyun Choi

Ottawa-Carleton Institute for

Mechanical and Aeronautical Engineering

University of Ottawa

Ottawa, Ontario

Canada, K1N 6N5

ISBN 0-612-11545-3

Canada

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

ABSTRACT

Flexible Manufacturing Cells (FMC) may be considered the most significant development in small batch manufacturing. Setting up and operating costs prove to be the most major hiderance to large scale implementation and use of the FMCs, particularly by small and medium size industries. Incompatibility between the different components constituting the cells, and the lack of a unified language/approach to programming and coordinating them, are cited as the cause of the complexity of setting up and subsequently operating the cells.

In order to eliminate these difficulties, a new philosophy for setting up, programming, and control of FMCs has been developed. This thesis reports part of the effort for developing the new "UniSet" philosophy and the components of the UniSet Environment. UniSet has been developed as a Unified Manufacturing Instruction Set based on a comparison of machine tool and programming primitives. UniSet allows programmers, if they so desire, to concentrate on only one instruction set rather than numerous machine programming languages.

Task level UniSet has been designed using an object-oriented and a knowledge based approach to eliminate cell programming difficulties. An expert cell programmer can be replaced by a less experienced manufacturing operator.

A software environment for Computer Integrated Manufacturing (CIM), called UniSet Environment, has been developed to address the shortcomings of setting up and operating FMCs comprising incompatible components from the different suppliers. The UniSet Environment

provides the users with a consistent platform to configure, program, simulate, and control an FMC irrespective of the nature of its constituents. The Environment is coded an Object-Oriented Programming (OOP) language, Smalltalk. The Environment consists of four modules which capture complete information about the cell program, existing cell facilities and databases, and distribute the information into the corresponding Object Models. Each of the module is responsible for a specific function.

The Setup module is primarily used for defining the cell, its components and their functionalities, and reconfiguring a cell or building preparation databases. The Programming module allows the user to edit cell programs which may be developed in UniSet, Task level UniSet, or native machine languages. The programs written in Task level UniSet are used to generate detailed UniSet codes which are translated into specific machine codes by the generation and translation module.

The Simulation/Compilation module is used for generating the control structure - Task Initiation Diagram (TID) - for the operation of the cell according to the user program as well as knowledge contained in the databases. The Control module is used to coordinate the workstation, as well as carry out the task of harmonizing between the different communication protocols of the devices.

ACKNOWLEDGMENTS

I would like to express my appreciation to my supervisor, Dr. A. Fahim, for his continued support and guidance, and for his sincere personal concern.

Special thanks to Dr. Y. Lee at University of Ottawa and K.C. Kim and K. Cho at Pusan National University for their moral support and encouragement during study. I am particularly thankful for the assistance and friendship of S. Rhi, M. Kim, B. Kim and I wish them great success.

I am indebted to many colleagues and friends, Rudy Tolkamp, Gilles Doucet for their various contribution to this work.

I would especially like to thank my father, mother, and brothers for their support during my studies. Finally, I want to thank my wife, Youngok, and my children, Hye-Mi, and Hye-Ji Genevieve, for their love, support and encouragement.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1
INTRODUCTION

This chapter addresses the definition of the problem based on the current status of implementation of Flexible Manufacturing Systems (FMSs). Existing automation languages and environments, and Object-Oriented techniques are described in order to provide the background knowledge associated with the current effort to create a Computer Integrated Manufacturing (CIM) environment. The suggested solution and the thesis overview are outlined in the last section.

1.1 Problem Definition

One of the most recent developments in automation has been Flexible Manufacturing Systems (FMSs). The concept of FMSs involves the complete manufacturing activity from the head office to the shop floor. In the ideal FMS, paper exchange would be virtually eliminated and information would be automatically exchanged between computers and machine controllers. The shop floor would be composed of Flexible Manufacturing Cells (FMCs) interconnected by material transportation devices such as Automated Guided Vehicles (AGVs) or conveyors.

The Flexible Manufacturing Cell is generally recognized as the best production tool for small to medium batch manufacturing. An FMC consists of a number of programmable devices, such as

robots, Numerically Controlled (NC) machines, automation controllers, and vision systems, as well as suitable sensors, interacting harmoniously to turn out a product. The main advantage of FMCs is that they are flexible and can therefore be scheduled to manufacture different products or can be easily reprogrammed to handle changes in product design. FMCs typically perform machining operations, but can also perform assembly. A machining cell would include one or more machine tools serviced by a suitable robot. An assembly cell may include one or more robots interacting with automatic clamps and fixtures, and a suitable in-feeder and output buffer.

Deciding on how to effectively integrate cell components into one environment is difficult. Currently, it is only possible to build systems when machine-to-machine interface problems have been designed-out, which typically involves buying all equipment from the same vendor. Unfortunately, FMCs are usually constructed from components provided by multiple vendors, or from newly developed components, which rarely conform to unifying standards. Each producer has designed its own sophisticated machines and robots with dedicated control languages. While this approach is counterproductive from the standpoint of developing integrated manufacturing cell technology, it allows for rapid advances and development at the machine level. Due to the lack of standard communication protocols, programming environments, and languages, the problems associated with programming and coordinating the cell, with all its incompatible components, are left to the user.

In order to operate a manufacturing cell, information about process plans and part programs must be prepared. How to produce this information quickly and effectively is critical to the operation. The required information is generated by specific modules like process planners, or part programming modules. Other modules for cell control, and monitoring, manipulate the information

during the operation of the manufacturing system. Due to the lack of integration of these functions, the time required to develop a complete cell program for a product can be very long.

1.2 Background

Existing automation languages and environments that have been developed successfully are reviewed here in detail. The Object-Oriented technique, specifically associated with Smalltalk, is also described.

1.2.1 Existing Automation Languages and Environments

Over the last few years there have been a number of research efforts to solve the problems associated with developing automated manufacturing facilities. Existing and proposed manufacturing languages and programming environments are grouped in the following sections based on their field of applications.

1.2.1.1 Industrial Robot Programming Languages

Robots can be programmed in two basically different ways: on-line and off-line. On-line programming, also referred to as teach programming, consists of moving the robot manually through the desired motion path in order to record the path into the controller memory. The teach pendant is usually used to control the various joint motors, and guide the robot arm and wrist through a series of points in space. The robot later repeats the motions it has been "taught" whenever called upon to do so.

In off-line programming, on the other hand, a computer program is written (using any one of a number of special robot programming languages) and later loaded into the controller memory. Current approaches to programming can be classified into two major categories: robot-level languages and task level languages. In a robot-level language, a task is explicitly described as a sequence of robot motions. The main advantage of a robot-level language is that it enables the data from external sensors, such as vision and force, to be used in modifying the robot motions. On the other hand, task level programming describes the task by specifying goals for the manipulation of objects, rather than the motions of the robot needed to achieve those goals. Task-level programming is still in the research stage [1].

Some general guidelines for developments in robot programming languages and systems, are suggested by Lozano-Perez [2]. According to his proposals, the key requirements for robot programming systems are in the areas of sensing, world modeling, motion specification, flow of control, and programming support. With these requirements in mind, some robot programming languages that have been developed are reviewed here after.

WAVE

The WAVE system, developed at Stanford, was the earliest system designed as a general purpose robot programming language [2]. WAVE was a new language, whose syntax was modeled after the assembly language of the PDP-10 and produced a trajectory file which was executed on-line by a dedicated PDP-6. The philosophy in WAVE was that motions could be preplanned and that only small deviations from these motions would happen during execution. This decision was

motivated by the computation-intensive algorithms employed by WAVE for trajectory planning and dynamic compensation. Better algorithms and faster computers have rendered this rationale obsolete for the design of robot systems.

In spite of WAVE's low-level syntax, the system provided an extensive repertoire of high-level functions. WAVE pioneered several important mechanisms in robot programming systems; among these were

1) the description of positions by the Cartesian coordinates of the end-effector;

2) the coordination of joint motions to achieve continuity in velocities and accelerations;

3) the specification of compliance in Cartesian coordinates.

AL

In 1974 a Stanford University group began the development of a second generation robot programming language based on ALGOL, AL, and incorporated in it advanced constructs for controlling multiple arms in parallel and cooperative tasks. AL was designed to support robot-level and task level specifications. AL runs on two computers. One is responsible for compiling the AL input into a lower level language that is then interpreted by the second real-time control computer. An interpreter for the AL language has been completed [1]. AL was designed to provide four major kinds of capabilities:

1) The manipulation capabilities provided by the WAVE system: Cartesian specification of motions, trajectory planning, and compliance.

2) The capabilities of a real-time language: concurrent execution of processes, synchronization, and conditions.

3) The data and control structures of an ALGOL like language, including data types for geometric calculations.

4) Support for world modeling, especially the AFFIXMENT mechanism for modeling attachments between frames including temporary ones such as those formed by grasping.

AL was the first robot language to be a sophisticated computer language as well as a robot control language. AL has been a significant influence on most later robot languages [2].

AML

AML is the robot language used primarily in IBM's robot products like the RS-1 [3]. The C programming language is used for implementing AML. C's portability allows AML to run on various models of IBM computers, and on non-IBM computers based on the Motorola MC-68000 processor family processor [4]. The design philosophy of AML is to provide a system environment where different robot programming interfaces may be built [5]. For example, extended guiding and vision interfaces can be programmed within the AML language itself.

The language supports operations on data aggregates, which can be used to implement operations on vectors, rotations, and coordinate frames. It also supports joint-space trajectory planning subject to position and velocity constraints, absolute and relative motions, and sensor monitoring that can interrupt motions. AML allows the user to design high-level commands by

providing a powerful base language. The AML environment includes motion checking and simulation for robot movements.

VAL II

VAL II [6] is a computer-based control system and language designed for the Unimation Industrial robots. It is an enhanced version of the VAL robot programming language released by Unimation in 1979 for its PUMA series industrial robots. VAL II provides the capability to easily define the task a robot is to perform, where the tasks are defined by user-written programs. Other benefits include the ability to respond to information from sensor systems such as machine vision, improved performance in terms of arm trajectory generation, and working in unpredictable situations or using multiple reference frames. The basic capabilities of the VAL language are as follows:

1) Point-to-point, joint-interpolated, and Cartesian motions (including approach and deproach motions);
2) Specification and manipulation of Cartesian coordinate frames, including the specification of locations relative to arbitrary frames;
3) Integer variables and arithmetic, conditional branching and procedures;
4) Setting and testing binary signal lines and the ability to monitor these lines and execute a procedure when an event is detected.

VAL's support of sensing is limited to binary signal lines. These lines can be used for signals and also for limited sensory interaction.

1.2.1.2 NC Machine Programming Languages

NC part programming is concerned with the planning and documentation of the sequence of processing steps to be performed on an NC machine. The planning portion of a part programming requires knowledge of machining as well as geometry and trigonometry. The sequence of processing steps in NC involves a series of movements of the processing head with respect to the machine table and work part [7].

A number of NC machine programming languages have been developed. The most common NC machine programming languages are Word Address and APT. APT is a higher level language while Word Address is a lower level language.

Word Address

Word Address was developed for use with NC machine tools and is still used by most modern CNC machine tools. Word Address syntax is very cryptic and has very little correlation to English. Hence the commands are very difficult to remember. An NC machine program in Word Address consists of a series of sequential statements or blocks specifying the cutter motion, and operations to be completed. A statement can be further divided into words. Each word in a statement consists of a character identifying the meaning or address of the word and a number representing its content. For example, the statement

N1G92X1Y2Z1.4;

consists of the words N1, G92, X1, Y2, Z1.4. The words N1 and G92 represent the sequence of number and absolute zero point, respectively. The words X1Y2Z1.4 specifies a tool position of x=1, y=2, and z=1.4.

APT

APT is the acronym for Automatically Programmed Tool. Initially developed in 1956 at MIT, it is the most popular part programming language and has gained worldwide acceptance.

APT is a high-level NC programming language, as such it simplified a great deal NC programming work. An NC cutter path can be defined by geometric entities specified by the APT geometric definition statements. The APT vocabulary contains many words. Programs written in APT have to be processed and reduced to Word Address programs before they can be used by an NC machine. The output from the APT processor is a description of the tool position and the desired sequence of operations, and is known as the CLDATA file. This file is further translated by a postprocessor into the Word Address codes for an NC machine.

1.2.1.3 Manufacturing Cell Control Programming Languages

CML is a cell control programming language that has been successfully implemented in industry. The Cell Management Language (CML) is a powerful computing environment for building

complex manufacturing systems. It was developed by Carnegie Mellon University and Westinghouse Electric Cooperation in 1983.

CML is based on the non decomposable primitives that can be combined into complex programs. Programming in CML is rule-based. Thus, programs are not sequential but, are goals oriented, allowing real-time decision making. The CML environment is specifically designed to allow control of cells that include multi-vendor machines. To cope with multi-vendor machines, the environment includes standardized tools for the programmer to construct interpreters for each machine. Interpreters are required for each machine's language and for data that is sent to and from the machine. The interpreters are based on natural language understanding mechanisms and relational database techniques [8,9].

Between 1984 and 1987, CML was used to build two flexible manufacturing cells. The first cell focused on machining operations. The second cell [10-13] manufacture preforms for turbine blades. Wright [14] reported that the system does effectively reduce set up times, increase machine utilization, and minimize the number of bad parts produced. In 1985 and 1986, Westinghouse took CML on as a commercial venture.

1.2.2 Selections of Object-Oriented Approach

Object-Oriented Programming (OOP) is defined by Pinson et al. [15] as "*programming implemented by sending messages to objects.*" Programs written in Object-Oriented languages involve the definition of objects which store both the data and procedures required for the processing to be performed, and manipulating the object data to produce a certain result.

The benefits of using OOP are to enhance modularity and flexibility and to reduce new code [16]. In the modeling of a manufacturing system, OOP language provides the capability to represent the system naturally. Thus, the software objects correspond to the elements of the real system. Intercommunication between objects can be partially represented by the methods defined in the object class. Using this scheme, a new element can be added easily to a certain model. The following section will address the detailed object-oriented concept and language, specifically Smalltalk.

1.2.2.1 Object-Oriented Terminology and Characteristics

A set of fundamental object-oriented concepts found in many object-oriented languages is summarized below. These concepts form the basis of the object-oriented approach to data and system modeling for the UniSet Environment.

In the object-oriented approach, any entity of an application is considered as an object. Every object has a unique object identifier. An object is defined by Booch [17] as "*An object has state, behavior and identity* . . . " The state includes all of an object's relevant properties. The properties may be either static or dynamic. Static properties usually remain constant over the life of an object. Dynamic properties can change regularly. The behavior of an object is the set of methods (procedures) which operate on the state of the object. The word *instance* is synonymous with object.

The state and behavior encapsulated in an object are accessed or invoked from outside only through explicit message passing.

A **class** groups all objects which share the same set of attributes and methods. Every object belongs to one and only one class as an instance of that class.

To describe an object and objects in object-oriented programming languages, four general terms are frequently employed: abstraction, encapsulation, inheritance, and polymorphism. These terms are described below.

Abstraction

Each object is unique. However, abstraction removes certain distinctions so commonalities between objects are recognized. Without abstraction, only differences between objects can be descended. With abstraction, it is easy to selectively omit certain distinguishing features of one or more objects, allowing one to concentrate on the features they share. Abstraction is defined as: " *the act or process of separating the inherent qualities or properties of something from the actual objects to which they belong*" [17].

Abstraction is one way to manage the complexity of the objects in the world. It allows the treatment of many individual phenomena with a single notion. Therefore, the result of abstraction is a class. In other words, abstraction is a relation between classes and objects. It specifies that each class can be instantiated by one or more objects. In addition it specifies that each object must be classified as one or more classes.

Encapsulation

Encapsulation is closely related to abstraction. It refers to the act of capturing the state and behavior of an object entirely within the object. An object should be a black box so that users do not have, and do not need access to the internal details or construction of an object. Good encapsulation ensures that none of the internal details of an object are available to the object's clients.

The combination of an adequate abstraction and good encapsulation makes for easily used and highly portable software. New objects can replace old objects regardless of internal details as long as the external view and responses are the same. Encapsulation also makes for safe software. Changes made to one class should not affect the behavior of other classes.

Inheritance

Inheritance is the ability of objects to obtain information about their internal states and behaviors from more abstract ancestors. Object-Oriented languages contain a mechanism that allows inheritance between classes. Inheritance involves a *superclass* and *subclass(es)*. Superclass is a term that refers to the next most abstract object in the ancestral hierarchy. Subclass is a term that refers to the class that inherits from the more abstract class. For instance, classes **Milling Machine**, **Lathe**, and **Drilling Machine** are subclass of class **NC machine**.

Inheritance promotes code reuse, acts as an organizational tool, promotes generalization, and eases program modification, extension and maintenance. Inheritance promotes code reuse because the structure and behavior of the superclasses are passed to subclass(es). Common behaviors and attributes should be coded into the superclass from which related classes can inherit. Instance acts

as an organizational tool since classes with the same superclass are usually closely related. Subclasses are usually specializations of superclasses.

Polymorphism

Polymorphism is the capability for different objects to respond to the same meaning of a message. Polymorphism allows the same message selector to be used by different classes for different methods. Polymorphism is also known as name overloading.

To understand how one word can have many meanings, one must understand the mechanism involved in executing a method. In Smalltalk a message is sent to the receiver accompanied by zero or more arguments. The receiver interprets the message based on the method defined in the receiver's class or any of its superclasses. If receivers are of different classes, the same message can invoke completely different methods.

When conscientiously applied, plymorphism reduces the number of messages a programmer must commit to memory. Polymorphism also allows code to be used on any object provided the object's class implements the appropriate method.

In Smalltalk polymorphism is very valuable, For example, when trying to run a machine the message **run** can be sent to instances of classes **Milling Machine**, **Lathe**, and **Drilling Machine**. Instances of these classes all respond to this message by running themselves in their own unique way. The methods for **run** in each class know the details of how to run the instances.

1.2.2.2 Implementation Language Selection - Smalltalk

Smalltalk was chosen as the implementation language for the UniSet and UniSet Environment. Smalltalk is the purest Object-Oriented Programming Language currently available [18]. It has many features that make it well suited to the development of the UniSet and UniSet Environment.

The main advantage of Smalltalk is a collection of built-in classes allowing for creation of complex applications simply by connecting existing pieces of the code. The base Smalltalk/v for Windows system comes with more than one hundred classes and two thousand predefined methods that are completely accessible to the user [15].

Smalltalk is referred to as an environment for rapid prototyping. Prototype systems can be developed quickly to check new ideas and concepts. Smalltalk not only provides a graphical user interfaces for program development, but also provides classes that the programmer can use as building blocks for developing application specific GUIs.

Smalltalk also provides a special window, called *Debugger* that allows for inspection of the objects and their variables at any time and for further validation of the code. The debugger can be used when the message **self halt** is encountered, when a user interrupt is requested, or when a runtime error appears.

The hardware platform that UniSet is implemented on is very important. The use of IBM PCs for control application is popular. Smalltalk, and thus UniSet, is implemented on a personal computer requiring a minimum of an 80486 processor. Smalltalk/v for Window operates under MS Windows. It is file compatible with Smalltalk/v Mac. Thus, Smalltalk and any of the Smalltalk applications are highly portable.

Smalltalk also provides complete class support for file access. Class FileStream provides a complete set of messages for writing and reading ASCII files.

The memory management is not of concern to a Smalltalk programmer. It provides its own memory management and purges the system of released objects so that memory does not become an object graveyard [19]. Smalltalk can manage up to 16 megabytes of memory.

Smalltalk is not particularly well suited for mathematically intensive operations and high speed communications. However, Smalltalk can be linked to primitives and programs written in 'C' [20]. 'C' is well suited to such applications. Thus, the power of 'C' can be combined with Smalltalk's reliability and the advantages of OOP.

1.3 Thesis Overview

Ideally, programming of an FMC would take place on one computer in one language with one software environment. Difficulty in programming cells would be resolved by developing a Unified Instruction Set (UniSet) which provides consistency independent of machine languages. By introducing the Task level UniSet, a user's job in programming cells is reduced. The overall objective of this thesis is to develop UniSet and Task level UniSet as a manufacturing programming language and to produce a cell operation environment that can be used by cell operators as a computer integrated manufacturing tool.

This chapter presented the definition of the problem based on the current status of implementation of FMCs, and described some of the back ground knowledge associated with development of objectives of this thesis.

Chapter 2 elaborates on the past and current research works in the fields of automation languages and environments, artificial applications in manufacturing, cell operations, computer aided process planning, object-oriented technique applications in manufacturing, and simulation in manufacturing.

Chapter 3 outlines the philosophy and history of UniSet, and the operation of an FMC under UniSet.

Chapter 4 describes the instruction set of UniSet, based on the results of a detailed survey and comparison of VAL II, ASEA, APT, and Word Address programming languages.

Chapter 5 presents the development methodology of the Task level UniSet which is based on tasks performed by the cell components.

The detailed structure of the UniSet Environment is addressed in Chapter 6. The Object Data Models involved in a CIM are introduced. The methodology for design of the UniSet Environment is also addressed in this chapter.

Chapter 7 outlines the design of the classes and relationships between them based on the results of analysis for the Set up Module. This module consists of four sub-modules which contribute to building the Data Object Models.

In Chapter 8, the Program Development Module that is used to create a cell program is outlined, together with the methodology of the automatic generation of native machine programs.

Chapter 9 addresses the Task Initiation Diagram (TID) development module, and the Simulation and Control modules. Also, the design of classes and objects involved in these modules is outlined. Conclusions and recommendations for future work are provided in Chapter 10.

Appendix A has been included to present a precis to the basic notations of the Object Modeling Technique (OMT) and to serve as a reference while reading the thesis. A transformation of the orientation from the Euler angles to the Quaternion is given in Appendix B. The comparison between codes generated from the system and codes developed by a user is presented in Appendix C.

CHAPTER 2
LITERATURE SURVEY

This chapter outlines the most prominent developments in six areas of research that are related to the UniSet development. These six areas are: automation languages, applications of artificial intelligence in manufacturing, cell operation, object-oriented concepts and applications in manufacturing, computer-aided process planning, and simulation in manufacturing.

2.1 Automation Languages and Systems

Most existing automation languages are reviewed in the previous chapter. Thus, this literature survey focuses on research for developing or utilizing environments that generate robot or NC machine programs with existing languages. According to the application domain, automation languages and systems are classified into three categories: robot, NC machine, and cell programming languages and environments.

2.1.1 Robot Programming Languages and Systems

The functionality of existing robot simulation systems such as GRASP and RIPE are surveyed here. Bonney et al. [21] and Wright et al. [22] discuss how the GRASP robot simulation

system is used for off-line programming. The GRASP facilities of interactive geometric modeling, configuration control of robots, path control and high level programming, and time coordinated activities can be used to plan and evaluate proposed robot workplaces, and for case studies. Miller et al. [23] introduce the Robot Independent Programming Environment RIPE which was developed at Sandia National Laboratories. According to the authors, the software environment developed can facilitate the rapid design and implementation of complex robot systems to support various applications.

Developments in the area of task level programming languages and related environments dealing with only robots have been reported in the literature [63, 24]. Gini et al. [25] describe an approach to robot programming that supports task-oriented specifications for manipulation activities. Plan formation and program generation techniques have been proposed for use to transform task specifications into executable programs. Mettala et al. [26] established a natural language, plan generating, artificial intelligence environment with the translation facilities to obtain robot control character strings for the IBM 7545 manipulator. The communications and functions which are necessary for the robot to perform the basic pickup, move, and putdown operations, are developed in a cohesive system. The plan generating techniques of STRIPS are used for the implementation of these operations. However, since all the systems are developed for specific robots and languages, practical implementations which generally involve a variety of robots and languages would require completely new development work.

A system for quickly generating a path among polyhedral objects is proposed by Tseng, et al [27]. This trajectory planning system is designed to provide a robot manipulator with a collision-free path among these objects, but this system requires a complete information about the robot

configuration and involved objects.

2.1.2 NC Machine Programming Languages and Systems

The development of systems for generating a part program has been discussed in the literature. Most of the approaches in this direction propose the use of artificial intelligent techniques such as knowledge-based systems [28, 29], or of CAD/CAM [30, 31] approaches. Since the UniSet Environment includes the facilities for generating part programs, two systems, AUTO-CNC-PP and NC-Lathe, that have been developed successfully are studied. El-Midany, et al. [30] developed the AUTO-CNC-PP system which generates automatic NC programs for complicated milling components. The system is designed to provide several advantages through the integration of AUTOCAD files and dBASE IV. According to authors, the advantages of this system are: entry errors are impossible, and processing and working times are reduced.

A simulator, entitled NC-Lathe, is developed by Zhang et al. [31], to simulate the NC lathe operation, and to train the part programming technique. NC-Lathe includes five modules. These are: the operation selection module, the part program development module, the machine set up module, the simulation animation module, and the design considerations module.

2.1.3 Cell Programming Languages and Environments

With the exception of the work of Bourne et al. [10], systems which deal with cell programming languages and environments reported in this literature survey have focused on developing programming systems for experienced users, not for operators [32, 35]. Bourne et al.

[11] show how a number of linguistic devices can be used to eradicate numerous ways of describing operations from the action oriented descriptions. The resulting language is a formal variant of a natural language with a Lisp-like syntax.

In order to study functionalities of developed packages, a number of researches are surveyed. Volz et al. [33] discuss how Ada can be used to program a robot-based manufacturing cell as an example of a real-time embedded system. Also, the computing issues in manufacturing cells are discussed with respect to Ada. The principal advantages and difficulties in using Ada for programming robot-based manufacturing cells are summarized based on the software issues, and a case study is demonstrated for verifying these observations. Levas, et al. [34] present an informal characterization of the process of workcell application design, and described the architecture and implementation of a software environment, WADE, intended to aid designers of workcell applications. Its design is object-oriented and open-ended, which is suitable for easy experimentation, extension, and refinement. According to the above systems, the suggested tool in designing an environment is an object-oriented approach which provides a simple to use and intuitive user interface, and allows for an easy development of a system model.

Some methodologies and software architectures of a flexible manufacturing cell are suggested by Matsuda et al. [36], Rankey [37], and Caseli [38]. Matsuda proposes design concepts, functions, and software architecture for autonomous manufacturing cells for the automatic production of mechanical products with high quality. Autonomous manufacturing cells can be the most important component of an autonomous decentralized manufacturing system. According to the authors, an autonomous manufacturing cell requires three basic functions: (1) selecting an executable job, (2) executing the job, and (3) reporting the job results. Rankey [37] examines an

overview of a generic, structured methodology and architecture primarily in order to facilitate the study of existing manufacturing systems (including metal cutting, electronic and electro-mechanical assembly, robotised welding and automobile assembly, etc.) and to aid the design of new flexible manufacturing cells and systems with a CIME environment. Caselli et al. [38] discuss the integration of structure, function and knowledge in manufacturing workcell modeling, simulation and design. After an overview of applications of semantic and object-oriented data models in the manufacturing domain, issues relating to the control synthesis for manufacturing workcells are presented. In particular, a data model encompassing functional and control features, along with application domains structural knowledge is developed. This model assists in explicitly representing the control aspects of engineering design within an object-oriented database and supports a task-level, functionality-driven, manufacturing workcell design.

2.2 Artificial Intelligence Applications in Manufacturing

The majority of AI applications in manufacturing rely on symbolic knowledge representation and processing (i.e., expert systems, knowledge-based systems). The alternative approach of Neural Networks also deserves attention. However, solutions to most manufacturing problems involve not only heuristic knowledge, but also mathematical calculations, optimization techniques [130-143], and man-machine interfaces.

Expert system is one of the most successful implementations of AI in manufacturing environments. By definition, an expert system is a computer program that simulates the thought

process of a human expert to solve complex decision problems in a specific domain. Computer integration of CAD, CAPP and CAM (Computer Integrated Manufacturing (CIM)), is a very serious problem for industry. Expert systems are employed as one solution for dealing with problems of CIM. A number of research works have been devoted to it [39-43]. The operation of manufacturing systems requires intensive knowledge. How to represent this knowledge is a major issue. According to developed methods by AI researchers, knowledge representations are accomplished by rules, logic, or frames. A number of research works for developing system installed knowledge bases have been reported in the literature [49-59].

A review of the literature also reveals that expert systems have been developed successfully for several other manufacturing domains, such as diagnosis, scheduling, process control, and automatic program generation . . . etc., [44-48].

Cathar et al. [45] suggest issues in building a conceptual modeling approach that can have sufficient semantic power to represent the complexities of decision making in a CIM system. To express the information in a conceptual model they employ a hybrid methodology that integrates the concepts of object-oriented programming, including message passing semantics, and temporal logic.

The expert system described by Raggenbass et al. [47] generate from the CAD data of a part a complete numerically controlled (NC) manufacturing plan, which contains only punching operations, only laser operations or a punching-laser combination. The main points in the fully automatic NC program generation are the recognition of the sheet part geometry, the determination of the manufacturing variant, the selection of the optimum punching tools, taking account of

violation criteria, the arrangement of many parts on a plate, and the optimization of tool movement paths.

Neural networks are used for dealing with the difficulty in choosing recommended machining parameters in the UniSet Environment. These networks consist of simple processing elements or nodes capable of processing information in response to external inputs. The neural network applications and perspectives in Manufacturing have been surveyed in the literature [60-63]. Wang [60] presents a neural network approach to multiple-objective optimization of cutting parameters. First. the problem of determining the optimum machining parameters is formulated as a multiple-objective optimization problem. Then neural networks are proposed to represent manufacturers' preference structure.

2.3 Operations of Flexible Manufacturing Cells

Several approaches have been developed to deal with the problems of scheduling and controlling flexible manufacturing cells and systems. The majority of these approaches focus on developing methods of modeling cells and systems, for example, petri net, extended petri net, or timed petri net [64-74]. Jockoivc et al. [69] consider the highest control level organization of a Flexible Manufacturing Cell (FMC) by using petri nets. The net is decomposed into individual robot nets and machine nets. A derivative petri net, called Updated Petri Net, is developed by Harhalakis et al. [74]. It facilitates the modeling of relationships between operations of various related application systems, CIM database updates and retrievals. These approaches allow the behavior of

the system to be represented as event graphs for which rules governing operations of the system can be derived and verified.

Other approaches utilizing knowledge base and concepts of artificial intelligence have been proposed in the literature [75-96]. The main interest of knowledge-based approach is to achieve the goal of unmanned manufacturing cells. The basic functional design of a robot or machine tool controller which is suitable for the unmanned manufacturing cell is discussed by Fussell et al. [84]. Chen et al. [95] examine a class of flexible manufacturing cells (FMC) containing a robot. The role of the robot in a cell is to load parts onto machines, to unload parts from machines, and to transport parts between machines. Since the productivity of an FMC is directly proportional to the level of productive work performed by the robot, the manner in which robots move between machines affects productivity. The problem of finding efficient robot schedules/tours is therefore one of substantial economic significance in the operation of an FMC. The authors devise a rule-based system to assist the cell supervisor in making good decisions by dynamically coordinating the available information during the production process.

The hierarchical control structure is by far the most commonly used architecture. This architecture contains a number of control entities arranged in a pyramidal structure. Generally, the lowest levels are dedicated to tasks such as machine drive control and position sensing. The top levels control, coordinate and manage the entire system. UniSet employs this architecture with two levels: the cell controller and the machine controllers. O'Grady et al. [97] describe two major hierarchical control structures (the advanced Factory Management System AFMS and the Automated Manufacturing Research Facility (AMRF)). The advantages and difficulties of both types of manufacturing cells are explained in his work.

In addition to the hierarchical control architecture, the distributed control structure is suggested by many researchers. This structure is characterized as one where there is no global information describing the state of the system, all information is being distributed amongst the subsystem which has their decision on local factors. UniSet also employs this control structure by using the Virtual Device (VD) concept. Many available DNC systems provide features which are not used in a shop that has only a few NC machine tools. Several of these NC systems also require that special hardware, a particular computer system and software, and expensive options be purchased and installed on a modern controller for each machine. In order to solve these problems, a low cost alternative has been developed at the National Research Council of Canada [98] which requires minimal hardware upgrading. Emphasis has been placed on offering a decentralized set of services for the machine operator rather than a centralized control structure. Basnet et al. [99] propose a parallel-processing system for locating parts and controlling an industrial robot. The system employs Transputers and Ocean to achieve parallelism. Transputers communicate with one another by means of point-to-point communication links. Ocean is usded to program these transputers. A new algorithm for obtaining the coordinates of the parts using the sensed vibration and deflection signals is described. The algorithm dispenses with the lengthy and complex equation-solving procedures previously required.

Most research works in the literature dealing with cell monitoring are concerned with the development of systems dealing with the detection of failures and unexpected events. These works involve several approaches such as modeling techniques, or expert systems [100-113].

Having summarized the principal requirements for comprehensive supervision of machine tools and manufacturing cells, the concept of a hierarchical monitoring and diagnostic system for

manufacturing cells is introduced by Monostori [106]. Pun et al. [108] present two correlated intelligent processes: an Intelligent FMS monitoring process, which is to be used by a human monitoring operator to manage FMS planning and control, and an Intelligent learning-aid process to help the monitor to learn and to improve his skill in monitoring. This paper consists of three parts: (a) clarification of the meaning of an intelligent process, since its double utilization in this paper may appear pretentious and ambiguous; (b) the intelligent FMS monitoring process; (c) the intelligent learning-aid process. Sturges [111] discuss an approach to monitoring cutting tool/part interactions in milling processes using processed acoustic emission (AE) signals and the combined geometric aspects of the tool and the part. The expected events marking interactions of the tool edge with the part during a cut are computed from a priori knowledge of tool and part geometry.

2.4 Object-Oriented Concept and Applications in Manufacturing

The design methodology of the UniSet is an object-oriented paradigm. Numerous researches works on object-oriented modeling techniques, Smalltalk applications, and Object-Oriented Programming (OOP) applications in manufacturing have been sighted. The terms "Object Oriented Programming" and "Object Oriented Language" are used in several conflicting ways in the literature. The concept and application techniques of the OOP have been discussed in the article [114-118, 126]. These works demonstrate that OOP is an appropriate language for the complex software systems, such as user interfaces.

Since the implementation language in UniSet is Smalltalk, emphasis was placed on applications developed in Smalltalk in the literature search. Some studies are far from the manufacturing domains [119, 120], but the purpose of the survey here is to take advantage of the programming techniques developed. The main disadvantage of Smalltalk is a lack of the computational capability. In order to solve this problem, Thomas et al. [20] study a pragmatic approach which they call "Smalltalk + C," that allows each language to be used where appropriate. Smalltalk, for example, is used to represent and manipulate high-level information while C is used to implement small, time/resource critical low-level facilities - an approach utilized in the UniSet Environment and found to be effective.

A number of research works on object model development in manufacturing have been reported in the literature [121-129]. Onosato et al. [124] describe an experience of building a virtual factory using solid modeling and object-oriented programming techniques. In a virtual factory, each element such as a machine, a workpiece, and a container, is modeled as an object in Eslips, an object-oriented programming language with solid modeling function. Attributes and behaviors of each element are described as internal variables and methods of an object. Spatial factors of factory elements, for example, shapes, positions, and distances, are described in, or extracted from, solid models of the elements. The combination of object-oriented programming and solid modeling improves both the preciseness of a factory model and the efficiency of implementation.

Joannis et al. [129] propose an object-oriented approach to the specification of manufacturing systems. The specification model is built around a set of concurrent cooperating objects whose behaviors are described using communicating finite state machines. According to the

above works, one of the interesting features of these approach is that the system's environment can be included in the object model.

2.5 Computer-Aided Process Planning (CAPP)

Literature surveys indicate that the variant and generative approaches are the two main ones used for automating the process planning function.

(1) Variant process planning

Group technology is the basis for automating the process planning task by a variant method by which the components are grouped into families according to their manufacturing characteristics. A standard process plan for each family of parts is established and stored in the computer database with reference to its family code. Process plans are produced either by retrieving the standard plan for a particular family or by modifying the standard plan to suit the new component if it differs slightly from the standard one. This technique is unsuitable if there exist a large number of families of parts, and parts that do not strictly fit in them.

(2) Generative process planning

A generative process planning system creates an optimum process plan automatically from first principles using knowledge encoded in the software by analyzing the part geometry and other factors that influence the manufacturing process.

Since UniSet selects generative approach for process planning, literature surveys here are focused on the generative approach. Process planning tasks require high expertise which are based on experience.

AI approaches offer methodology to represent informal experiences as formal formats. Majority works focused on knowledge representation and reasoning mechanism [144-163]. A number of expert systems developed include facilities for the tool management which are associated the UniSet Environment. An Expert system developed by Ssemakula et al. [156] employ the important factors that influence process planning sequence optimization, such part geometry, available machine and cutting tools, and cutting forces, etc. Maropoulos [165] presents a new cutting tool selection methodology, namely the Intelligent Tool Selection (ITS), which covers the whole spectrum of tool specification and usage in machining environments. The selection process has five distinct levels and starts by deriving a local optimum solution at the process planning level, which is progressively optimized in the wider context of the shop floor. The selection method allows the implantation of the minimal storage tooling (MST) concept, by linking the ordering of new and replacement tools to production control. ITS also uses the concept of tool resources structure (TRS), which specifies all tooling resources required for producing a component. By using the framework provided by ITS, TRS and MST it can be shown that tooling technology interfaces with diverse company functions from design and process planning to material scheduling and tool management. The selection methodology results in higher utilization of tools, improved efficiency of machining processes and reduced tool inventory.

Torvinen et al. [164] describe the CAD/CAM system which has been developed at Tampere University of Technology. The tool management system, as well as the other self-programmed

applications, is implemented in a PC environment under MS-DOS version 3.3 or higher. The program system described in their work is built around three types of databases: the CAD/CAM database, application-specific databases (e.g., the tool database), and common databases (including the planning rule database connecting manufacturability data to the design process).

One of the programming languages in the Task level UniSet, is partly derived from machinable feature-based descriptions. Thus, in order to generate executable programs, process planning should be required, such as tool selection, NC code generation, etc. Several artificial intelligent and object oriented approaches have been applied to feature based planning [168-171]. In view of the results so far achieved, this AI approach is not successful, but has potential. Juri, et al. [166] investigate the use of a feature-based approach for modeling engineering parts to support the product data requirements of an intelligent planning system, employing the frame-based and rule-based knowledge representation techniques from artificial intelligence research. But the approach has been applied to only the domain of rotational parts. Another expert system for processing machined components using part features is discussed by Subbarao [167]. The system developed can select the processes, machine tools, cutting tools and various cutting parameters based on the attributes of the part features. But it is implemented for a limited number of features for rotational parts using a commercially available expert system shell.

2.6 Simulation in Manufacturing

Simulation is one of the most important tools for verification of a system. UniSet environment includes the simulation module. Thus, simulation environment design techniques are focused on in this section.

The current state of the art in the simulation of manufacturing systems is addressed by Shannon [172]. The definition of simulation, and its advantages, uses, potential causes of failure, present trends, use of graphics, and required skills, are also discussed by Shannon. Some other important characteristics, such as visual interactive model development, visualizations of the simulation output, user-model interaction, and intelligent advising, etc., are discussed by Vujosevic [173]. Object-Oriented languages seem to be suitable for simulation languages. In order to prove this fact, a comparison between traditional factorial programming and object oriented programming has been done by Atabakhsh et al. [174] Developments of Discrete event simulation systems using an object-oriented approach have been discussed in the literature [175-179]. One of the simulation environments developed using Smalltalk is SmartSim, developed by Ulgen et al. [177]. It can be used by manufacturing engineers as a computer aided design tool for designing manufacturing systems. SmartSim has been found to increase the ease with which simulation models can be constructed and modified. It is able to accomplish this due to the synergy of an icon-based user interface, and availability of interactive animation. This environment however, does not have decision support capabilities.

CHAPTER 3
PHILOSOPHY AND HISTORY OF UniSet

This chapter outlines the philosophy and development history of the UniSet and UniSet Environment. The anatomy of a Flexible Manufacturing Cell operating under UniSet is also described.

3.1 Philosophy of UniSet

An FMC may be described as a computer-integrated group of multiple NC machines or workstations which would be linked by material handling devices (robot, conveyor, AGV, etc.) for the automatic processing of different products parts, or the assembly of parts into different units.

Most of the machines comprising the cell are typically from different manufactures with different communication hardware standard, protocols, and programming languages. This situation can cause problems for system integration engineers, programmers, and operators. It requires that they be experts in dealing with each machine in order to manipulate the manufacturing cell. As the variation of machines in a factory increases, more skilled workers are required, and the

complexity and time to setup and produce lengthen. This results in an increase in the cost of production due to automation, which is the reverse of its philosophy.

In order to deal with these problems, many research groups have created universal or standard programming languages or protocols for the multi-vendor equipments in a manufacturing cell. As examples are the Manufacturing Automation Protocol (MAP) which was developed by a GM research group to deal with standardization of communication protocols between the computational components in the manufacturing environment, and the standard for the global programming language.

The general philosophy adopted in these, and similar works, was to create a standard for machine producers to adhere to. The vast majority of these standards however, have not been generally adopted by machine tool manufacturers. These manufacturers prefer to use proprietary protocols and languages, or super sets of the standards. Translators to and from the standards are generally provided.

The UniSet approach has a totally different philosophy for dealing with the problem of multi-vendor equipment in an FMC. Studies conducted early in the project showed that similar types of machines from different machine tool manufacturers, as well as different types of machines (i.e., lathes, robots, etc.) have great deal of commonalities in their abstract functionalities. The differences between the command languages for the same abstract function are mainly lexical. This commonality between the abstract functions forms the first basis of UniSet. UniSet is not considered a standard since manufacturers would not have to adhere to it in the design of their machines.

UniSet allows programmers to concentrate on only one instruction set rather than numerous machine programming languages. Furthermore, it has the capability of representing all the possible activities of the cell and of coordinating these activities (e.g., start machining etc.). A translation facility translates the instructions of UniSet into target machines native codes. Thus, implicitly, a manufacturing cell can be considered to be a single machine with a language, called UniSet, and programmed to describe how the cell components must be used to manipulate the raw material.

Another main advantage of UniSet is that it is nonexclusive. That is to say, a programmer may elect to program a cell component in its native language rather than UniSet. This capability allows the direct inclusion of programs generated by a CAD/CAM system for tool paths in the cell program.

In the context of OOP, UniSet instructions are polymorphic. Each instruction is translated into a basically similar function for a different machine. A cell program in the UniSet environment is developed in a serial fashion. The user does not need to develop a complete program for a device or a machine in the cell as a contiguous set of instructions as will be seen later in Section 3.3.2.

In order to eliminate the programming difficulty, the cell environment has been enhanced by the addition of Task level UniSet. Task level UniSet allows a cell developer to hide the detailed commands of a cell task from the user. The user requests the cell to perform a task out of a list, or combine tasks to form a complete cell operation cycle. During compilation the environment decomposes the tasks into basic level UniSet commands and coordinates task interaction.

3.2 History of Development of UniSet

Research work on UniSet and the UniSet Environment started at the University of Ottawa in 1988 to deal with the problems of integrating multi-vendor equipment in an FMC. The initial work is documented in personal notes of Fahim [191]. Since then a number of individuals have contributed toward the evolution of UniSet, and some of the work have been published [182, 183].

Taylor [180] reported that a unified instruction set for both robot and NC machines could be possible after a comparison study of two popular programming languages, VAL II and APT. He also created the rudiments of instructions of the unified instruction set using the major and minor word concepts originally implemented in APT.

In 1990, Szukalo [181] contributed to the UniSet research by investigating and establishing some central core set of commands to be used with UniSet. His research effort did not include the translation system from the UniSet commands to machines native codes.

A second generation unified instruction set and a broad outline of a UniSet Environment was developed by Tolkamp [182,192]. The emphasis in this work was to develop a coherent grammar for the UniSet syntax and a translation facility, and to demonstrate the use of OOP for developing such a system. In his thesis work Tolkamp demonstrated a rudimentary, yet functional, UniSet Environment with a limited instruction set and a translation facility.

The current work represents the third generation UniSet and UniSet Environment. In this work, a complete set of basic instructions, the concept of the Task Level UniSet, and a fully

functional environment with all its components has been developed. Furthermore, the author documents the philosophies of using OOP concepts for developing such a manufacturing system.

3.3 Anatomy of a Flexible Manufacturing Cell under UniSet

This section describes typical cell constituents and their interaction under UniSet. The concepts of cell programming using the UniSet Environment is addressed, and the suggested control structure is outlined.

3.3.1 Machines Forming the Cell

As illustrated in Figure 3.1 an FMC may consist of NC machines, robots, conveyors, and sensory devices (i.e., simple sensors and vision systems). In order to perform given tasks in an FMC, these constituents have to operate harmoniously under the overall command of a cell controller.

The material processing operations in an FMC must be automated, programmable, and easily alterable. From this perspective, CNC turning centres and CNC machining centres comprise the majority of the processing equipment in an FMC. These machines are not only capable of being easily reprogrammed, but are also capable of accommodating a variety of tooling via a tool changer and tool storage system. It is common for a CNC machining centre to contain 60 or more tools (mills, drills, boring, etc.), and for CNC turning centres to contain 12 or more tools (right-hand turning tools, left-hand turning tools, boring bars, drills, etc.) [184].

The use of industrial robots is becoming an established practice in manufacturing. Although the possible tasks performed by robots are varied, their major function in an FMC is grasping a part, transporting it to another location and release, i.e., material handling. Robots are also frequently used for loading and unloading tools. Since robots can be reprogrammed and retooled relatively quickly and easily, they are the most attractive material handling equipment in FMCs dealing with small batch jobs.

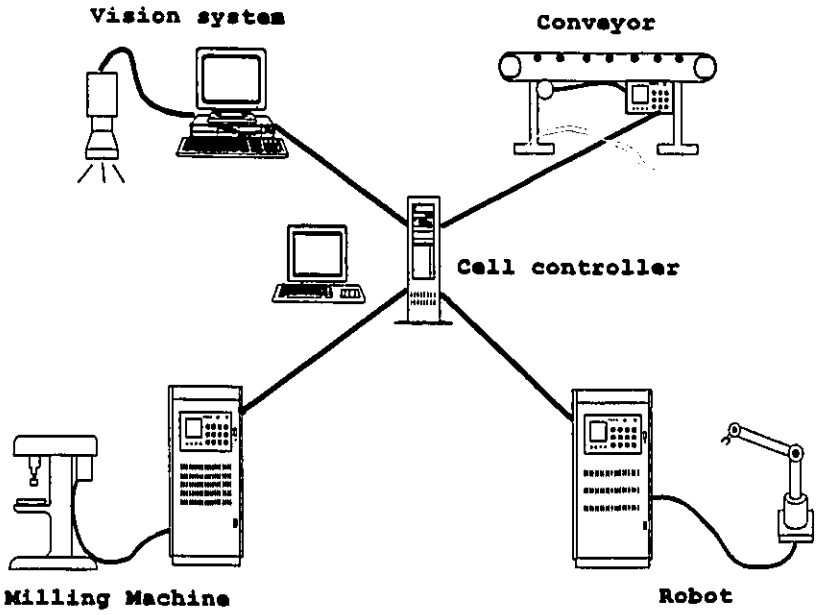


Figure 3.1: Typical components and communication in an FMC

A gripper or an end effector is defined as the special device that attaches to the manipulator's wrist to enable the robot to accomplish a specific task. Because of the wide variations in tasks that are performed by robots, the gripper must usually be customized for a specified job. In some cases a gripper changer may be provided to allow the robot system to adapt to different gripping requirements.

A conveyor system is used when materials must be moved in relatively large quantities between specific locations over a fixed path. Most conveyor systems are powered to move the materials along the pathways, other use gravity to cause the load to travel from one elevation in the system to a lower one.

In an FMC, conveyors are usually used in conjunction with some other material handling devices, such as robots or automatic loaders, and are generally short in length. A component is placed onto the conveyor at one end and is routed directly to its destination. In an FMC the conveyor must be controlled by the cell controller.

Sensing is one of the most essential aspects of an FMC. The main reason of using sensors in a manufacturing cell is to monitor the system to detect errors, and to obtain data such as part location, orientation, etc., which can be used by the cell controller, or a cell constituent. Currently used sensors in manufacturing cells can be categorized as simple sensors (i.e., limit switches, etc.) and programmable sensors (i.e., vision systems). Limit switches can be mechanical or optical based. They have proven to be reliable and easy to interface in a manufacturing cell. Their functionality includes the detection of the movement of parts or part carries in an environment. These sensors can be interfaced to either a specific component or directly to the cell controller.

A vision system on the other hand is typically a programmable sensor. Vision systems are currently used in cells to perform, among other things, the important tasks of part identification, part location and orientation, and part inspection.

3.3.2 Concept of Cell Programming

The general structure of an FMC program is that of interleaved device program fragments synchronized by appropriate sensor signals. Some of these program fragments may be concurrent. In UniSet, a cell program may be developed in a serial fashion with the program fragments threaded together by the synchronizing signals. This approach of serial code fragment development is both natural and intuitive, and greatly simplifies FMC programming.

In UniSet the cell program may be developed in mixed languages (Task level UniSet, UniSet, and native languages of the cell constituents). This approach would take advantages of unique features of a machine or a programmer's valuable experience in any particular languages.

In order to control cell operation using a UniSet program, it must be transformed into a format that a particular machine can use. As illustrated in Figure 3.2, the program segments for each individual machine are generated and translated separately. The translation and the generation facilities are imbedded in the UniSet Environment.

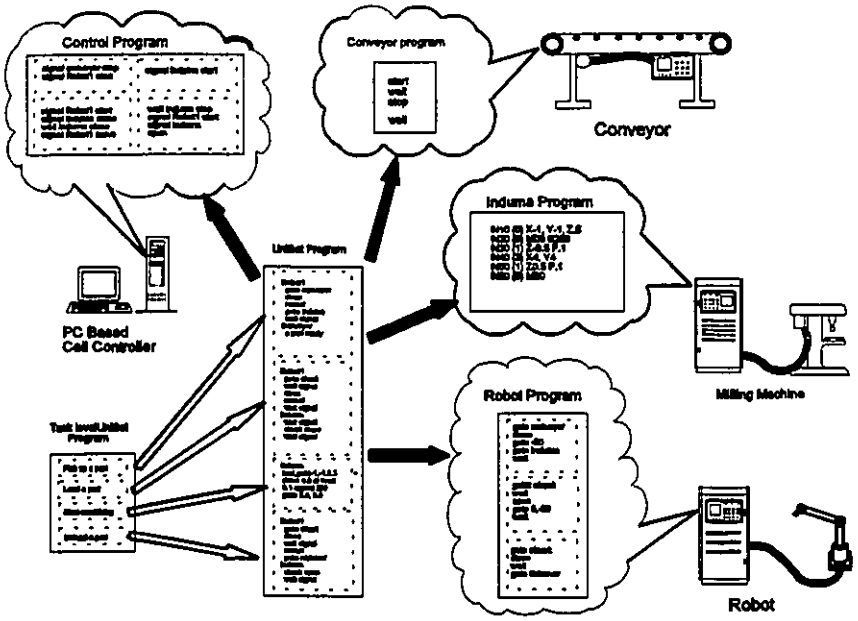


Figure 3.2: Generation and Translation Mechanism

3.3.3 Control Architecture under UniSet

In order to operate a cell under UniSet, the cell constituents have to be connected in a star network with the hub being a dedicated computer, i.e., cell controller (see Figure 3.1). The actions of each device are controlled by its own machine-level controller. The machine-level controllers are managed by the single cell controller. This structure represents a typical hierarchical control architecture.

Each machine constituting the cell is represented by a Virtual Device (VD). A VD is considered to be a shadow machine that has a structure mimicking the physical machine in abstraction. A VD can be decomposed into a main component and a number of other components

which interact or interface with other VDs in the cell. The VDs are used as holders of information and states concerning the respective machine components, and also as communicators with the physical machines. All VDs are portrayed as objects in UniSet, and can interact together concurrently using the multitasking capabilities of Smalltalk. This VD approach allows the cell controller to be easily configurable when a machine or device is added or removed from the cell. Due to the use of the VD concept, the logical control structure of UniSet is that of a distributed control architecture.

The representation scheme of cell activity as a decision logic structure is essential to control manufacturing cells. Most of the approaches that have been developed so far are adaptations of ones used in digital logic or electric circuits. In these approaches the cell or machine state is primary. The state of the cell or one of its constituents changes as result of the firing of an event. This approach is not suitable for control or verification of FMCs and FMSs. For such systems the task to be performed rather than the state must be regarded as primary. A task is executed when a particular set of conditions is met.

The methodology developed for UniSet regards the tasks to be performed by the cell or any of its constituent machines for being primary. Sensory signals indicating the change of state of machines are used to trigger or initiate tasks. Tasks may be simple and require a relatively short time to execute, i.e., "conveyor carrying a part," or may be complex and lengthy, i.e., "robot unloading a part." The methodology is depicted by a set of diagrams, referred to as Task Initiation Diagrams (TID), and their accompanying rules.

CHAPTER 4
Unified Instruction Set: UniSet

Although all concerned (user, developer, and NC machines producers) agree that a standard automation language and communication protocol have to be developed and adhered to, they have agreed to disagree on what those standards should be. The prevailing fact is that by the time consensus is achieved, the technology has progressed enough that the standard has to be revised. In spite of the **ISO** efforts, currently each manufacturer of NC machine/Robot develops their own dedicated control language. As mentioned in Chapter 3, preliminary studies of automation languages carried out by Taylor [180], and Szukalo [181] concludes that automation languages contain many commands that are similar in functionality.

UniSet has been developed by Tolkamp [182] as a manufacturing instruction set based on a comparison of machine tool and robot programming primitives. UniSet commands adhere to a consistent format with quasi-English syntax. This should allow the instruction to be easily understood and committed to memory. Most UniSet commands are polymorphic, that is to say, they represent a similar function for different machines. But the current UniSet version provides primitive instructions, and is not enough to program enhanced cell operations, i.e., conditional branches, etc. In this section the updated UniSet will be examined.

4.1 UniSet Instruction Format

There are six formats for UniSet instructions. These are shown in Table 4.1. The first instruction format is **UnaryCommand** and consists of only one word. **UnaryCommands** represent functions that are repeated with no variation. For instance to open a robot gripper involves only the instruction **Open** without any arguments.

The second instruction format is known as **Command**. A **Command** is a single word accompanied by arguments. The command word is separated from the arguments by a full colon(:). The colon serves as a syntax delimiter.

Table 4.1 UniSet Instruction Format

Class	Format	Example
UnaryCommand	command	stop
Command	command: <arg>	go:(1@2@3)
ModifiedUnaryCommand	command, modifier	Mist, off
UnaryModifiedCommand	command, modifier:<arg>	go,rapid:(1@2@3)
KeySymbol	command, <symbol>:<arg>	point, P1: (1@2@3)
BinaryModifiedCommand	command, modifier, modifier:<arg>	go,left,tool:(1@2@3)

The third instruction format is **ModifiedUnaryCommand** which consists of two parts, the first being the root instruction which is similar to a major word in APT. The second part of the instruction is a modifier. Modifiers serve as qualifiers for the root instruction.

The fourth instruction format, **UnaryModifiedCommand**, consists of a command followed by a modifier with arguments.

The fifth instruction format is **KeySymbol**. These commands are used to define and name the geometry description. The second part of the instruction is a symbol. The arguments for the symbol follow that symbol.

The last instruction format is the **BinaryModifiedCommand**. This format allows the addition of a modifier to the **UnaryModifiedCommand** instruction. The comma is used to separate terms, serves as the syntax delimiter.

There are several symbols used in the Syntax of UniSet. Braces, {}, identify an expression which should occur at least once. Brackets, [], identify optional expression. The vertical bars, |, separate alternative terms. Parentheses, (), group alternative terms.

The detailed representation for comparing the primitives and defining UniSet commands is presented in the following sections.

4.2 Motion Commands

Motion commands are used to describe the motion of a robot end-effector or the tool of the NC machine. They are by far the most complex commands in an NC machine or robot language vocabulary. The commands are classified into three categories to perform three kinds of basic motions: Joint Interpolation motion, Linear Interpolation motion, and Circular Interpolation motion. For the first two motions, there are two ways that robots and NC machines move with respect to their coordinate system; absolute and relative motion. In an absolute motion, all desired locations

of robots or NC machines are obtained from one fixed reference point. On the other hand, a relative motion specifies the location in relation to the previous one.

4.2.1 Joint Interpolation Motion Command

Joint interpolated motion is used in general terms here to indicate the motion of the end effector or a tool resulting from the uncoordinated motion of the different axes drives. In NC machines nomenclature this motion is referred to as point-to-point. It is primarily used for rapid movements when the exact path is not important, as in the cases of robot pick and place operations or to locate a tool in preparation for a more accurate movement. Joint interpolation motion commands for UniSet and some target NC machine and robot languages are tabulated in Tables 4.2 and 4.3 for both absolute and relative modes respectively.

With reference to the tables, Word Address joint interpolation motion is specified by a **G00** code with x, y, and z (x, z for the lathe) as the coordinates of the destination point, and a and b are the angles of the fourth and fifth axes of an NC machine. Codes **G90** and **G91** are used to define absolute and relative motion, respectively. When positioning a tool in preparation for machining, an important factor to consider is the cutter diameter compensation. This functionality is typically built into the NC controller. The magnitude of the compensation is deduced automatically from the tool definition instruction. Codes **G40**, **G41**, and **G42** indicate compensation cancel, compensation left, and compensation right, respectively. The compensation can occur in the XY plane (code **G17**), in the XZ plane (code **G18**), or in the YZ plane (code **G19**). These commands allow the user to

avoid calculating the exact tool centre location. This compensation is valid only in absolute motion mode.

In APT the motion mode toggles to joint interpolation (or point-to-point motion in NC machines terminology) by specifying **RAPID** before the motion commands **GOTO** and **GODLTA** for absolute mode and relative mode respectively. After a **RAPID** specification, motion remains in joint interpolation mode without the need for respecifying **RAPID**, i.e., only **GOTO** and **GODLTA** commands need to be issued. Motion mode toggles to Cartesian interpolation (contouring motion in NC machine nomenclature) when a **GOTO** or **GODLTA** with a feed rate is specified. Since APT is primarily an NC machine language, this approach results in very compact and elegant means of commanding tool motion for positioning and for machining actions. The necessary argument for **GOTO** is either a symbol of a predefined point or an explicitly defined consisting of the three coordinates *x, y, z*. The optional vector *i, j, k* defines the orientation of the tool for NC machines with more than three axes. APT also provides for a command **GO/TO** to position the tool in preparation for machining along a straight line *line1*. The cutting edge would be a tangent to *line1* and either on (**ON**) a check line *line2*, before *line2* (**TO**) but a tangent to it, or past (**PAST**) *line2* and tangent to it. Both *line1* and *line2* lay in a predefined part plane *pl1*.

Being a robot language, VAL II provides a set of four commands specifically tailored for pick and place operations. **MOVE,** commands the robot end effector to move to a specified position and orientation. **MOVET** differs from **MOVE** in that it changes the amount of the end effector opening depending on the value of *float* during motion. **APPRO** commands the robot end effector to move to a point which is offset by the value of *offset* along the tool Z axis from a position and

orientation specified relative to the current ones. The **DEPART** instruction commands the end effector to move along the robot z axis by a distance specified by *offset*.

For an ASEA robot, the keyword **ROBOT COORD** must be specified before a **POS** (position) command for a joint interpolation motion. The joint coordinate mode remains in effect for all subsequent **POS** commands until a **RECT COORD** command is issued. The **POS** command uses three coordinates *x, y, z* for position and quaternion *Q1, Q2, Q3, Q4* to define the orientation with respect to the wrist coordinate.

Based on the above discussion and an analysis of the syntax for joint interpolation of Word Address, APT, VAL II, and ASEA presented in Tables 4.2 and 4.3, two UniSet commands are created. **Goto,rapid:( )** and **Go,rapid,tool:( )** have the **UnaryModifiedCommand** or **BinaryModifiedCommand** format and are used for absolute and relative motion respectively. While descriptive names are used for the commands arguments in Tables 4.2 and 4.3, once a target machine is specified (and hence its programming language) OOP feature of polymorphism is used to adapt the arguments descriptive names. Graphical interpretations of how UniSet caters for the different languages entries in tables 4.2 and 4.3 are shown in Figures 4.1 and 4.2 respectively. It should be noted at this point that all the arguments of UniSet map directly to those of the native languages with the exception of those of the ASEA robot. UniSet uses Euler angles (ϕ, θ, φ) referred to the robot coordinate frame to describe orientation while the ASEA language uses quaternion. Thus parameter transformation from (ϕ, θ, φ) to quaternion is required. In addition the translation from UniSet to the ASEA syntax either issues **ROBOT COORD** or does not after checking the robot motion mode. V=*i*% is appended automatically during translation, where *i* is a previously declared or a default value.

Table 4.2 Joint Interpolation; Absolute Mode

Word Address	point to point			G90[G40]G00{XxYyZz|XxZz}[AaBb]
	Tool compensated motion	XY plane	LF	G17G41G90G00{XxYyZz|XxZz}[AaBb]
			RT	G17G42G90G00{XxYyZz|XxZz}[AaBb]
		XZ plane	LF	G18G41G90G00{XxYyZz|XxZz}[AaBb]
			RT	G18G42G90G00{XxYyZz|XxZz}[AaBb]
		YZ plane	LF	G19G41G90G00{XxYyZz|XxZz}[AaBb]
			RT	G19G42G90G00{XxYyZz|XxZz}[AaBb]
APT	point to point			RAPID;GOTO/{*point*|*x,y,z*}[,*i,j,k*]
	contour motion			GO/TO,*line1*,TO,*pl1*,{TO|ON|PAST},*line2*
VAL II				MOVE {*x,y,z,o,a,t*|*location*} MOVET {*x,y,z,o,a,t*|*location,float*}
ASEA				ROBOT COORD POS V=*i*% *x,y,z,Q1,Q2,Q3,Q4*
UniSet				**Goto,rapid[,- tool|0|tool]:{*location*|*line1@line2@pl1*| x[@y]@z[, ϕ@θ[@ψ]][,*offset*|*x@y@z*]}**

Table 4.3 Joint Interpolation; Relative Mode

Word Address	G91G00{XxYyZz|XxZz}[AaBb]
APT	RAPID;GODLTA/*x,y,z*[,*i,j,k*]
VAL II	APPRO {*location*|(*x,y,z,o,a,t*),*offset*} DEPART *offset*
ASEA	ROBOT COORD POS V=*i*% *x,y,z,Q1,Q2,Q3,Q4* OFFSET *x,y,z* MM
UniSet	**Go,rapid:{*location*|*offset*|(x[@y]@z[,ϕ@θ[@φ]])}[,*x@y@z*]**

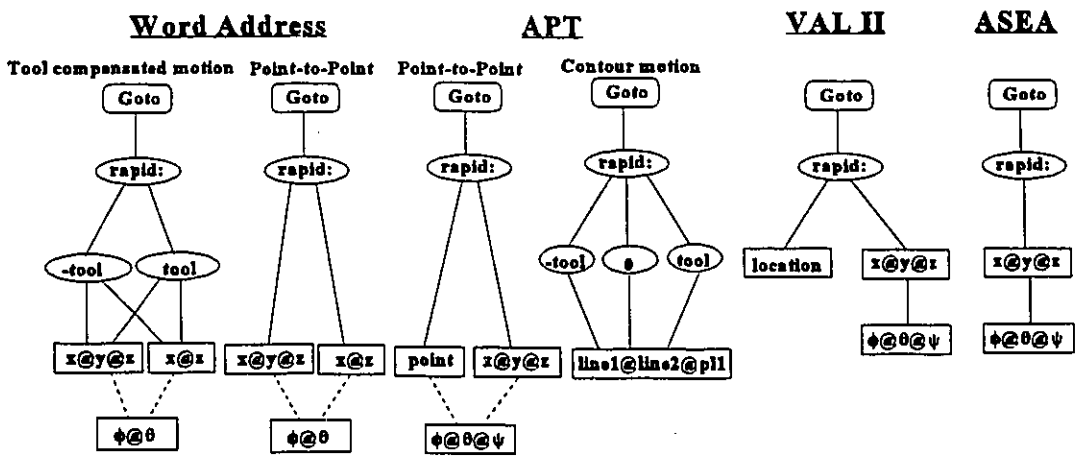


Figure 4.1: UniSet Mapping to the Target Languages for Joint Interpolation Absolute Mode.

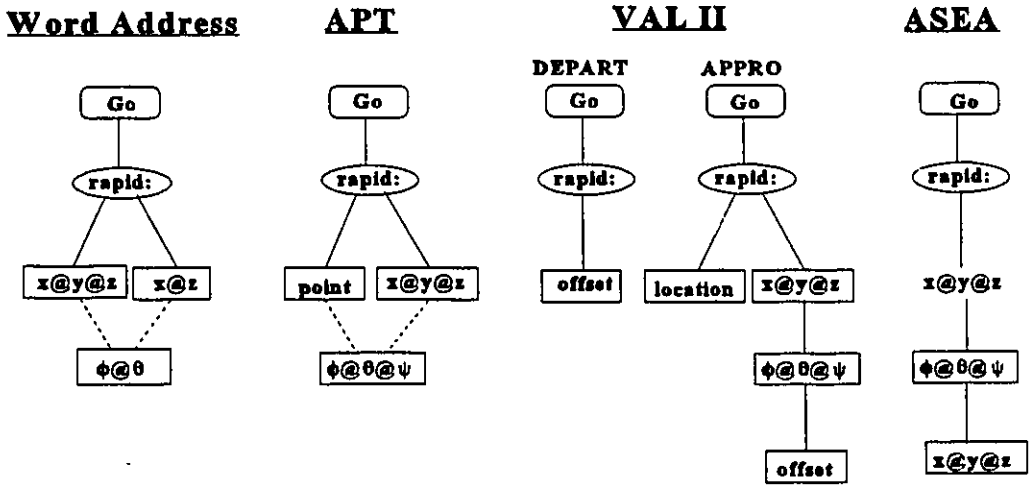


Figure 4.2: UniSet Mapping to the Target Languages for Joint Interpolation Relative Mode.

4.2.2 Linear Interpolation Motion Commands

Linear interpolation motion is used to guide an end effector of a robot or the tool of an NC machine along a straight line. Motion along other specified paths is possible, if these paths have been piecewise linearized and the line's coordinates specified. During motion, all the axes drives are coordinated to produce the exact path. Tables 4.4 and 4.5 show a comparison of the linear interpolation motion for the absolute and relative modes respectively. With reference to the tables:

For the Word Address format code **G01** defines this type of motion. All the other **G** codes described earlier for the joint interpolation motion are applicable. In addition if the optional feed rate *f* is not specified, the previously declared value is used.

While the APT words **RAPID;GOTO** and **RAPID;GODLTA** indicate a motion using a joint interpolation, when the word **RAPID** is removed, the words **GOTO** and **GODLTA** alone command motion using linear interpolation in the absolute and relative modes respectively. The position and orientation arguments for **GOTO** are identical to those for the joint interpolation motion. The necessary argument for **GODLT** is a number indicating tool motion in the machine Z direction or an explicitly defined point. When the explicit coordinate of a relative point is issued, the optional orientation argument may be used for NC machines with more than three axes. Both commands require feed rate information to perform the move. If the feed rate argument *f* is not used in the syntax, the previously declared feed rate is used. When the APT joint interpolation absolute mode command **GO** is issued in preparation for a machining operation, the command that follows is either of the linear interpolation absolute motion commands **GOFWD**, **GOLFT**, or **GORGT**. The first argument is a symbol *sl* for a motion guiding geometric entity, in this case a line. The

command **GOFWD** causes motion along *s1* where the symbol defines the same line *line1* used in the preparatory command **GO/TO**. **GOLFT** and **GORGT** cause motion to start along the left and right segments of *s1* respectively.

Motion continues till the second geometric check entity, usually a line, defined by the argument *s2*. The tool stops **ON**, **To**, or **PAST** *s2*, where these three vocabulary words have the same meaning as before. Motion takes place in the part plane *pl1* defined by the machining preparatory command **GO/TO**. The use of the feed rate argument *f* is the same as discussed above.

The VAL II commands for linearly interpolated motion are **MOVES** and **MOVEST** for the absolute mode, and **MOVES HERE** and **MOVEST HERE** for the relative mode. Similar to the case of the joint interpolation motion, the second format in each mode uses the argument *float* to affect the end effector state during motion.

ASEA uses the same move command for the linear interpolation and joint interpolation motions. As mentioned earlier, the command **ROBOT COORD** toggles the mode to joint interpolation, while the command **RECT COORD** toggles the mode to linear interpolation.

With reference to tables 4.4 and 4.5, the commands for linear interpolation in UniSet are **Goto** and **Go** for the absolute and relative modes respectively. To determine the direction of motion in ambiguous circumstances, the vocabulary words **left** and **right** are used. The arguments for the commands are identical to those for the joint interpolation, except for the addition of the feed rate *f*. A Graphical representation of the correspondence between UniSet and the different languages entries in tables 4.4 and 4.5 are shown in Figures 4.3 and 4.4 respectively.

Table 4.4 Linear Interpolation; Absolute Mode

Word Address	point to point			G90[G40]G01{XxYyZz|XxZz}[AaBb][,Ff]
	Tool compensated motion	XY plane	LF	G17G41G90G01{XxYyZz|XxZz}[AaBb][,Ff]
			RT	G17G42G90G01{XxYyZz|XxZz}[AaBb][,Ff]
		XZ plane	LF	G18G41G90G01{XxYyZz|XxZz}[AaBb][,Ff]
			RT	G18G42G90G01{XxYyZz|XxZz}[AaBb][,Ff]
		YZ plane	LF	G19G41G90G01{XxYyZz|XxZz}[AaBb][,Ff]
			RT	G19G42G90G01{XxYyZz|XxZz}[AaBb][,Ff]
APT	straight cut			GOTO/{*point*|*x,y,z*}[,*i,j,k*][,*f*]
	contour motion			{GOFWD,GOLFT,GORGT}/*s1*,{TO|ON|PAST},*s2*[,*f*]
VAL II				MOVES {*x,y,z,o,a,t*|*location*} MOVEST {*x,y,z,o,a,t*|*location,float*}
ASEA				RECT COORD POS V=*i*% *x,y,z,Q1,Q2,Q3,Q4*
UniSet				**Goto[,left|right][,-** ***tool*|*0*|*tool*]:{*location*|*s1@s2*|*x*[@*y*]@*z*[,ϕ@θ[@φ]]}[,*offset*|*x@y@z*|*f*]

Table 4.5 Linear Interpolation; Relative Mode

Word Address	G91G01{XxYyZz|XxZz}[AaBb][,f]
APT	GODLTA/{*number*|*x,y,z*}[AaBb][,f]
VAL II	MOVES HERE:{*x,y,z,o,a,t*|*location*} MOVEST HERE:{*x,y,z,o,a,t*|*location,float*}
ASEA	RECT COORD POS V=*i*% *x,y,z,Q1,Q2,Q3,Q4* OFFSET *x,y,z* MM
UniSet	**Go:{*location*|*x*[@*y*]@*z*[,ϕ@θ[@φ]]}[,*offset*|*x@y@z*|*f*]**

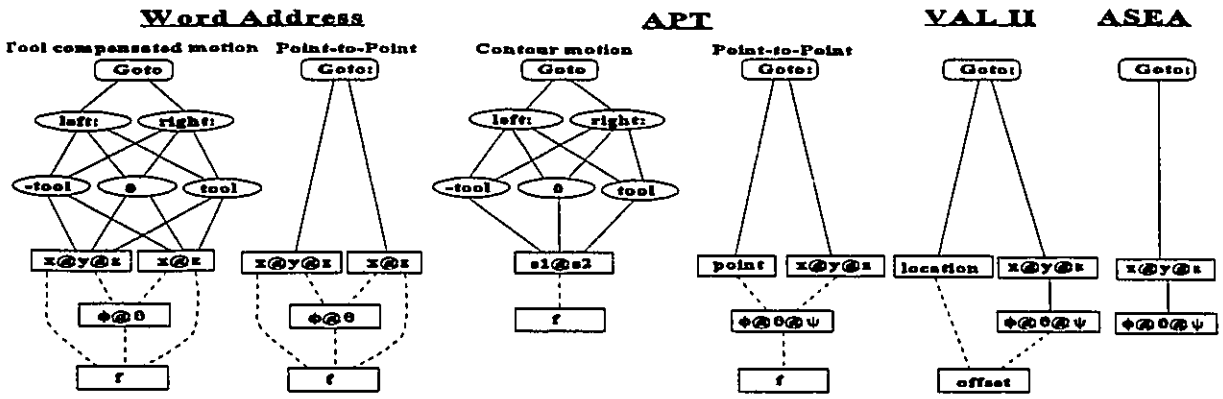


Figure 4.3: UniSet Mapping to the Target Languages for Linear Interpolation Absolute Mode.

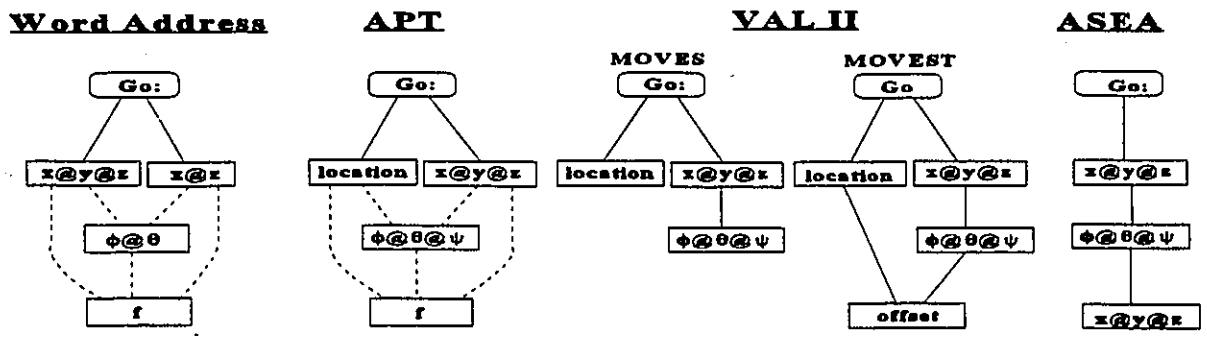


Figure 4.4: UniSet Mapping to the Target Languages for Linear Interpolation Relative Mode.

4.2.3 Circular Interpolation Motion Commands

Circular interpolation motion describes arc or circle path movements. The desired circular movement is approximated by a series of linear motions either by the post processor or the machine controller. While there are dozens of ways for defining a circle or arc, the parameters defining one that is unique can be mapped from one form to another. The choice of form is essentially a matter of practicality and convenience, and varies from machine to machine. With the exception of defining the circle or arc interpolation motion using three points (start, intermediate, and end), the direction of motion is ambiguous and must be specified. A comparison of the four target machine languages is provided in Table 4.6.

With reference to the table, Word Address provides two means for defining the geometry for circular interpolation motion. In both, the current tool position is taken as the start of the circular arc, and the first command argument is taken as the end. The second command argument can be the centre or the radius of the circle or arc depending upon whether it is a vector or a scalar. The circular motion is defined in the XY (**G17**), YZ (**G19**), or the XZ (**G18**) planes. Due to the ambiguity of the direction of motion, code **G02** is used to define the clockwise direction, and **G03** is used to define the counter clockwise direction.

In APT, a circular interpolation motion is specified by defining **ARCSLP/ON** in a preceding instruction to the circular motion. **ON** indicates that the interpolation will be carried out by the machine controller. Most NC machines manufacturers provide circular interpolation routines in their controllers. The APT instruction defaults are used as indicated in the table.

VAL II does not provide circular interpolation motion. Consequently, UniSet carries out the process of interpolation and uses the VAL II linear interpolation commands to carry out the motion.

To perform a circular interpolation motion on an ASEA robot, the sequence shown in Table 4.6 is required. **RECT COORD** is required only if the robot is not in linear interpolation mode. The current robot location and orientation are taken as the start of the circle, the position and orientation given after **CIRCLE** define an intermediary point on the circle, and the last instruction defines the position and orientation of the end of the circle.

The UniSet command for a circular interpolation motion is **GoCircle**. The optional modifiers **CW** and **CCW** set the direction of motion when ambiguous. The optional modifier **3P** indicates that the arguments provided are three points, and **2PC** indicates that they are two points and the centre.

Table 4.6 Circular Interpolation

Word Address	XY plane	CW	G17G02X*x*Y*y*{I*i*J*j*|R*r*}
		CCW	G17G03X*x*Y*y*{I*i*J*j*|R*r*}
	XZ plane	CW	G18G02X*x*Z*z*{I*i*J*j*|R*r*}
		CCW	G18G03X*x*Z*z*{I*i*J*j*|R*r*}
	YZ plane	CW	G19G02Y*y*Z*z*{I*i*J*j*|R*r*}
		CCW	G19G03Y*y*Z*z*{I*i*J*j*|R*r*}
APT			ARCSLP/ON TLOFPS, {GOFWD|GOLFT|GORGT}/*circle,line*
VAL II			N/A
ASEA **robot must be in rectangular coordinates mode**			RECT COORD POS V=i% *x,y,z,Q1,Q2,Q3,Q4* Pos V=i% CIRCLE *x,y,z,Q1,Q2,Q3,Q4* POS V=i% *x,y,z,Q1,Q2,Q3,Q4*
UniSet			**GoCircle[,CW|CCW][,3P|2PC]:{*circle|circle@line|x1@y1@z1*[,*ϕ1@θ1@φ1*],*x2@y2@z2*[,*ϕ2@θ2@φ3*][,*x3@y3@z3*[,*ϕ3@θ3@φ3*]]}[,radius]**

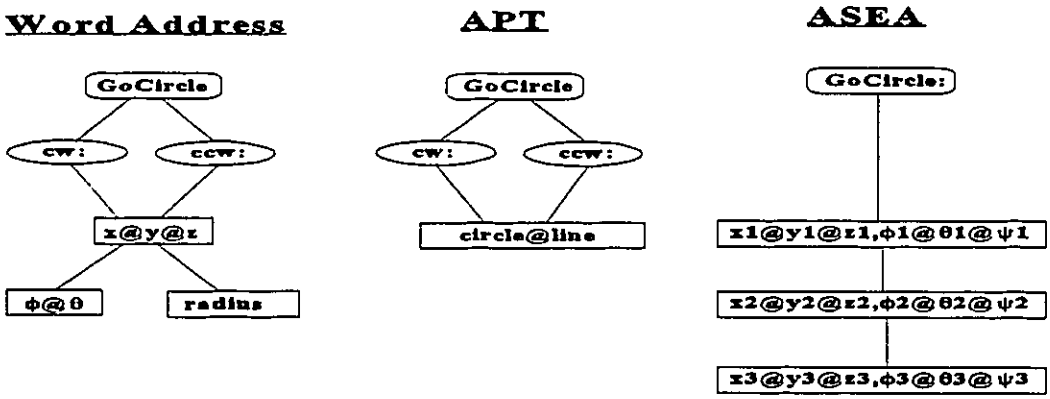


Figure 4.5: UniSet Mapping to the Target Machines for the Circular Interpolation Mode.

4.3 Configuration Commands

The configuration commands are typically defined at the beginning of the program. They are used for machine preparation and initialisation of parameters and symbols. Generally speaking, the machine parameters and the symbols remain in effect for the execution life of the program. Depending on the machine however, some of the machine parameters may be reset repeatedly during the course of the execution.

4.3.1 Speed Command

The speed command sets the speed of end effector, in the case of a robot, or the tool feed rate in the case of an NC machine. The set speed can be changed at any time. With the exception of the ASEA robot language, speed programming is straight forward. The ASEA requires the setting of a base speed. Every move instruction also requires setting a value which indicates the percent of

base speed at which the end effector moves. A comparison of speed instructions for the target languages and for UniSet is shown in table 4.7.

Table 4.7 Speed Instructions

Word Address	Unit/min	G94F*number*
	Unit/rev	G95F*number*
APT	Unit/min	FEEDRAT/*number*
	Unit/rev	FEEDRAT/ITR,*number*
VAL II	Inches/sec	SPEED *number* IPS ALWAYS
	mm/sec	SPEED *number* MPS ALWAYS
	Percentage	SPEED *number* ALWAYS
ASEA	Basic Speed	V=*number* MM/S
	Max Speed	MAX=*number* MM/S
UniSet		**Speed[,unit/sec|unit/rev|percent|max]:*number***

4.3.2 Spindle and Coolant Control Instruction

Specific to the operation of machine tools is the control of the spindle and coolant. Since spindle and coolant controls are vital machine tool functions, UniSet must include such primitives. Robots do not need such commands, and hence the commands are not applicable for ASEA and VAL II. A comparison of the spindle and coolant control instructions for Word Address and APT, as well as for UniSet is provided in Tables 4.8 and 4.9.

For spindle control both Word Address and APT have equivalent commands: off, clockwise on, and counter clockwise on. The unit of rotation of the spindle is revolutions per minute. In APT, the instruction **SPINDLE/ON,CLW** is used in the middle of a program to turn the spindle on again, with the same rotation speed as defined previously. UniSet provides the command **Spindle** with necessary modifiers to control spindle rotation and direction.

Table 4.8 Spindle Control Instructions

Word Address	Spindle off	M05
	Clockwise	M03S*int*
	Counter-Clockwise	M04S*int*
APT	Spindle off	SPINDLE/OFF
	Clockwise	SPINDLE/{*int*|on},CLW
	Counter-Clockwise	SPINDLE/{*int*|on},CCLW
VAL II		N/A
ASEA		N/A
UniSet		Spindle, {off|cw|ccw}[,*number*]

Word Address does not provide a consistent way to implement the coolant control. The extra "MISCELLANEOUS" (M) functions provided with the controller must be used. APT on the other hand provide five specific instructions for coolant control.

4.3.3 Geometry Definition Commands

Programming a contouring motion basically consists of directing a tool to move along the

Table 4.9 Machine Tool Coolant Control Instructions

Word Address		Extra 'M' Function
APT	Flood	COOLNT/FLOOD
	Mist	COOLNT/MIST
	Tapping	COOLNT/TAPKUL
	On	COOLNT/ON
	Off	COOLNT/OFF
VAL II		N/A
ASEA		N/A
UniSet		**Coolant,{off|on|flood|mist|tapping}**

profile of a part. Thus, the profile of a part is the guideline for the cutter profiling motion. Usually the outline of a part is made up of various geometric entities, such as lines, points, circles etc. These elements must be defined before a cutter or a tool motion is specified or programmed.

Table 4.10 Geometry Descriptions

APT	Point	*symbol*=POINT/*x,y,z* *symbol*=POINT/INTOF,*line1,line2*
	Line	*symbol*=LINE/*point1,point2* *symbol*=LINE/*point*,PARLEL,*line*
	Plane	*symbol*=PLANE/*point1,point2,point3* *symbol*=PLANE/*point*,PARLEL,*plane*
	Circle	*symbol*=CIRCLE/CENTER,*point*,RADIUS,*number* *symbol*=CIRCLE/*point1,point2,point3*
VAL II	Point	POINT *symbol*=*x,y,z,o,a,t* HERE *symbol* ; Teach-In Method
ASEA	Point	STO *symbol* ;Tech-In method
UniSet	**Point**	**Point,*symbol*:*x*@*y*@*z*[,ϕ@θ@φ]**
	Line	**Line,*symbol*:*point*|*x1*@*y1*@*z1*,*point*|*x2*@*y2*@*z2***
	Plane	**Plane,*symbol*:*point*|*x1*@*y1*@*z1*,*point*|*x2*@*y2*@*z2*,*point*|*x3*@*y3*@*z3***
	Circle	**Circle,*symbol*:{*point1*|*x1*@*y1*@*z1*,[*point2*|*x2*@*y2*@*z2*,[*point3*|*x3*@*y3*@*z3*]]} [,radius]**

APT provides sufficient geometric entities for most matching application, but Word Address is not rich in entity definition. In VAL II a point can be defined explicitly using the position and orientation of the end effector. Both VAL II and ASEA provide a teach-in method for defining and storing a point entity. Table 4.10 shows some of the geometry definition commands for the target languages and for UniSet.

4.4 Control Commands for Auxiliary Devices

Robot grippers, machine tools chuck, tool holders and vices have the same functionality in that all are used for grasping an object. Such devices are considered as auxiliary ones, and can

generally be added independently by the user to suit the application. Generally speaking automated NC machines can be purchased with both a tool chuck and a vice installed. Word Address instructions for the tool chuck are usually machine specific and implemented using the extra **M** MISCELLANEOUS codes, for example **M21** and **M22**.

VAL II provides instructions for opening and closing the gripper plus proportional control of the gripper. The gripper commands **CLOSE** and **OPEN** are proportional commands, when accompanied by a number, causing the gripper to be opened to the number specified.

The gripper control for the AREA robot is a boolean function **GRIPPER** that alternates the gripper state.

Gripper command comparisons and UniSet instructions including control command for non-programmable, user installed devices, **OPEN** and **CLOSE**, are shown in Table 4.11.

Table 4.11 Clamping Control Instructions

Word Address[1]	Clamp 4th axis	M21
	Unclamp 4th axis	M22
APT	N/A	N/A
VAL II	Close gripper	Close
	Open gripper	Open
	Proportional open or close	Open *number* or Close *number*
ASEA	Close gripper	GRIPPER
	Open gripper	GRIPPER
UniSet		**{Open|Close}[:*number*]** **Device,*deviceName*:{start|stop}**

1 These extra 'M' codes vary between machines

4.5 Cutting Tool Change Instruction

Most modern machine tools can automatically change tools during a machining operation. The tool axis is always the spindle axis (except for lathes), and the differences in tool dimensions for different tools appear in the tool length and diameter only. In Word Address, Code **G10** is used to specify the tool number, cutter diameter, and length offset. For APT, the diameter is taken care of by the **CUTTER** statement as shown in Table 4.12.

After defining a tool specification, a tool can be changed by instructions **G45** and **LOADTL** in Word Address and APT, respectively. UniSet provides both tool definition and tool change as a **Command** instruction format as shown in Tables 4.13.

Table 4.12 Tool Cutter Definition

Word Address	G10H*numberXdiameterZlength-offset*
APT	CUTTER/*diameter*
UniSet	**Tool:{*number,diameter*}**

Table 4.13 Tool Change Instruction

Word Address	M06T*number* G45H*number*
APT	LOADTL/*tool-No*
UniSet	**ToolChange: <*number*>**

4.6 Program Control Instructions

Program control instructions are a necessity for advanced programming. Machine tool and robot languages provide varying degrees of program control. Word Address provides very simple control structures. ASEA's control structures are more developed. VAL II and APT control structures involve a much higher level of sophistication. Based on low level control structures, it is possible to implement the higher level control structures. UniSet primitives will be concerned with both the low level primitives and high level primitives, such as loop control, etc.

Table 4.14. shows the instruction list for program end and pause. For Word Address, Code **M30** or **M02** are controller specific and are used to terminate the part program. All other languages provide instructions for the program end. The pause instruction is also supported by all languages, but only VAL II allows program execution to resume. UniSet Instructions are developed to cater for all these variances as shown in Table 4.14.

Table 4.14 Program End or Interrupt Instruction

Word Address	program end	M30|M02
	interrupt	M00
APT	program end	END
	interrupt	STOP
VAL II	program end	STOP
	interrupt (resume)	PAUSE (PROCEED)
ASEA	program end	Return
UniSet		**{End|Interrupt|Resume}**

Table 4.15 Time Delay Instruction

Word Address	G04X*number*
APT	DELAY/*number*
VAL II	DELAY *number*
ASEA	WAIT (TIME *number* S)
UniSet	**Delay: <*number*>**

A pause is very useful during machining or during robot movement to produce an accurate move, for example, in dealing with sharp corners or for a robot refined motion. As illustrated in Table 4.15, all languages provide instructions for delaying operation for a given time period.

In order to create a program effectively, program loops and conditional branching instructions are needed. APT and VAL II provide these functionalities. Currently, UniSet provides two types of high level program control commands, **Do** and **IF** as shown in Table 4.16.

Table 4.16 Program Loop and Branch Control Instructions

APT	Do/k,I=a1,a2 ----- k) contin	IF logic a1,a2,a3
VAL II	Do ---- until (logical Expression)	IF (logic) goto a1
UniSet	**Do[,*symbol:number1,number2*]** ---- **unitil[:*logic*]**	**go, If:*logic*,{*a1*|*a1,a2,a3*}**

4.7 Logical Communication Control Instructions

Machines must be able to communicate to the cell controller, sensors, and other devices. Word Address does not provide specific means for communicating information. Thus, **M** codes must be employed for this purpose. APT is lacking in this area as well. Both robot languages provide adequate output and input signals. A comparison of these instructions is shown in Table 4.17.

In VAL II, a communication line is set when the arguments of either **SIG** or **WAIT** positive, and reset when the argument is negative. ASEA language provides means for setting, resetting, toggling, or pulsing a specific communication line. In addition it is capable of monitoring an input communication line and reporting its state using a logic function. UniSet logical communication instructions consist of the two commands, **Wait** and **Signal** for input and output signals respectively. Similar to the ASEA the output line can be set, reset, toggled or pulsed.

Table 4.17 Communication Control Instructions

Word Address		Extra 'M' code
APT		N/A
VAL II	out put signal	SIG({*line number*|-*line number*})
	input signal	Wait({*line number*|-*line number*})
ASEA	out put signal	{SET|RESET| INVERT|PULSE} OUTP line number
	input signal	WAIT UNTIL INP *line number*={0|1}
UniSet		**{Wait|Signal}, {reset|set|invert|pulse}:*line number***

CHAPTER 5
Task level UniSet

UniSet Commands provide the user with an instruction set suitable for command level programming. At the command level, every action that the cell constituents have to perform in order to accomplish a task must be detailed and coordinated with other actions by the cell programmer. For example, the task of loading a part onto the NC machine in the FMC of Figure 3.1 requires a large set of commands to the robot to pick the part from the conveyor belt, and deliver it to the chuck of the machine. Some actions have to be also taken by the conveyor belt and by the NC machine.

To coordinate the interaction of the robot with the conveyor belt and those of the robot and the NC machine, interlock signals will have to be exchanged between the various controllers. All these details have to be thought of and programmed by the cell programmer. The programmers' task is further complicated by the fact that three different machine languages may be involved.

While UniSet has alleviated the problem of dealing with multiple machine languages, cell programming still requires considerable expertise in manufacturing and a great deal of attention to details. At the preliminary stage of developing an FMC program where the machine interaction is of paramount importance, some of the details like feed rate, depth of cut, and exact robot trajectory between source and destination, may be more of a bother than an immediate necessity.

If the system has already been provided with "expertise" in the different domains of manufacturing and FMC control, it would have the necessary details about how to execute its basic functions. The remaining work left to the programmer would be to describe what tasks need to be performed to accomplish their goals. Based on the above idea, a suitable task definition language, called Task level UniSet, has been developed. This chapter addresses the development methodology of the Task level UniSet.

5.1 Syntax of Task level UniSet

Task level UniSet has rules regarding syntax and command structure. Without consistent syntax and design, the programmer's job is more difficult. The syntax of the Task level UniSet is similar to the sentence structure of English. However, in order to be recognized by the system certain syntax rules must be followed.

The following Extended Backus Naur Formalism (EBNF) syntax specification includes the grammatical constructs for Task level UniSet:

Command = [machineName] action object {[preposition] [arguments]} "."
object = [adjective] noun [noun]
preposition = prepositionName: {threeDimensionExpression|noun}
threeDimensionExpression = number "@" number "@" number
arguments = {parameterExpression ":"} {valueType} [;]
valueType = integer| float

An EBNF specification is a sequence of syntax rules. The right-hand side of each rule defines syntax in terms of other rule names and terminal symbols of the language. Braces, {} identify expressions which may occur several times. Brackets, [], identify optional expressions. A

string is written as a character sequence enclosed by paired, unmatched, double quotations (" ") and identifies terminal symbols of the defined language.

The syntactical entities for the terminal categories of **action, noun, parameterExpression** considered in the development are dependent on the **machineName** entity, i.e., implemented machine.

The **machineName** is the name of a cell component which carries out the given task. If the next task requires the previous used machine tool, the current **machineName** can be omitted. The **action** represents the task to be completed and should be a verb. The **object** consists of a noun or two **nouns** that may be following an adjective and expresses a target **object**, for example *aPart*. A Blank space is required between any two elements in a line. One or more **prepositions** and **arguments** are needed in the command structure. Both of these have a numeric value using the":" character. The ";" character serves as a syntax delimiter between preposition and arguments. The "." character is the end of command delimiter. Examples satisfying the above rules are shown in Table 5.1.

Table 5.1 Examples Tasks

task	name	action	object	preposition	argument
aMill make slot at:1@1@1; length:12; width:12; depth:1.	aMill	make	slot	at:1@1@1	length:12 width:12 depth:1
aLathe cut external rotation surfacet at:1@1@1; length:2; depth:2.	aLathe	cut	external rotation surface	at:1@1@1	length:2 depth:2
aRobot load aPart to:aMill	aRobot	load	aPart	to:aMill	

5.2 Development of the Task level UniSet

FMC programming generally ties up the cell throughout the development stage. This is because every fragment of code needs to be tested before incorporating it within the program stream. The philosophy behind the development of Task level UniSet, and the fact that it is designed along the concepts of OOP, allow its instructions to be reused freely once they have been created and tested. Instructions have been developed, and are continually being developed, for the control of robots, sensory devices, vision system, machine tools, part feeders, and other devices used in manufacturing cells. Although many robot and NC machine control languages exist, few have the ability to control all modern day cell functions. The Task level UniSet is intended to combine all necessary control features into one cell programming language.

Task level UniSet allows programming at the task level and automatic decomposition of cell components tasks into a hierarchical sequence of more and more elementary sub-tasks. Here, an operation is considered in terms of sub-tasks that the cell components must perform. Task level UniSet is the task oriented description, i.e., "what to do," and is a formal variant of natural language. Its commands may be termed implicit, as they are not characterized by the detailed description of the actions that the machines have to make in order to accomplish its task. All the information needed to perform the task is encapsulated. Cell programs developed using Task level UniSet commands can be easily read and understood by less experienced operators, because it features meaningful naming of commands, and variables. Furthermore such operators can easily develop

functional cell programs, since they can deal with one cell language that does not require a great deal of knowledge about the minute details of cell programming and control.

Class hierarchy of some tasks performed by cell components is shown in Figure 5.1. With reference to this figure, these tasks are classified into two groups: **cell configuration dependent tasks**, and **cell configuration independent tasks**. Cell configuration dependent tasks are those that require some coordination among cell components to carry out the task. For example, the task **load** as in "aRobot load a part to:aMill" require that the actions of aRobot and aMill be coordinated.

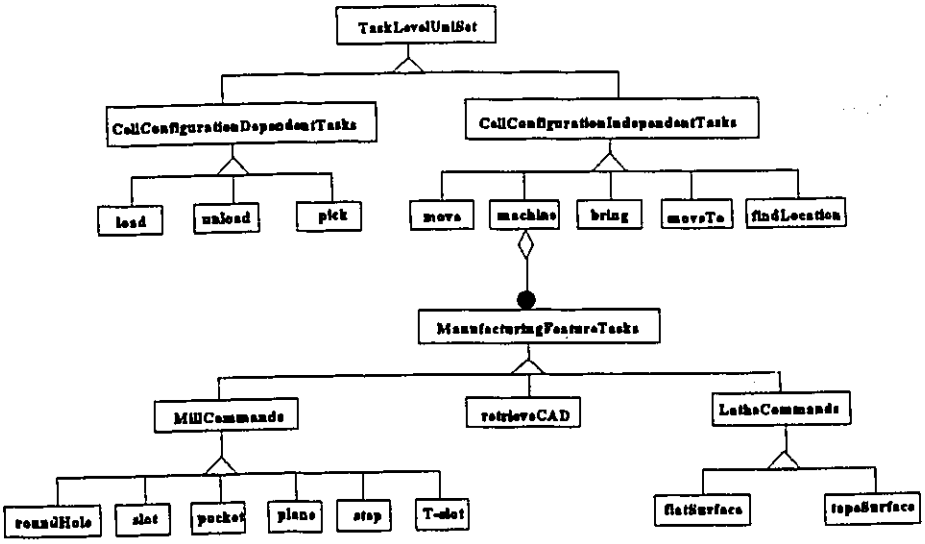


Figure 5.1: Hierarchy of classes defined in the Task level UniSet

Cell configuration independent tasks require only one cell component to perform the task. The task **moveTo** as in "Robot move to:*MachineName*" is an example of a cell configuration independent one, because it is carried out by the Robot without interacting with other components. Another example is task **machine** as in "Mill machine *aPart*". This task is invoked by one of many

instances of class **ManufacturingFeatureTasks**. This task, in turn, is classified into the **MillCommands**, the **LatheCommands**, and the **retrieveCAD**. The first two command groups are further decomposed into more specific commands such as **slot** and **tapeSurface**. The **retrieveCAD** command is used to load and execute a tool path file generated by a CAD system and post processed.

5.2.1 Cell Configuration Independent Tasks

As has been mentioned above, these tasks are performed by a single cell constituent like a robot or NC machine without requiring any interaction with other cell constituents. Some tasks that have been defined in the UniSet environment are explained below:

[conveyorName] bring *aPartName*

This command is used to bring a part (*aPartName*) into the cell by a conveyor.

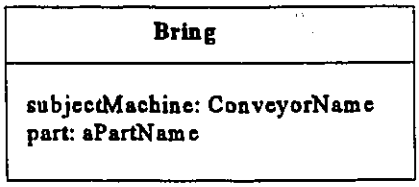


Figure 5.2: Class for the bring Command

The class of the command is **Bring** is shown in Figure 5.2. As shown in the Figure it has two instance variables: the *subjectMachine* and the *part*. The instance variable *subjectMachine*

contains the name of the machine to perform this task, in the example above *ConveyorName*. While the other contains the name of the part (*aPartName*) that is carried by the conveyor. The values of these instance variables are obtained from arguments of the command.

[visionSystemName] findLocation <*aPartName*>

This command is used to obtain data about the position and the orientation of a part (*aPartName*) using a vision system (VisionSystemName).

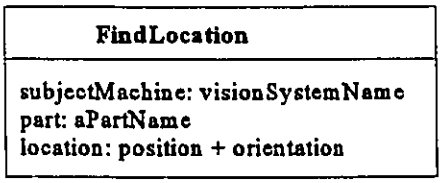


Figure 5.3: Class for the find Command

Figure 5.3 shows the class for this command. As can be seen, three instance variables are defined in this class. The instance variable *subjectMachine* is associated with a vision system. The instance variable *part* contains information about a part (*aPartName*) to be recognized by a vision system. The result of the computation is stored in the instance variable *location*. The value of *location* is the position and the orientation of the part.

[robotName] move to: <*machineName*>

This command is used to define a movement of a robot (RobotName) with a part or without a part.

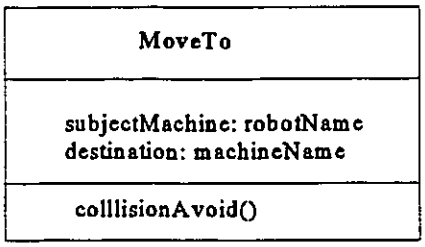


Figure 5.4: Class for the moveTo Command

As shown in Figure 5.4 two instance variable are defined in this class. The instance variable *subjectMachine* holds the name of the machine to perform this task. The instance variable *destination* contains the desired final location, in Figure 5.4 the argument (*machineName*). During the move collision avoidance should be considered. The operation *collisionAvoid()* is responsible for ensuring a collision free path within the cell.

[machineName] retrieve *aPartName* from:<*CADFile*>

This command is used to retrieve a code from a CAD file (*CADFile*) into a part name (*aPartName*). The command allows the importation of a tool path file developed separately from the UniSet Environment.

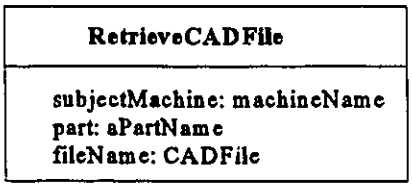


Figure 5.5: Class for the retrieve Command

The class **RetrieveCADFile** for this command is shown in Figure 5.5. Three instance variables are defined: the *subjectMachine* which holds the name of the machine (*machineName*) to execute a *CAD file*, the *part* which contains a drawing part name, and the *fileName* which contains the name of the file including a drive and a directory name.

[machineName] machine *aPartName*

The class **Machine** is the class of the machining tasks that the cell machines can perform. This class contains several subclasses for different manufacturing feature tasks.

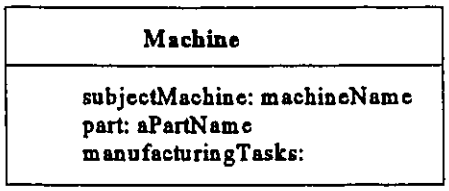


Figure 5.6: Class for the machine Command

The description of the class **Machine** is shown in Figure 5.6. It has three instance variables: the *subjectMachine* which refers to a kind of NC machine, the *part* which contains the name of a part to be machined, and *manufacturingTasks* which collects many manufacturing feature tasks.

The manufacturing feature tasks are based on the manufacturing feature description of the material which is to be removed from raw stock with a machine tool. Each command corresponds to a feature and its name contains a particular feature. For example, the task "make slot" corresponds to the **slot** feature. Also manufacturing features correspond to a specific machine tool that have the capability to machine the feature.

The task model data have been implemented using an object-oriented approach for dealing with features and their components. All features have similar physical attributes such as position, orientation, and geometry description, as well as similar operations. Using an object-oriented approach, different features can be dealt with similarity. This isolates the local characteristics of the individual features from the general function of the system and facilitates the adding of new features.

Two kinds of verbs are used in this command set, **make** and **cut**. **Objects** represent the desired feature that must be performed by the *machineName*. This is the key word of the machining commands in the Task level UniSet. The Preposition **at** is needed to determine where the feature is located with respect to the reference position. These locations vary according to the selected feature. For example, for the hole the location is the top centre of the hole, whereas, for the T-slot, the location is on the left side. Arguments, such as depth, width, etc., are used to fill in values for attributes of the feature object, and to support information to complete the required task.

Some important feature commands that have been developed will be explained below.

[MillName |DrillName] make hole at:<*origin*>; **radius:**<*float*>;**depth:**<*float*>.

With the reference to Figure 5.7, the manufacturing feature related to this command is a hole. The hole can be machined by a mill or a drill. Radius and depth are defined as the geometric parameters shown in Figure 5.7. The origin position is the centre of the hole on the part top surface.

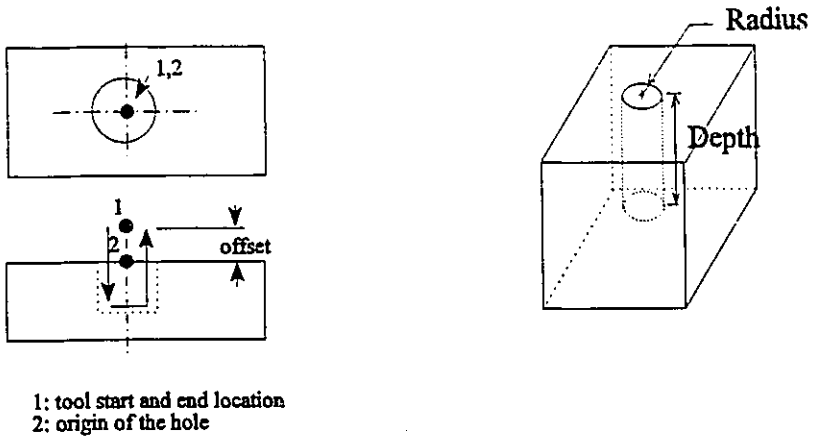


Figure 5.7: Tool Path Diagram for the Hole operation

Other salient information are the tool start and end location with respect to the workpiece. The start location is obtained by adding an offset to the origin position along the tool axis. The offset value is given by the Environment. After machining the hole, the tool should return to the end location For this command the two locations coincide. Lines with arrows represent the tool path to make a defined hole.

[MillName] make slot at:<*origin*>;**width:**<*float*>; **length:**<*float*>; **depth:**<*float*>; **radius:**<*float*>.

This command is used to machine a slot feature. Figure 5.8 shows the tool path diagram for machining a slot. The slot geometry consists of its width, length, depth, and radius. The symbols **dx**, **dy** and **dz** are the X, Y, and Z direction tool increments, respectively. The first two values are chosen according to depth of cut, length, and width. The other value **dz** is determined based on the depth and tool geometry.

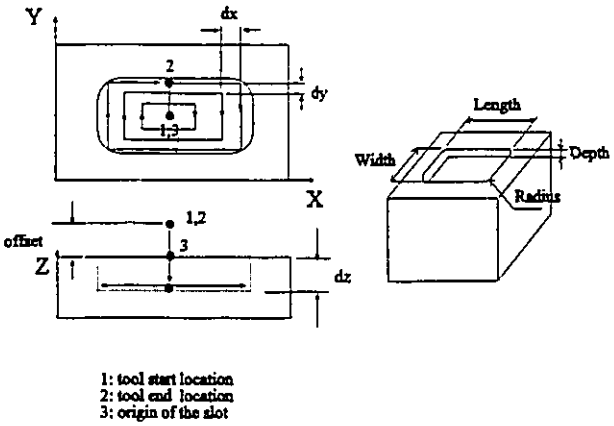


Figure 5.8: Tool Path Diagram for the Slot operation

[LatheName] cut external rotation surface at:<*origin*>; **length:**<*float*>;**depth:**<*float*>.

This command is used to cut a depicted feature, called "external rotation surface". As shown in Figure 5.9, this feature is defined by the depth and length as geometric parameters. Similar to the previous commands, three points are defined. This operation typically requires multiple passes. The depth of cut **dx** indicates one pass distance. dx and **depth** affect the number of tool paths.

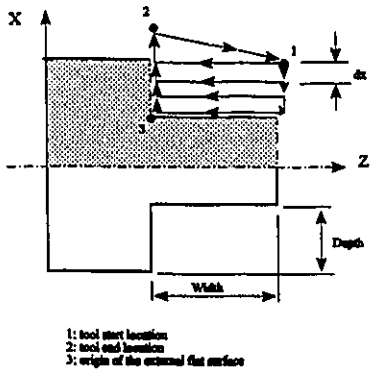


Figure 5.9: Tool Path Diagram for the External Rotation surface operation

5.2.2 Cell Configuration Dependent Tasks

In order to complete these type of tasks, coordination with other cell components is needed. Typically these tasks involve an interaction between a robot and another cell component. Each such task can be decomposed into sub-tasks, called operations. Each operation involves a single machine and corresponds to a single activity, for instance "robot: close gripper," and "mill: close vice". These operations encapsulate knowledge about a set of sequences of detailed codes and are translated into UniSet and specified machine codes. The concurrent mechanism among cell components needed to perform a Cell Configuration Dependent Task is represented in the Operation Initiation Diagram (OID). In this diagram all the operations, states, and relationships between the cell constituents involved are expressed graphically and logically.

5.2.2.1 Operation Initiation Diagram

The Operation Initiation Diagram (OID) is the second layer diagram of the Task Initiation Diagram (TID), which will be detailed in Chapter 6. Figure 5.10 shows an example of an Operation Initiation Diagram for the task "load". All states and operations are cryptic instead of verbose. The verbose descriptions for the symbols of the states and operations are shown in Table 5.2-5.8.

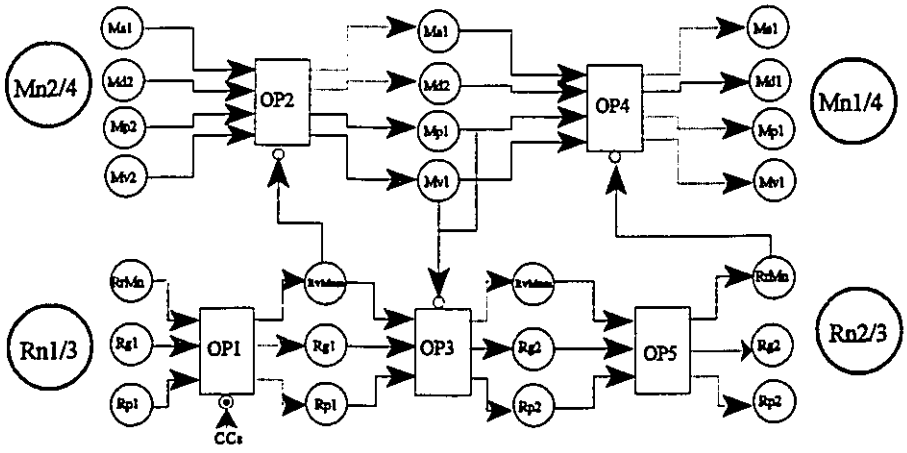


Figure 5.10: Operation Initiation Diagram for the Task "Load"

The OID is composed of two basic components: a set of operations, OP, and a set of states. The operations, OP, is categorized into two groups: guided operations OP_g and unconditional operations OP_u. A guided operation is one that requires an external trigger to start it. In the example diagram, operations OP1, OP2, OP3, and OP4 are examples of the guided operations. Unconditional operations are those that can start automatically on the onset of all the necessary states. OP5 is an example of this kind of operation.

The initiation of an operation requires a specific set of states to exist. This state set is necessary to ensure the successful and safe completion of the operation. The state set is given at any instant by a collection of states of its constituents. This state set is classified into two kinds: **rest**, and **visit**. A device or machine must be in the **rest** state before its next task can be initiate. The state sets for Robots and NC machines are shown by thick line circles, e.g., *Rn1/3* and *Mn1/4*. The last

number in the symbol shows how many individual states are required to determine this state set. The possible rest states for the robot, the conveyor, and the NC machine are listed in Tables 5.3 to 5.5 respectively, and the sensory and logical information are listed in Tables 5.6 to 5.8 respectively.

The **busy** states are transitional states between operations or two executions without interaction. They can be recognized from the robot state symbol, *Rtn* as will be seen in the OID of Figure 5.12. The small letter t indicates the transit state of the robot. These states are useful to avoid collisions with obstacles.

The **visit** states, S_v, indicates an interaction between two machines and hence requires coordination among them. The symbol of these states have the pattern "**R-M--**" for the robot, as an example, the state *RvMnm*. The small letter *v* represents the visit location state of the robot, *Mn* represents a machine served by the robot, and *m* represents the index of one of the visit locations.

Some of the visit states are the auxiliary states. They are states to trigger the start of another operation. With reference to Figure 5.10, all the states necessary for the initiation of OP1 originate from Robot *Rn*. To prevent an automatic initiation of the operation when that state conditions are satisfied, an auxiliary signal/state is introduced. The special auxiliary signal is a signal for the Cell controller, as example *CCs*. These states are indicated on the Operation Initiation Diagram by a circle with a token leading into an operation box.

States that are altered as a result of an operation are indicated by solid lines leading out of the operation box. These states are used later as auxiliary states. All other states that do not undergo transitions as a result of an operation are indicated by dashed lines leading out of the operation box. The numbers of states across an operation box must be constant. Based on above terminology for the Operation Initiation Diagram:

The Operation Initiation Diagram OID is defined as the four-tuple, OID(task)=(OP,S_v,C,O). The symbol S_v defines the set of visit state. The condition operator **C**, defines the set of state and guiding conditions necessary for each operation OP_i i.e. C(Op_i). The output operator **O**, defines the set of states resulting from each operation OP_i, i.e., O(OP_i).

The general format of these operators are as follows:

C(OPi) = {S_v} [Auxiliary states] O(OPi) = {Altered states}.

Considering the above exemplified Operation Initiation Diagram, the items of the four-tuple is given by:

OID(load) = (OP,S_v,C,O)

OP = {OP1, OP2, OP3, OP4, OP5}

C(OP1) = {RrMn,Rg1,Rp1},[CCs]	O(OP1) = {RvMnm}
C(OP2) = {Ma1,Md2,Mp2,Mv2},[RvMnm]	O(OP2) = {Mp1,Mv1}
C(OP3) = {RvMnm,Rg1,Rp1},[Mp1,Mv1]	O(OP3) = {Rg2,Rp2}
C(OP4) = {Ma1,Md2,Mp1,Mv1},[RrMn]	O(OP4) = {Md2}
C(OP5) = {RvMnm,Rg2,Rp2}	O(OP5) = {RrMn}.

5.2.2.2 Task level UniSet Commands

Some important commands that have been developed are explained below.

[Rn (Robot name)] pick *aPartName* **from:** *Cn (conveyor)* **to:** *Mn (machineName)*

This command is used to pick up a part (*aPartName*) from feed conveyor *Cn* and goes to a rest position near the machine **Mn**, and wait. The OID for this task is shown in Figure 5.11. Information about the pick up location from conveyor *Cn* and the destination rest location near machine *Mn*, pertaining to robot **Rn** are supplied by the UniSet environment. These information are typically provided during the cell set up stage.

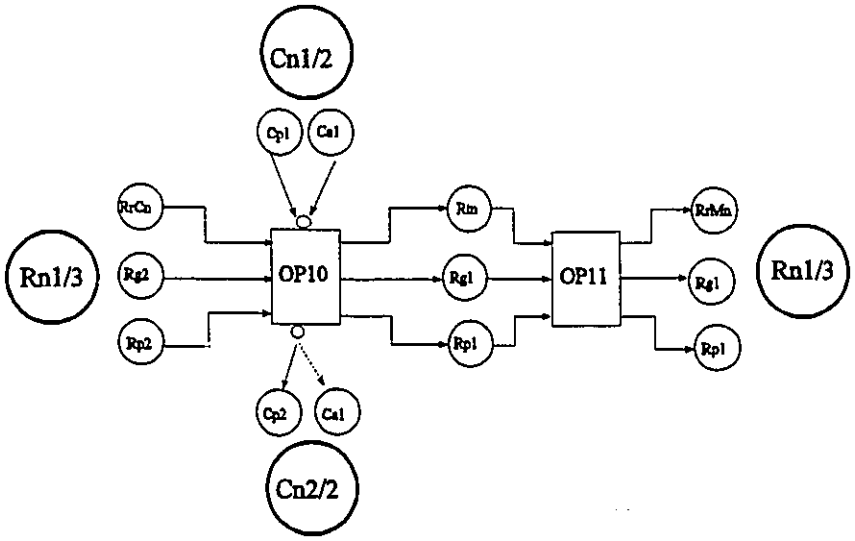


Figure 5.11: Operation Initiation Diagram for the Task "Pick "

[Rn (Robot Name)] load *aPartName* **to:** *Mn (machineName)*

This command is used to load a part (*aPartName*) into the destination machine *Mn*. This task comprises three robot sub-tasks (OP1, OP2, and OP5), and two machine sub-tasks (OP2 and OP4) as shown in the Operation Initiation Diagram is shown in Figure 5.10.

[Rn (Robot Name)] unload *aPartName* **[from:** *Ms (machineName)*] **to:***Md*|*On (machineName*|*outbin)*

This command is used to unload a part (*aPartName*) from machine *Ms* location and deliver it to out bin, or load it into machine *Md* location. As can be seen in figure 5.12, three different machines, robot, machine *Ms*, and machine *Md* or *outbin*, participate in this task. If the third machine is an NC machine which requires a part, operation OP9 encapsulates a load task. Otherwise, operation OP9 is to deliver a part to an outbin destination location.

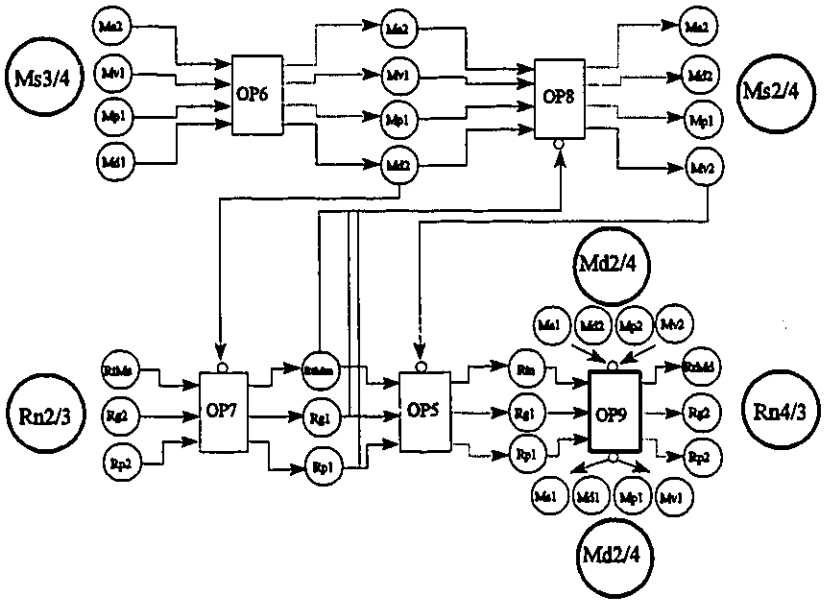


Figure 5.12: Operation Initiation Diagram for the Task "Unload"

Table 5.2 Operations list

Name	Description	Name	Description
OP1	Robot approach to load	OP2	Machine close vice
OP3	Robot open gripper	OP4	Machine close door
OP5	Robot retract from machine	OP6	Machine open door
OP7	Robot approach to unload	OP8	Machine open vice
OP9	Robot deliver or load to machine	OP10	Robot pick part
OP11	Robot move to machine with a part		

Table 5.3 Conveyor Generic States

State	Part	Activity	Description
Cn1/2	Cp1	Ca1	part is available
Cn2/2	Cp2	Ca1	part is not available

Table 5.4 Robot Generic Composite states

Name	Gripper	Part	Location	Name
Rn1/3	Rg1	Rp1	RrMn	ready to load part
Rn2/3	Rg2	Rp2	RrMn	ready to unload part
Rn3/3	Rg2	Rp2	RrCn	ready to pick part
Rn4/3	Rg2	Rp2	Rrn	wait after delivered part

Table 5.5 NC Machine Generic Composite States

State	Vice	Part	Door	Activity	Name
Mn1/4	Mv1	Mp1	Md1	Ma1	ready to machine
Mn2/4	Mv2	Mp2	Md2	Ma2	ready to load
Mn3/4	Mv2	Mp1	Md1	Ma3	ready to unload

Table 5.6 Robot sensory and logical information

state	symbol	explanation
Gripper	Rg1	close
	Rg2	open
Part	Rp1	present
	Rp2	absent
Location	Rr[Mn|Md|Ms|Cn]	rest location
	Rv[Mn|Md|Ms|Cn]	visit location
	Rtn	transit location

Table 5.7 NC machine sensory and logical information

state	symbol	explanation
clamp (i.e. vice)	Mv1	close
	Mv2	open
Part	Mp1	present
	Mp2	absent
Door	Md1	close
	Md2	open
Activity	Ma1	ready to machine
	Ma2	stopped
	Ma3	machining

Table 5.8 Conveyor sensory and logical information

state	symbol	explanation
Part	Cp1	present
	Cp2	absent
Activity	Ca1	stopped
	Ca2	moving

CHAPTER 6
UniSet Environment

This chapter addresses the overall structure of the UniSet Environment, the object data models involved in the CIM, and the hybrid knowledge representation methodology imbedded in the environment. The UniSet system components visibility and design methodology of the environment are also discussed in this chapter.

6.1 Over view of the UniSet Environment

The UniSet Environment is designed as an object-oriented knowledge-based system to provide a system for configuring, programming, simulating, and controlling manufacturing cells. Figure 6.1 portrays schematically the UniSet Environment. With reference to the figure, the environment comprises four modules which capture and process information about the cell program, existing cell facilities and data, and distribute the information into appropriate object mode, so that the cell can perform production tasks.

Each of these modules is responsible for a specific function as follows: (a) The cell setup module is primarily used for defining the cell and its components and their functionalities, reconfiguring a cell, or building preparation databases. (b) The programming module allows the user

to edit cell programs which may be developed in Task level UniSet, UniSet, or native machine

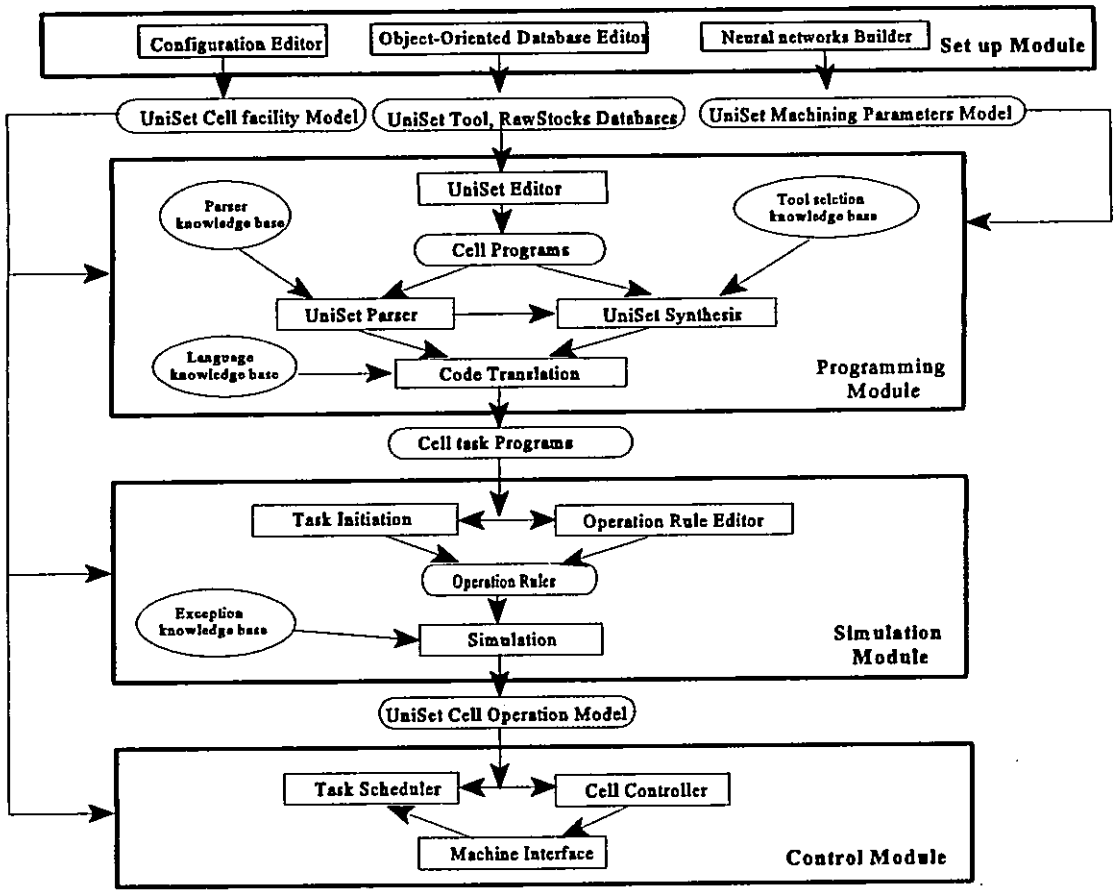


Figure 6.1: Architecture of the UniSet Environment

language. The programs written in Task level UniSet are used to generate detailed UniSet codes which are in turn translated into specific machine codes by the generation sub-module. (c) The compiler/simulator module is primarily used for generating the control structure for the operation of the cell according to the user program as well as the knowledge contained in the data bases. This control structure can also be used to simulate the cell operation and visually ensure that the control structure is free from errors. The cell user can invoke and exercise control over the compiler and the simulator sub module. (d) The control module is used to coordinate the workstation, as well as carry out the task of harmonizing communication between the devices. The task of this module is performed by the Task Scheduler which relies on information from Virtual Devices (VDs) established in the Set up module to mirror the physical devices of the cell.

All information involved in Flexible Manufacturing Cells is represented as object data models in the UniSet Environment. Object-Oriented programming provides mechanism for encapsulation and modularity as well as facilities for information retrieval and communication. Using these capabilities, the model can be developed and accessed through message passing by related modules. The models are classified into two major groups: a) UniSet Cell Configuration Model (UCCM), which contains information pertaining to the cell and facilitates the building of the UCOM, and b) UniSet Cell Operation Model (UCOM), which deals with production plans.

6.2 UniSet Object Data Model

All entities contained in the environment, such as the manufacturing cells, jobs, parts, machines, tools, operation programs, and operation knowledge, are represented by objects. Based on the nature of the entities and their representing objects, two Object-Oriented Data Models are created in the environment to collect objects with a similar role. The UniSet Cell Configuration Model (UCCM) captures the characteristics of the cell and cell equipment and collects information and data required in a manufacturing cell. The UniSet Cell Operation Model (UCOM) contains knowledge and data about the behaviour of the cell and cell equipment.

6.2.1 UniSet Cell Configuration Model (UCCM)

This model contains all the information and knowledge to carry out developing production plans. This information includes the cell facilities, the tool and raw stock databases, and machining parameters knowledge.

6.2.1.1 UniSet Cell Facility Model (UCFM)

The UCFM contains the structure of a manufacturing cell. Since a manufacturing cell consists of several devices, this model includes the characteristics of the equipment forming the cell as well as the relationships they can have with other equipment. The cell may contain general-purpose devices such as robots, material-handling equipment such as conveyors and AGVs, machining equipment such as NC machines, and simple devices such as part feeders, tool fixtures, and buffers for work-in-progress.

The information to be kept can be divided into two categories: static and dynamic information. Static information does not change as a result of cell activities. This information is accessible to all modules in the UniSet Environment. Static information is organized based upon a component's general feature, geometric, communication, and sensory characteristics.

The general characteristics include such salient features as the number of axes, programming language, etc. The geometric components of the equipment model contain the geometry of the parts that the equipment is composed of. All equipment has a physical manifestation. Therefore, all equipment requires a geometric description, such as volume, location, and rest position, etc. The geometric characteristics would enable the environment to produce a visual schematic representation of equipment's structure, the detection of collisions with other objects in the cell, and determine the spacial relationship between that piece of equipment and others. A solid geometry model is particularly useful for these purposes.

The communication characteristics describe the means that a specific equipment controller has for exchanging information with other equipment. These vary form digital or analog input/output lines, or more sophisticated protocols such as RS232, IEEE4888, and Ether Net, etc. The communication component of the equipment model captures the set of communication channels present in the equipment, and their parameters, such as voltage levels, limits, rates, and addresses. The sensory components of the model contain information about the equipment's ability to perceive various attributes of its environment using sensors such as pressure, temperature, force, and vision, etc. To fully describe the real-world objects in classes, these characteristics are represented by instance variables within each class.

The dynamic information includes a machine's current status and the time history of the state variables. These are used to coordinate the activities of the components in the cell. It is anticipated that the time history will support the development of error recovering mechanisms for the cell.

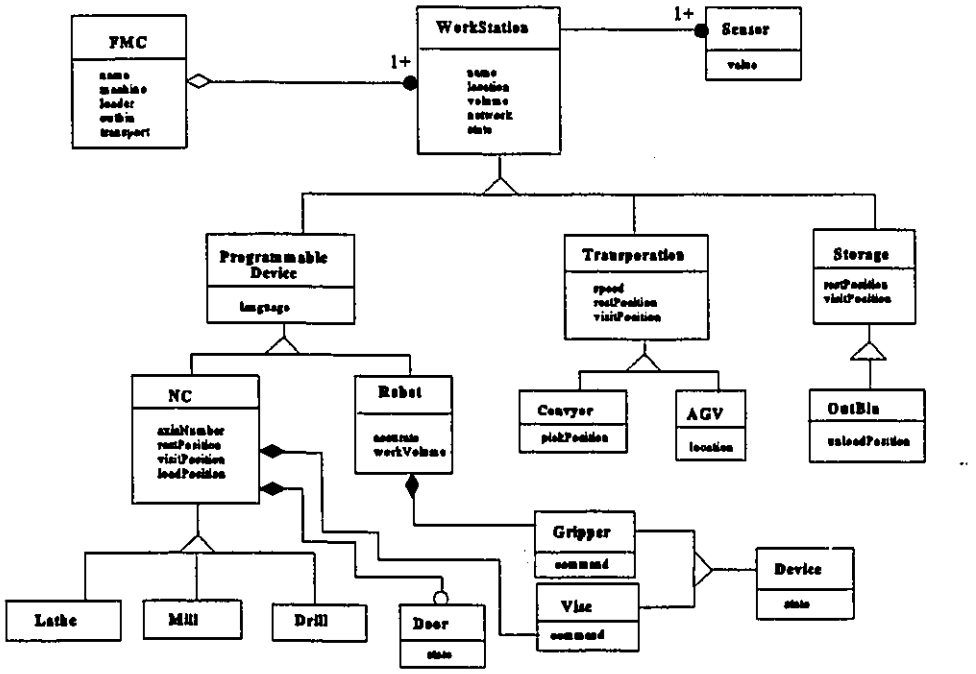


Figure 6.2: Architecture of the UniSet Cell Facility Model abstract

To represent this real-world semantics using an object-oriented model, the generalization, the aggregation, and the composition abstractions are used. Several object classes are defined in the environment to model a UCFM according to a desired manufacturing cell. The architecture of classes corresponding to the objects is shown in Figure 6.2. With reference to Figure 6.2, an instance of the class **FMC** contains many instances of the class **WorkStation** or the subclasses of the class **WorkStation**. Each instance of the class **WorkStation** uses many instances of the class **Sensor** or the subclasses of the class **Sensor**. The relationship between the instance of the class **Robot** and the

instance of the class **Gripper** may be an *association* or a *contain* relationship depended on the robot's configuration. The filled diamond symbol represents this relationship. The other two instances of the class **Vise** and the class **Door**, may have the same relationships to an instance of class **NC** as an instance of class **Gripper** to an instance of class **Robot**.

Details of the important classes of the UCFM are described as follows:

Class FMC, is the abstraction of common characteristics of flexible manufacturing cells. This class describes the relevant properties of a cell through seven instance variables. These are *name, specification, machines, robots, outbins, transports*, and *controller*. The value of the instance variable *name* is a character string that is identified to the user. The value of *specification* represents the size of FMC object. Other values of instance variables are cell constituent objects.

Class WorkStation is an abstract class. All equipment classes constituting an FMC, such as NC, Robot, AGV, etc., are subclasses of class WorkStation. This class includes geometric, communication, and sensory characteristics of its sub classes as instance variables. These instance variables and method related to them are inherited by all the **WorkStation** subclasses.

Class Programmable Device is used to represent machines which can interpret and execute locally stored programs. Each object of the class **ProgrammableDevice** has its own programming language.

6.2.1.2 UniSet Tool and Raw Stock Model (UTRM)

The object-oriented databases for raw stock and tools are built on the basis of the rational of the object-oriented approach to model the entities and relationships among entities.

The database for tools contains a collection of tools available. These tools are grouped into three categories: tools for drilling, tools for milling, and tools for turning. The tool database includes quasistatic and dynamic information. Quasistatic information of these tools pertains to general features, geometry, and material description. The general description includes code, cost, and tool life. With the exception of the code which is used to identity a tool, the tool information is used for choosing the best tool for a given task.

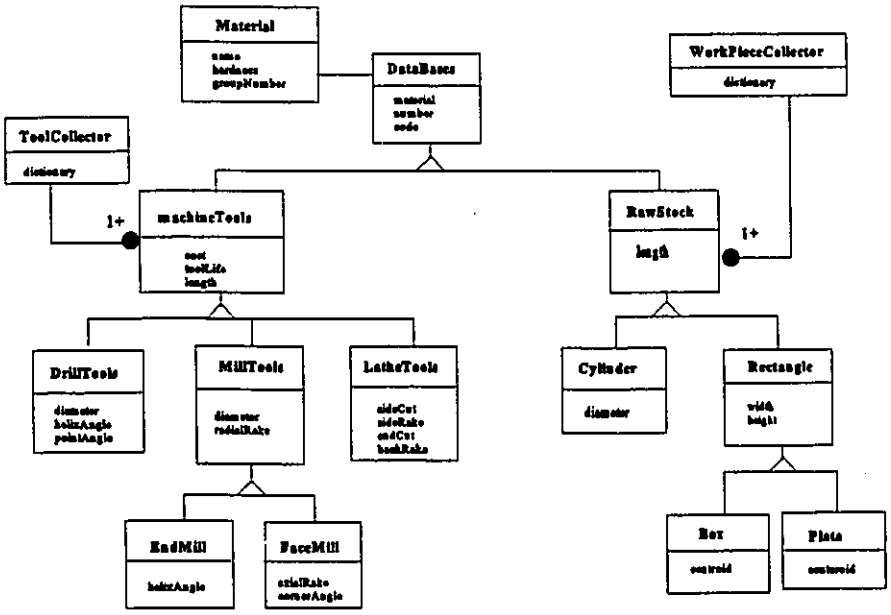


Figure 6.3: Class Hierarchy for the UniSet Tool and RawStock Model

The geometric description represents the physical characteristics of the tool such as a diameter, overall length, etc. Depending on the tool type, tool geometry varies. For example, geometries of tools for turning require descriptions of side and end cutting edge angles, side rake

angle, end relief angle, and back rake angle. These are defined as instance variables within a class. The material description includes hardness.

The dynamic information pertains to such things as the remaining tool life and the number of tools. These are updated after each operation. Figure 6.3 illustrates the class hierarchy for the database.

The Raw stock database is an instance of the class WorkPieceCollector and contains a collection of raw materials available in stock. The instance variable of the class **WorkPieceCollector**, *dictionary*, is responsibility for storing these raw material objects. The raw materials are classified into several types based on shape. The most basic of these is type cylinder and type parallelepiped. Type parallelepiped in turn is classified into two types, box and plate.

The raw materials are also characterized by the following quasistatic information: general, geometric, and material description. The general and material descriptions are similar to those of tools, but the geometric description is very simple and includes only width and length.

The dynamic information carried with a raw material consists of the number of pieces available. This number is updated when parts are purchased or used.

6.2.1.3 UniSet Machining Parameters Model (UMPM)

The optimal machining parameters (depth of cut, speed, and feed rate) are calculated using Back Propagation Neural Networks (BPN). The neural network sub module is available for use by a cell supervisor when new data related to machining parameters become available. Currently, the

UniSet environment provides three neural networks: for turning, for milling, and for drilling operations. These networks have been trained based on standard handbook data [189].

6.2.1.3.1 Description of the Neural Network Model

The multi-layered feed forward BPN, with full interconnection, consists of an output layer, input layer, and one or more hidden layers. Each layer has a set number of nodes that are chosen to fit the problem at hand (see Figure 6.4). Each node (processing element) is a neuron that processes a set of inputs applied either from outside (X_i) or from a previous layer (H_j) to produce the output by applying an activation function, f, to the weighted sum of inputs. The actual output, Y_k, of the BPN with two hidden layers is given by the following form.

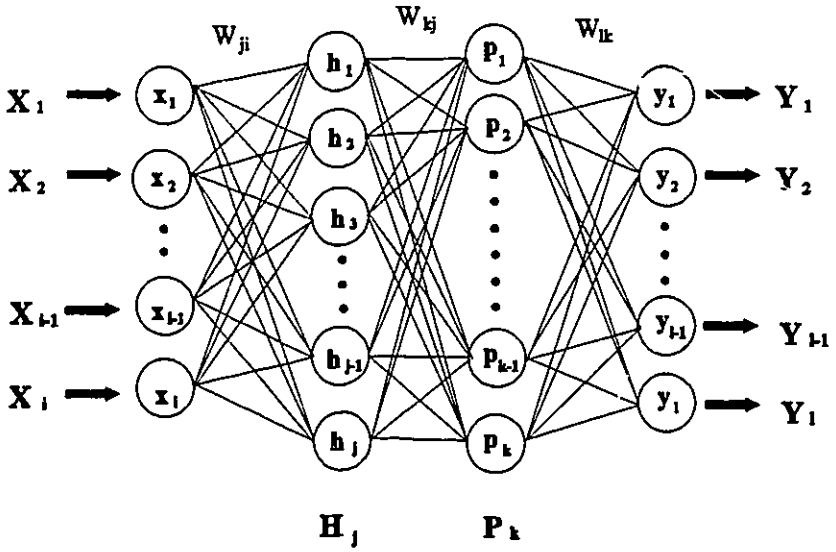


Figure 6.4: Structure of BPN Model

$$Y_l = f(y_l + \sum_k W_{lk} P_k) \qquad (1)$$

where,

$$P_k = f(p_k + \sum_j W_{kj} H_j)$$

$$H_j = f(h_j + \sum_i W_{ji} X_i)$$

The subscripts i, j, k, l represent the number of nodes in the input, hidden, and output layers, respectively. The biases (h_j, p_k, y_l) and connection weights (W) are determined in the same way during the learning of the network with the preselected training input data set. BPN learning

characteristics can be improved by adjusting the dynamic range of neuron output within +0.5 [189]. The improved activation function is given by the following form.

$$Y = f(NET) = -0.5 + \frac{1}{1 + e^{-NET}} \tag{2}$$

and the derivative of Y is given by

$$f'(NET) = (\frac{1}{2} + Y)(\frac{1}{2} - Y) \tag{3}$$

The back propagation learning algorithm is a generalized delta rule in which the connection weights, W(m) at iteration time m, is updated with the general second-order linear stochastic difference equation based on the gradient descent method [186].

$$W(m+1) = W(m) + \eta\ \Delta W(m) + \alpha\ \Delta W(m-1) \tag{4}$$

The η is the learning rate and α is the momentum term to be chosen for guaranteed and fast convergence. $\Delta W(m)$ is obtained by minimizing the summed squared error E_m with respect to W:

$$\Delta W(m) = -\frac{\partial E_m}{\partial W} \tag{5}$$

where,

$$E_m = \frac{1}{2}\sum_p (T_p - Y_p^m)^{\frac{1}{2}}$$

and T_p is the target output for training input pattern p.

Using the chain rule, Equation (5) renders the following for node l in the output layer:

$$\Delta W_{lk}(m) = -\frac{\partial E_m}{\partial Y_l^m}\frac{\partial Y_l^m}{\partial W_{lk}^m} = -\frac{\partial E_m}{\partial Y_l^m} f'(Y_l^m)\, P_k^m$$
$$= [T_l - Y_l^m]\,[\tfrac{1}{2} + Y_l^m]\,[\tfrac{1}{2} - Y_l^m]\, P_k^m \qquad (6)$$

Similarly, for the hidden layers Equation (5) yields:

$$\Delta W_{kj}(m) = -\frac{\partial E_m}{\partial P_k^m}\frac{\partial P_k^m}{\partial W_{kj}^m} = -[\sum_l \frac{\partial E_m}{\partial Y_l^m}\frac{\partial Y_l^m}{\partial P_k^m}]\, f'(P_k^m) H_j^m$$
$$= (\sum_l [T_l - Y_l^m]\, f'(Y_l^m)\, W_{lk}^m)\, f'(P_k^m)\, H_j^m$$
$$= (\sum_l [T_l - Y_l^m]\,[\tfrac{1}{2} + Y_l^m]\,[\tfrac{1}{2} - Y_l^m]\, W_{lk}^m)[\tfrac{1}{2} - P_k^m]\,[\tfrac{1}{2} + P_k^m] H_j^m \qquad (7)$$

$$\Delta W_{ji}(m) = -\frac{\partial E_m}{\partial H_j^m}\frac{\partial H_j^m}{\partial W_{ji}^m} = -(\sum_l \sum_k \frac{\partial E_m}{\partial Y_l^m}\frac{\partial Y_l^m}{\partial P_k^m}\frac{\partial P_k^m}{\partial H_j^m}) f'(H_j^m) X_i$$
$$= (\sum_l \sum_k [T_l - Y_l^m] f'(Y_l^m) W_{lk}^m f'(P_k^m) W_{kj}^m) f'(H_j^m) X_i$$
$$= (\sum_l \sum_m [T_l - T_l^m][\tfrac{1}{2} + Y_l^m][\tfrac{1}{2} - Y_l^m] W_{lk}^m [\tfrac{1}{2} + P_k^m][\tfrac{1}{2} - P_k^m] W_{kj}^m)[\tfrac{1}{2} - H_j^m][\tfrac{1}{2} + H_j^m] X_i \qquad (8)$$

A four layered BPN is used for obtaining machining parameters. The output layer has three (or two) nodes representing parameters (i.e., depth of cut, speed, feed) for the given pairs of the machine operation parameters, in this case tool material, work piece material, geometry, etc., which specifically constitute the number of nodes in the input layer.

6.2.1.3.2 Training Data Generation

The procedure for generating the randomized training input data consists of two steps. In the first step, the lower and the upper bounds on the input variables are defined. Within these limits, the sample set is randomly obtained. These limits are determined by considering the reasonable range of the values for each variable that is likely to occur in the operation of a particular cell based on its constituent machines. For the case of drilling in the demonstration cell, the range of parameter hardness is between 125 BHN and 535 BHN.

In the second step, an input data set of size n from k input variables is formed. As example, there are four inputs for drilling, these are the raw material, the hardness, the tool material, and the hole diameter. Regarding the number of training input data needed for generalization, Ahmad et al. [187] suggest that the input size n in the order of $k^{2/3}$ random patterns is sufficient for a network to learn with high degree of generality. In the drilling case, a sample size of n=90 is used. Some of the data set used for training the BPN is shown in Table 6.1.

Table 6.1 Training input data vectors for drilling

Input vector	Input Variables				Referenced output variables	
	Material	Hardness (BHN)	Hole diameter (Inches)	Tool material	Speed (fpm)	Feed (IPR)
1	1212	125	0.5	M10	125	0.01
2	1213	175	1.0	M10	125	0.018
3	1212	125	2.0	M10	125	0.025
4	1215	125	0.75	M7	130	0.015
5	1108	175	1	M1	120	0.018
6	11L13	225	1.5	M10	135	0.02
7	11L14	125	0.75	M10	130	0.015
8	12L13	125	2	M7	100	0.025
9	1005	150	0.25	M1	90	0.005
10	1012	150	1.5	M1	90	0.018
11	1019	200	0.5	M7	80	0.009
12	1021	200	1.5	M10	65	0.015
13	1030	200	2	M10	75	0.022
14	1040	250	0.75	M7	60	0.01
15	1049	300	1	M33	50	0.12

6.2.1.3.3 Network Learning Result

The network learning is performed with the training data to find out the weight matrix and node biases. In order to cope with the long training time of BPN, some measures for improving efficiency are taken. The input patterns and target outputs are normalized and scaled within the range -0.5 and 0.5. The learning rate (η) and the momentum (α) are adjusted during the training process for speeding up the convergence. Typical values of learning rate are 1.0 to 3.0 for the small training data set size, and 0.1 to 0.3 for the large training data set size. In the case of drilling, a learning rate of 0.2 was used. The values of momentum are kept less than 1.0. The termination criterion is set for a maximum output error of e=0.01. An example of the resultant weight matrix and the biases for the case of drilling is shown in Table 6.2.

Table 6.2 Weight Matrix and Biases for drill operation

Weight matrix

Layer	Node		W_{ji} (i:Preceeding)						
	i	j	1	2	3	4	5	6	7
from input to hidden	1		-0.63548	-6.88472	-2.96061	2.91050	2.50716	-0.52509	1.23056
	2		8.11659	13.75063	-1.47320	1.66996	-8.75082	5.73054	-9.90321
	3		1.61627	-6.37272	0.72570	-5.03440	3.53972	-2.83954	9.18652
	4		0.72026	0.15873	-2.38315	3.18416	-0.47322	-5.98722	3.05997
from hidden to hidden	1		1.37453	-3.63443	-8.86177	-8.96499	-4.26739	-4.20179	-2.45929
	2		1.30533	4.91780	0.90458	-0.32502	-0.06969	-0.00814	2.36585
	3		0.84227	-3.95956	3.87847	-10.10952	3.39331	17.23709	1.15399
	4		0.48097	1.06324	-12.29541	-0.35674	-2.41680	1.14906	5.16952
	5		2.22554	-1.60456	3.73718	-3.28816	0.07215	0.88020	0.10161
	6		-2.56526	-2.19187	-3.09466	3.00954	-1.94243	-3.62429	1.41826
	7		-3.48167	0.55946	-0.00896	2.49217	-2.23343	4.67254	1.94875
from hidden to output	1		-2.24265	4.05617	-4.74141	1.09121	-3.03116	-5.29675	-5.17504
	2		9.27810	-4.81895	-0.18701	-9.79425	2.15580	-3.08881	-3.76751

Biases

Node / Layer	1	2	3	4	5	6	7
hidden 1	0.75763	-3.67060	-5.54493	-1.96854	0.11851	1.41229	-2.18840
hidden 2	1.69916	1.13007	3.45615	-0.69308	-0.42873	-2.34371	-1.92592
output	-0.59779	-7.03206	---	---	---	---	---

The performance of the trained Neural Network for drilling is evaluated to examine how well it predicts the machining parameters. The test data set and the resulting output machining parameter's output values Vnn from the network are compared to the reference values Vref which are used for training as shown in Table 6.3. The table 6.3 shows that the Neural Network is able to predict the machining parameter values within 13%.

Table 6.3 Machining Parameters Responses from the Network on the trained data set

Input vector	Referenced output variables(Vref)		Network output variables(Vnn)		Ratio (Vnn/Vref)	
	Speed	Feed	Speed	Feed	Speed	Feed
1	125	0.01	125.1	0.011253	1.01	1.12
2	125	0.018	125	0.0167	1	0.93
3	125	0.025	124.2	0.025213	0.99	1
4	130	0.015	126.7	0.0139	0.97	0.93
5	120	0.018	126.9	0.01553	1.05	0.87
6	135	0.02	128.7	0.020914	0.95	1.04
7	130	0.015	124.6	0.0132	0.96	0.88
8	100	0.025	101	0.02121	1.01	0.85
9	90	0.005	88.12	0.005574	0.98	1.11
10	90	0.018	88.12	0.018	0.98	1
11	80	0.009	72	0.007687	0.9	0.85
12	65	0.015	72.6	0.01685	1.12	1.12
13	75	0.022	68.76	0.02	0.92	0.91
14	60	0.01	58.9	0.009018	0.98	0.90
15	50	0.012	43.72	0.01293	0.87	1.08

6.2.2 UniSet Cell Operation Model (UCOM)

The system does not have any task programmed because there is no logic to guide the succession of events. In this module, the environment uses the cell program to complete the UniSet Cell Operation Model (UCOM), including the cell process knowledge. Cell processes are represented by a Task Initiation Diagram using an object-oriented approach. An automatic system has been developed to construct rules to control the sequence of events in a given manufacturing cell. The operation model developed by this automatic system can be simulated and evaluated before implementation.

6.2.2.1 Task Initiation Diagram

Many of the approaches that have been developed so far, and reported in the literature, to represent the cell activities are adaptations of ones used in digital logic or electric circuits. The most common of these are petri nets and their derivatives. In these approaches, the cell or machine states are considered primal. The state of the cell or one of its constituents changes as a result of the firing of an event. This approach is not suitable for control or verification of flexible manufacturing cells

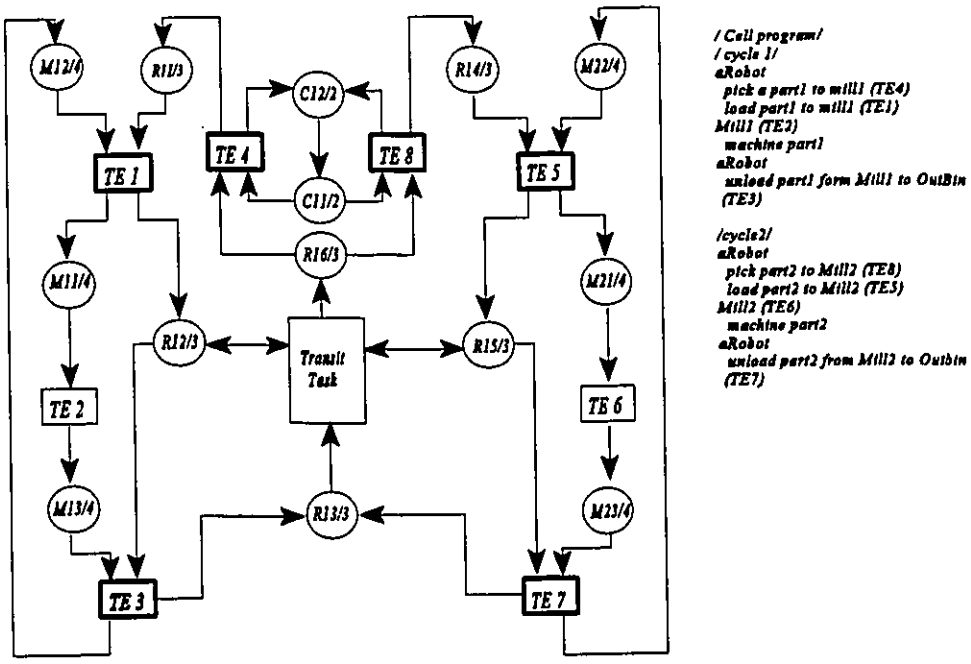


Figure 6.5: Example of Task Initiation Diagram

or systems. For such systems, the task to be performed, rather than the state, must be regarded as primal. A task is executed when a particular set of conditions is met.

The methodology developed for UniSet regards the tasks to be performed by the cell or any of its constituent machines for being primal. Sensory signals indicating the change of state of

machines are used to trigger or initiate tasks. A task may be simple and require a relatively short time to execute, or may be complex and lengthy. The methodology developed here may be depicted by a set of diagrams called "Task Initiation Diagrams" and their accompanying rules

Conventions and Terminology used for the Task Initiation Diagram

The task initiation diagrams are multi-layered. The topmost layer shows the dynamic behaviour of the task-level program of the cell. Figure 6.5 shows an example of a Task Initiation Diagram and corresponding cell program. For conciseness, states and tasks are cryptic instead of verbose. Where a symbol is chosen, its verbose description is displayed in a State Description Pane of the GUI. The task initiation diagram consists of two components: tasks and states. Tasks in the task initiation diagram would generally consist of a concerted group of subtasks or operations involving more than one constituent of the cell, and in such a case are termed composite tasks. These are shown by the framed boxes in the diagram. As an example **TE1** (load part1 to Mill1) involves concerted operations involving the Machine and the Robot. Tasks involving a single machine are called simple tasks and are shown by a box, e.g., **TE2** (machine part).

The cell state is given at any instant by the collection of states of its constituents. For initiation of a given task, a composite state consisting of a subset of the cell state needs to exist. These composite states are shown in the Task Initiation Diagram by ellipses, e.g., *R11/3* or *M13/4*. The last number of the symbols indicates how many individual states are required to determine this composite state. All state descriptions are listed in Tables 5.3, 5.4, and 5.5. A Task level UniSet

program does not include explicit information about the states necessary to initiate the tasks. This is because such information is embedded in the implementation of each task.

Structure of Task Initiation Diagram

Task Initiation Diagrams are composed of two basic components: a set of Rest states S_R and a set of tasks T. Tasks, in turn, are classified into three groups: the cell configuration dependant task (T_d), the cell configuration independent task (T_i), and the cycle transit task (T_t). T_t tasks are used for the transition from one cycle to another. These are tasks derived automatically by the system in order to complete a production job, thus, a programmer does not need to create them.

To complete the diagram, it is necessary to define the relationship between the states and the tasks. This can be done by specifying two functions connecting states to tasks: the condition function C, and the output function O. The condition function C defines, for each task T_i, the set of states for the task $C(T_i)$. Some condition functions may use guiding parameters in addition to a set of states. As an example, in the listing below, C(TT) uses a Remaining Processing Time (RPT) to cause transition to the desired state. The output function O defines for each Task T_i the set of output States for the transition $O(T_i)$.

These four terms define the structure of a Task Initiation Diagram. Formally, a Task Initiation Diagram TID is defined as the four-tuple TID=(T, S_R, C, O).

Consider the example in figure 6.5, the components of the structure are given below:

TID = (T, S_R, C, O)

T = {TE1, TE2, TE3, TE4, TE5, TE6, TE7, TT_i}

S_R = {Mnm/4, R1k/3}

n =1,2 m=1,2,3 k=1,2,3,4,5,6,7

C(TE1) = {M12/4, R11/3}	O(TE1) = {M11/4, R12/3}
C(TE2) = {M14/4}	O(TE2) = {M13/4}
C(TE3) = {M13/4, R12/3}	O(TE3) = {M12/4, R13/3}
C(TE4) = {C11/2, R16/3}	O(TE4) = {C12/2, R11/3}
C(TE5) = {M22/4, R14/3}	O(TE5) = {M21/4, R15/3}
C(TE6) = {M21/4}	O(TE6) = {M23/4}
C(TE7) = {M23/4, R15/3}	O(TE7) = {M22/4, R13/3}
C(TE8) = {C11/2, R16/3}	O(TE8) = {C12/2, R14/3}
C(TT1) = {R12/3,M22/4},[RPT(TE2)>Δt]	O(TT1) = {R16/3}

6.2.2.2 UniSet Cell Operation Model (UCOM)

A manufacturing cell executes a job during a certain time. A job consists of parts, and operation knowledge which contains information to complete the parts. Each part comprises a material and a cell program. Since a cell program can be developed in different level languages, it should be transformed at the outset into a machine's native language format. Depending on the translation required, a different submodule is used. UCOM provides a model for the complete information required to carry out a cell task.

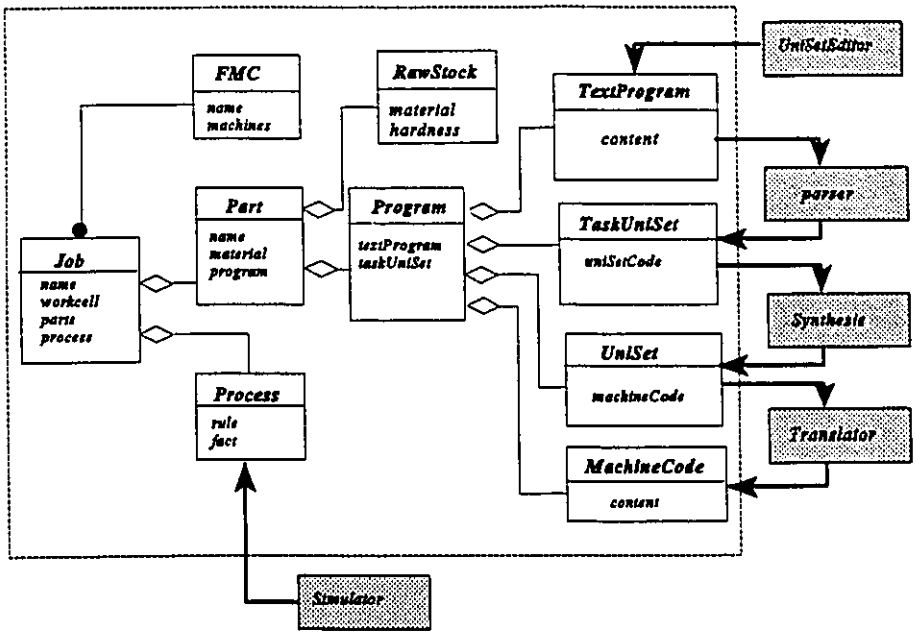


Figure 6.6: Structure of Classes in the UniSet Cell Operation Model

The architecture of the class hierarchy of the UCOM and its relationships with the modules is shown Figure 6.6. An instance of the class **FMC** may have many instances of the class **Job**. The object **Job** consists of an object **Part** and an object **Process**, which is needed to process on the object **Part**. Details of each class are explained as follows:

Class Job

Class **Job** is the abstraction of the common characteristics of jobs carried out in a manufacturing cell. Four instance variables are defined to describe a class **Job**, the job *name*, the *workcell*, the *parts*, and the *process*. The value of the *name* is used to identify the job. The value of the *workcell* represents a work cell in which the job is to be performed. A work cell operates one

or more parts simultaneously. The value of the *parts* contains the list of parts to be performed at a given time. The process knowledge for a job is captured in the value of the *process*.

Class Part

An instance of class **Part** should be able to provide detailed information about a part, including its name, material, and required cell program. To meet these requirements, three instance variables are defined in class **Part**. They are *name*, *materia*l, and *program*.

Class Program

An instance of class **Program** is completed through the programming module. Class **Program** has two instance variables, *textProgram* containing an edited program, and *taskProgram* representing collections of instances of class **TaskLevelUniset**. Each instance of class **TaskLevelUniSet** contains an equivalent sequence of instances of Class UniSet and in turn an Instance of class UniSet contains translated native code.

Class process

Class **Process** is described by two instance variables, *fact* and *rule*. An instance of class **Process** works like the knowledge base in expert systems. The value of the *rule* contains production process rules. The value of *fact* represents the state of a cell. As illustrated in Figure 6.6, an instance of class **Process** is completed through simulation/compilation module and used in the control module.

6.3 Hybrid Knowledge Representation

The UniSet Environment allows for the inclusion of various methodologies of knowledge and experiences concerning the operation of a Flexible Manufacturing Cells. Many methodologies

for a knowledge representation have been developed by AI researchers. Each of these methodologies has advantages and disadvantages with respect to a specific application.

Hybrid schemes, which make it possible to integrate the different knowledge bases into locally ordered structures within one system are used here to encapsulate knowledge in the UniSet Environment. Figure 6.7 illustrates the structure of the hybrid knowledge representation. The conventional procedural programs incorporate this knowledge into the instance methods of the various object classes, and groups these methods according to their application domains.

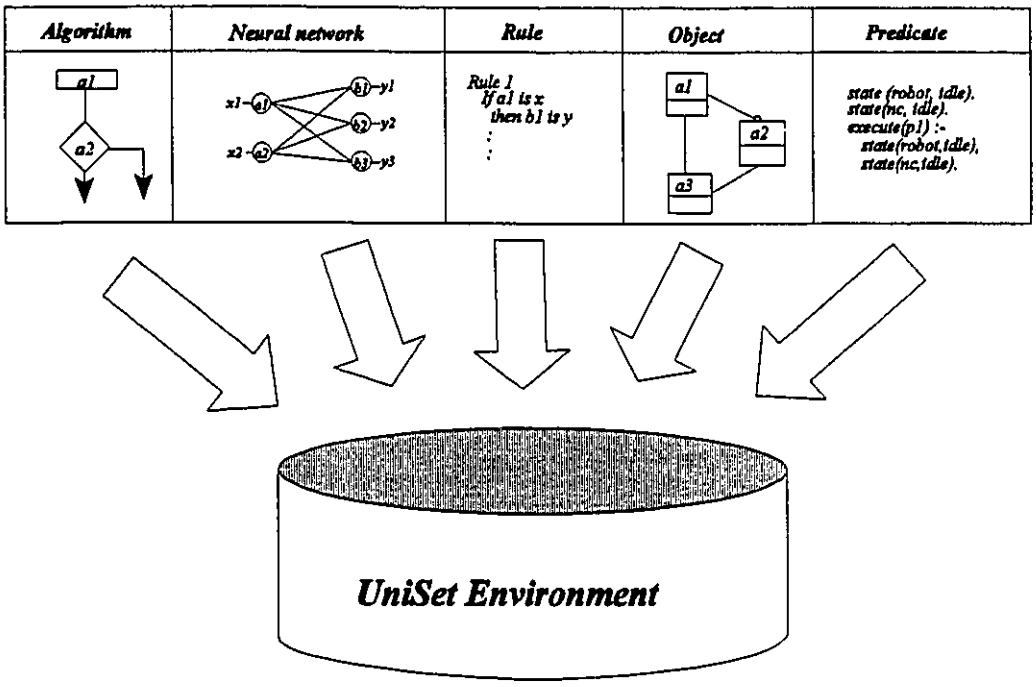


Figure 6.7: Hybrid Knowledge Representation in the UniSet Environment

The production rules, which contain a form of **IF**<presumption> **THEN**<conclusion>, are attached to a knowledge base to a class, such as Tool selection knowledge base, etc. To select machining parameters, a neural networks has been implemented. All information about entities comprising a cell are expressed in the form of objects. Parsing mechanism of the cell program is implemented in Prolog.

6.4 UniSet System Components Visibility of the UniSet Environment

Users are considered to be outside of the system and are simply the agents that trigger many of the events external to the UniSet Environment. A study of the role of users leads to the realization that there are actually three kinds of users of interest here: the cell operator, the cell supervisor, and the environment developer.

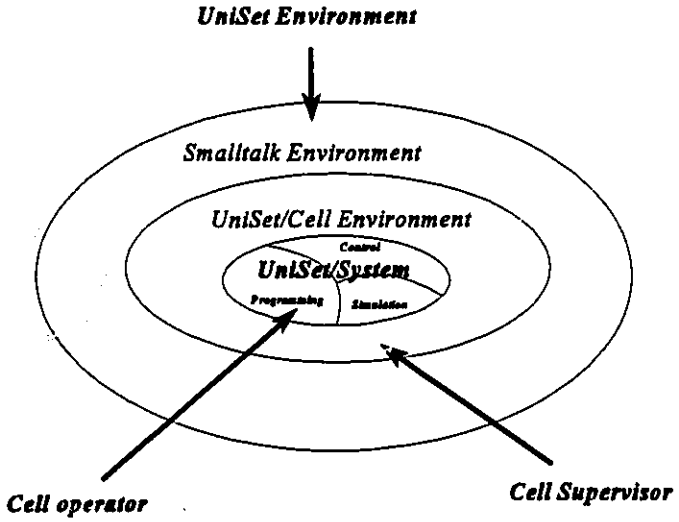


Figure 6.8: Diagram of User Accessibility

The visibility of the system components to these users is illustrated in Figure 6.8. It is useful to separate users into these three categories because their roles are fundamentally different. The cell operator can access the UniSet system, which includes the programming, simulation, and control modules. This allows the cell operator to manipulate the UCOM. The cell supervisor accesses the UniSet/Cell Environment, including the cell facilities configuration and the cell, UCCM. Because the UniSet Environment is organized around class representations of the objects in a manufacturing cell, a developer who is an expert in both the domain of FMC's and object oriented programming software, can extend the functionality of the environment by creating or modifying classes of objects using the Smalltalk Environment.

6.5 Methodologies in Designing and Developing the UniSet Environment

New objects and classes of objects are created by a process known as design. Existing objects and collections of objects can be understood through a collateral practice known as analysis. These two processes, analysis and design, should be required to solve the practical engineering problems. Bourne [188] examines these two fundamental engineering practices in the context of object-oriented methodologies and defines them as consisting of: ways of thinking about and understanding the world as collection of objects and programming methods for actually creating operable abstractions of objects in the world. These two methodologies can not be thought and considered

separately. The methodology for the development of UniSet Environment is the combination of these methodologies, called the object-oriented approach.

This approach consists of identifying objects and classes, identifying their characteristics and the relationships between them. Through this thesis, the Object Modelling Technique (OMT) [190] as a design specification is used to model the UniSet environment. It combines three views of modelling a system: the object model, the dynamic model, and the functional model. The object model represents the static, structural, and data aspects of a system. The dynamic model represents the temporal, behavioural, and control aspects of a system. The functional model represents the transformation, and functional aspects of a system. The three kinds of models separate a system into orthogonal views that can be represented and manipulated with a uniform notation. The different models are not completely independent- a system is more than a collection of its independent parts- but each model can be examined and understood by itself to a large extent. Detailed notations and terminologies are described in Appendix A.

CHAPTER 7
SETUP MODULE

The setup module is a protected module accessible only to the cell supervisor. This module is used to capture the information and build related object-oriented data models which are prepared for use by the subsequent modules (cell programming, simulation, and control). These models include the UniSet Cell Facility Model, the UniSet Tool and Rawstock Databases, and the UniSet Machining Parameters Model. User interfaces in the UniSet Environment have been developed based on the "Model-View-Controller" paradigm to accomplish the task of building the various cell model.

7.1 Model-View-Controller Framework

The Model-View-Controller (MVC) framework provides mechanisms for the development of the user interface architecture in the UniSet Environment. Figure 7.1 provides a conceptual framework for organizing a description of the MVC. The fundamental idea is to separate the underlying semantic information of a problem (the model) from the various ways of presenting the information to a user (the view). Also the interactive aspect of the problem (the controller) is distinguished from the data relationships implicit in the model and its views.

The **model** describes the underlying information and computations within the domain of interest. It is usually meaningful in the real world and would be understandable by a domain expert such as a manufacturing engineer, or machinist.

A **view** is an external format for presenting or visualizing model information. There are usually many different possible views of a piece of model information. There are different formats of presenting the information, such as an FMC layout. A *view* is a projection of its model, that is, it may select some of the model data and suppress the rest. Each view shows different aspects of the model and organizes the information differently. It may take several views (concurrently or over time) to see everything. Views can be text based as well as graphical. A table of numbers, names, or symbols is as much a presentation format as a picture.

A user controls an application by interacting with views. A **controller** is an object that receives events from outside actors, such as users or external devices, and translates them into operations on other objects, such as views and models. A controller interprets input events according to the current control context of the application, called control state of the controller.

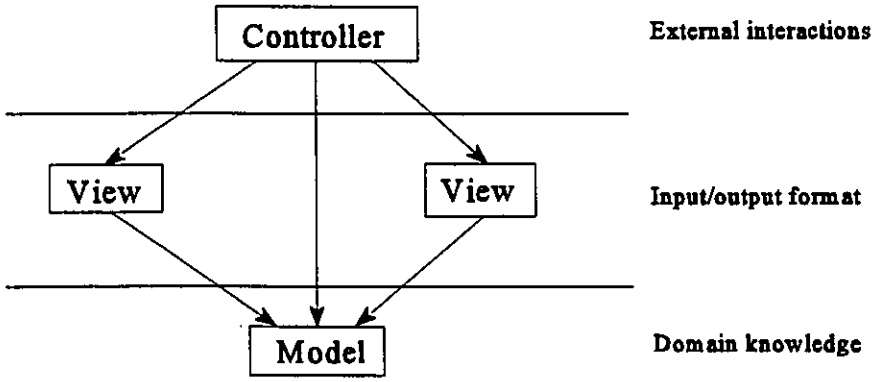


Figure 7.1: Model-View-Controller Framework

All models included in the UniSet Environment are presented in Chapter 6 and have been developed as a result of studies of a manufacturing domain. The remaining task is to show how views and controllers are designed according to the related models. The following sections and Chapters 8 and 9 will deal with these design aspects.

7.2 UniSet Configuration Editor

This editor is responsible for configuring **aFMC**, that is, building the UniSet Cell Facility Model. The Configuration editor lets a user manipulate objects (instances of the class or subclasses of **Workstation**) in the model by editing their visual representations in a syntactical and semantical consistent fashion.

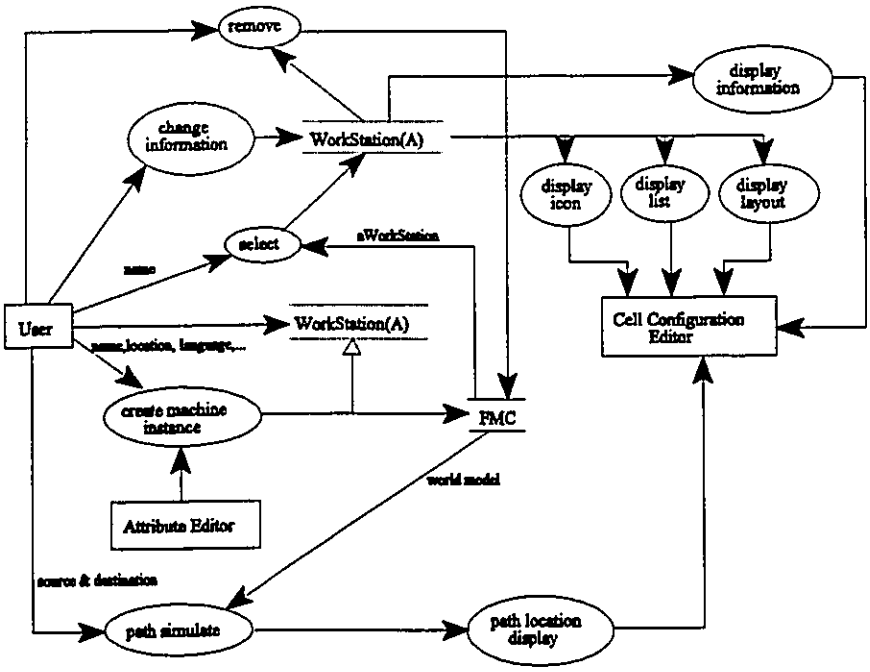


Figure 7.2: Functional Model for the UniSet Configuration Editor

The functionality of this editor is captured in Figure 7.2. There is an input actor: the *User*, who performs the configuration of **aFMC**. The output actor is the **Cell Configuration Editor**, responsible for displaying the information of **aFMC**. There are two locations where data is kept: the **FMC** data store, which contains a collection of instances of a **WorkStation** subclass, and the **WorkStation** data store, which is a temporary storage for one instance of **WorkStation** while it is being created or manipulated.

The processes in the diagram can be divided into three kinds: model building, display generation, and path simulation. The model building processes are; a) *create instances*, which creates a new machine and stores it in **aFMC** data store; b) *change information*, which modifies the

selected machine; c) *select*, which selects the desired machine view; d) *remove*, which deletes the selected machine from **aFMC** database.

The display processes are *display information*, *display icon*, *display list*, *display layout*, and

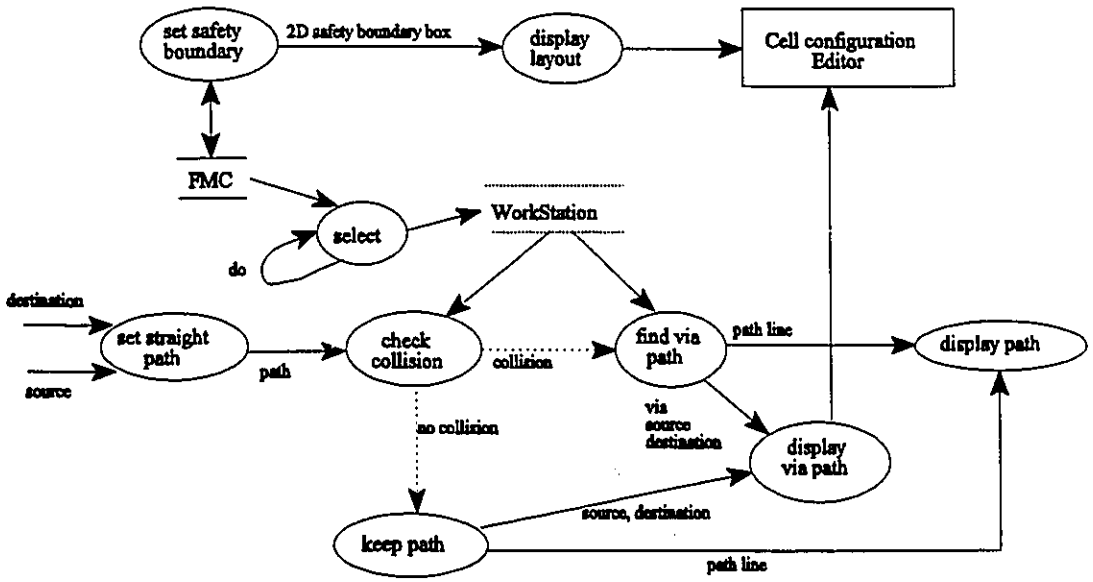


Figure 7.3: Functional Model for the Path Simulation Process

path location display. These processes convert the machine parameters and **aFMC** information into appropriate formats for views on the editors.

The *path simulate* process incorporates geometrical computations. It is responsible for finding a safe robot path between two machines. The Functional Model for this process is shown in Figure 7.3. A machine entry in the database contains the geometry of the machine as a set of polygonal surfaces. The **FMC** data store holds spacial information about the cell layout. The safe boundary of each machine is computed and displayed by considering a physical configuration of a robot.

The straight path calculation must be made relative to the destination and source location. The *check collision* process evaluates the possibilities of collision occurrence for this path within the world model. If there is no collision, this path is accepted as a final robot path. If there are possibilities of collision with other machines, the *find via path* process finds a collision avoidance path by defining a via point within the world model. The result of these processes is displayed as graphical and textual views. For example, the path is displayed as a series of lines and all pertinent locations (e.g., destination, via, etc.) are displayed as a vector of position and orientation with respect to both the world coordinates and the robot coordinates.

To satisfy the above functionality, the UniSet Configuration Editor window is designed as shown in Figure 7.4. The user is provided with icon-based graphical modelling capabilities, as shown in that figure. This allows the user to generate two representations of the system at the same time: (1) external (visual) representation, as a real-like icon-based picture on the screen, and (2) internal (logical) representation, as a set of objects representing corresponding system components, which comprise the UniSet Cell Configuration Model.

The modelling process starts with the defining general characteristics of the cell. A particular component is modelled by activating the corresponding interactive button. The user is asked to define the characteristics of the component. This editor places an icon and symbol, representing that component, on the screen in the proper location. The symbols and their spacing are scaled according to the physical dimensions. To facilitate the editing process, various panes and buttons are designed and arranged on the UniSet Configuration editor.

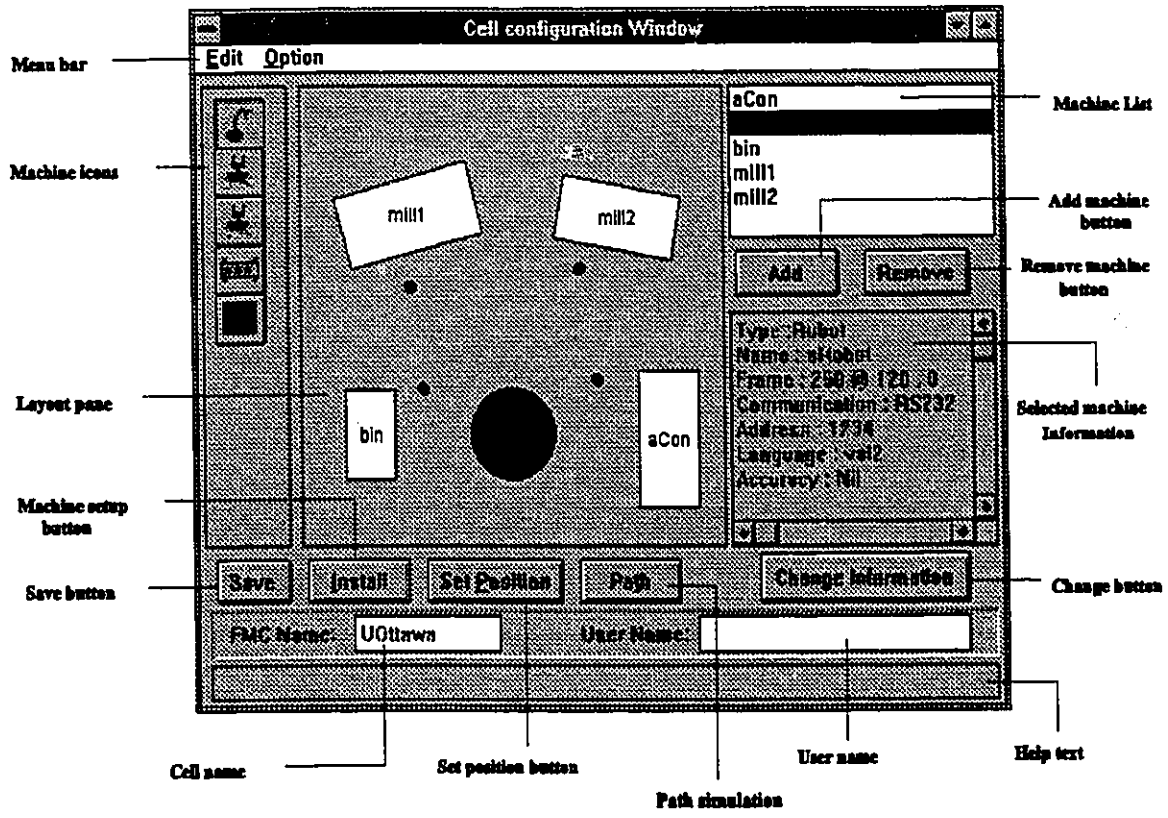


Figure 7.4: User Interface for the UniSet Configuration Editor

The functionality of the panes and buttons are represented in Figure 7.4 with an example cell (UOttawa). As seen in Figure 7.4, the example cell consists of two milling machines (Mill1 and Mill2), a robot (aRobot) for material handling, parts buffer (bin), and conveyor (aCon) for part delivery. The physical layout and detailed information for each component are shown in **Layout pane** and **Selected Machine Information Pane**, respectively.

The UniSet Configuration editor functionality is encapsulated in **CellConfigurationEditor** class. Design decisions for the UniSet Configuration Editor are captured in Figure 7.5 which shows the Object Model. The class **CellConfigurationEditor** has a single instance that provides all the

views, such as buttons and icons, to the user, and organizes the user input.

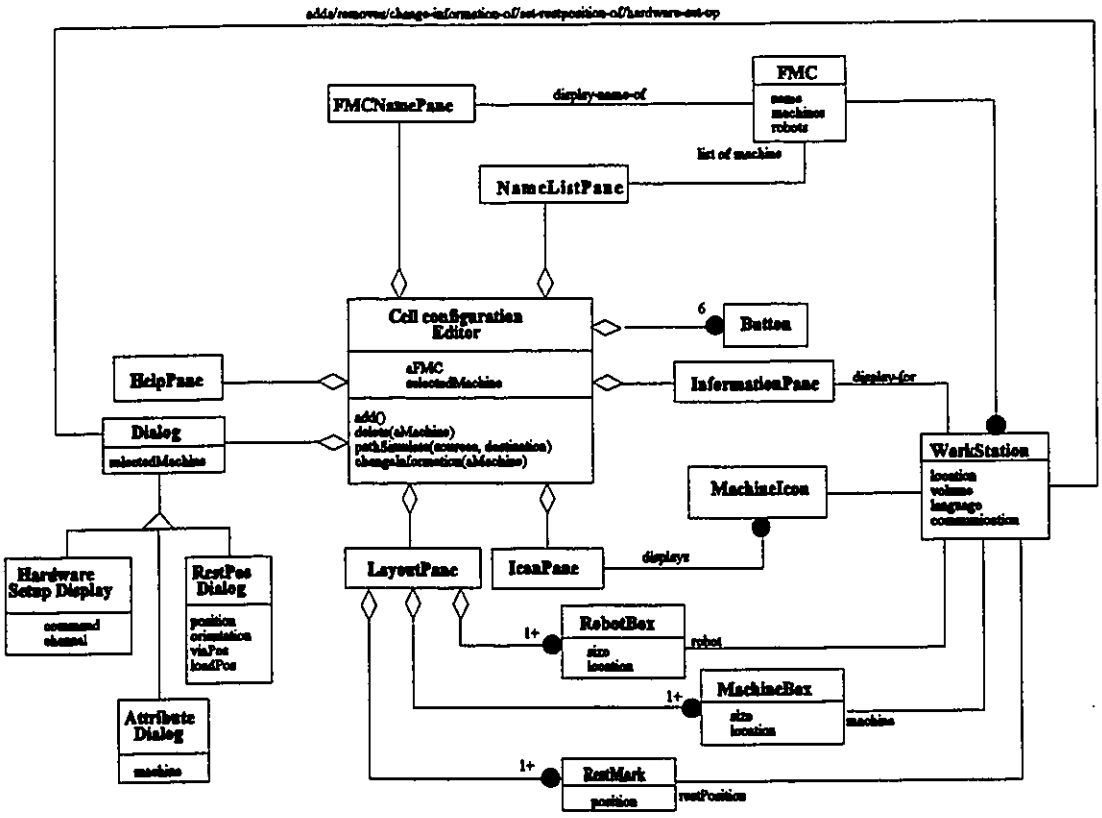


Figure 7.5: Object Model for the UniSet Configuration Editor

This editor has three subeditors. The **Attribute Dialog**, which defines attributes of a cell component, the **HardwareSetup Dialog**, which is used for defining installation specific commands for a cell component, the **RestPos Dialog**, which defines important locations of a cell component such as loading refernce frame, via frames, etc. Other dialogs are provided for adding, deleting, and changing a machine's information.

The three graphical objects, **RobotBox**, **MachineBox**, and **RedMark**, are used by the user to display logical objects, in the layout pane, that represent instances of robots, machines, and rest positions respectively. Each instance of **WorkStation**, for example **aRobot**, has its icon instance

of the class **MachineIcon**.

The Dynamic Model for the UniSet Configuration Editor represents how external events, such as button clicks, keyboard entries, are converted into operations on views and hence into operations on the UniSet Cell Configuration Model. This top level Dynamic Model is illustrated in Figure 7.6.

As shown in the figure, there are six important states that the UCCM can be in; *Displaying*, *Updating FMC*, *Information change*, *Selecting*, *New machine instance creating*, and *Path simulating*. The initial and final states for the editor are *Displaying*. In this state, all the views for the selected machine are displayed in the appropriate panes. The transformations between the two states can follow a multitude of paths depending on events controlled by the user.

The state *New machine instance creating* is triggered by the event **add machine** in the state *Displaying*. This state includes two sequential sub-states, the state *Selecting class* and the state *filling in attributes*. The return to the state *Displaying* is triggered by either of two events; **cancel**, which causes the UniSet Configuration Editor to abandon all actions taken in the state *New machine instance creating*, and **done** with an argument *machine*, which updates the FMC.

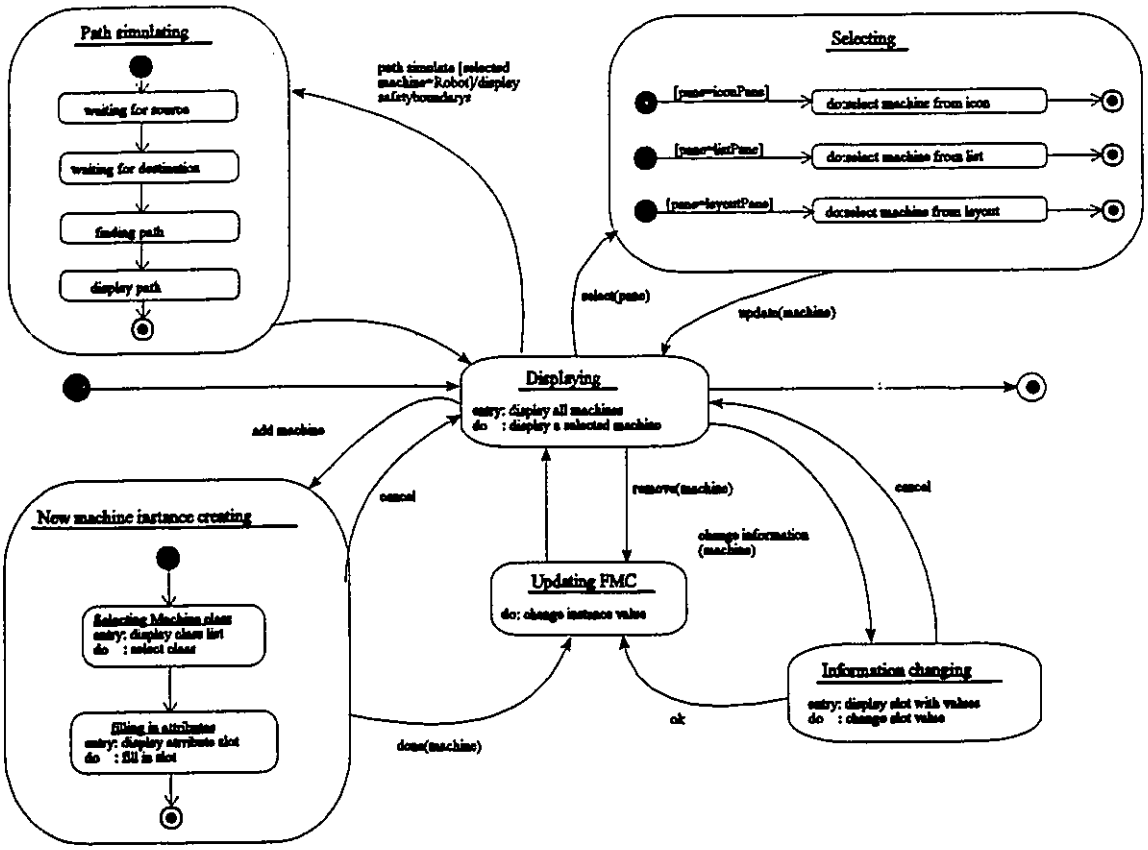


Figure 7.6: Dynamic Model for the UniSet Configuration Editor

The event **change information** with an argument *machine* triggers the state *Information change*. In this state, all the information pertaining to a selected machine can be modified. Again here, the event **cancel** causes a return to *Displaying* and abandonment of the changes, and the event **ok** would cause a return to state *Display* after updating the FMC. The change to state *path simulating* requires an event *path simulate* and a condition that the selected machine is a robot. At all times one machine has to be selected. The state *selecting* has three parallel substates depending on the argument of the event **select**.

7.3 Tool and Rawstock Databases Editors

These editors are used for building the tools and workpieces databases (called collects in the Functional Model terminology) needed for the machining operations in the FMC. The two editors, **Tool Database Editor** and **Workpiece Database Editor**, are very close in both functionality and operation. Figures 7.7 and 7.8 show the Functional Model for the two editors. Due to their similarity, only the **Tool Database Editor** will be discussed.

Tool data is held in the data store **Tool** and the database **ToolCollect**. The data store **Tool** holds the characteristics of a tool object. The **Tool Collect** holds all tool objects specified by a user.

The important processes are *add Tool*, *select*, *remove*, and *change*. The *add Tool* process guides the

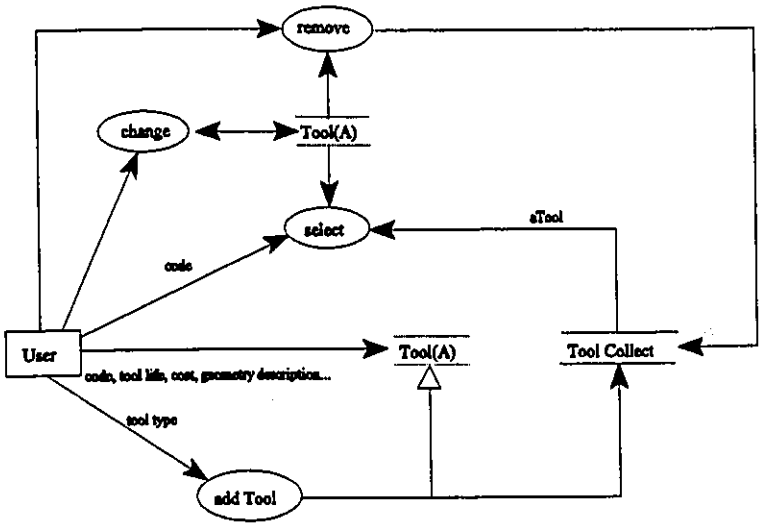


Figure 7.7: Functional Model for the Tool Database Editor

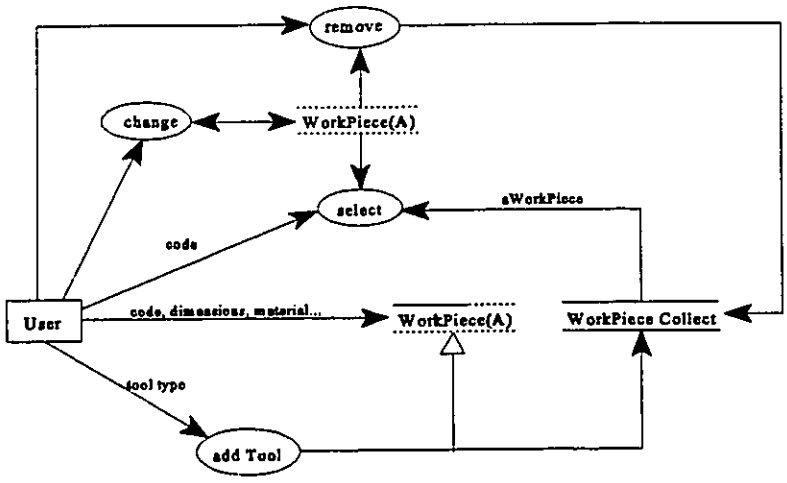


Figure 7.8: Functional Model for RawStock Database Editor

editing of a new tool pertinent information (code, geometry, material, and tool life), the result of which is a new tool added to those in **Tool Collect** database. The tool code is used to select a tool from the **Tool collect** database. A selected tool object is temporarily retrieved to the data store **Tool**, where its characteristics can be modified. A selected tool object may be deleted from the database.

The user interfaces for the two editors are illustrated in Figures 7.9 and 7.10. As shown in the figures, all the pertinent information is displayed as text. A selected tool information is also displayed as radio buttons. Two buttons are provided to add a new object to the database and to remove an existing object from the database.

Design decisions for these editors are shown in Figures 7.11 and 7.12. Since these editors have a rather simple and straightforward user interaction mechanism, their Dynamic Models will not be shown or discussed here. Again for brevity, only the Object Model for the tool database editor will be discussed.

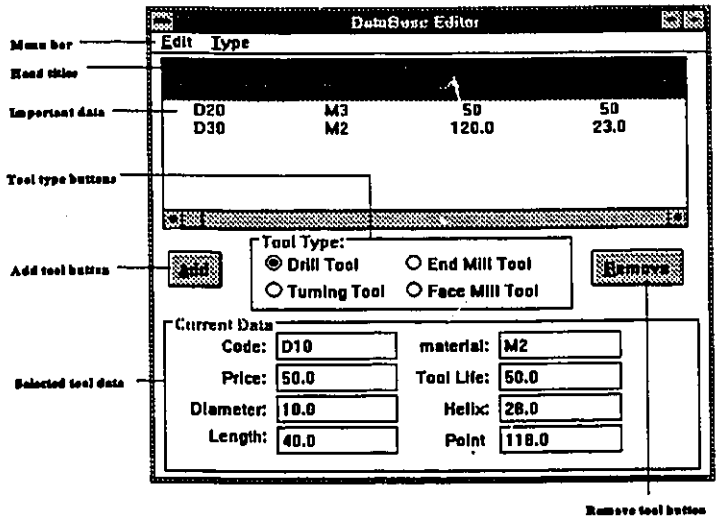


Figure 7.9: User Interface for the Tool Database Editor

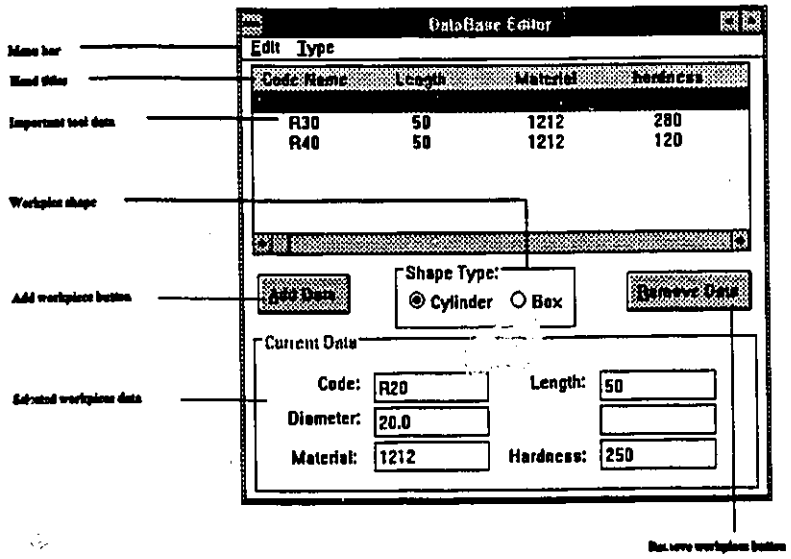


Figure 7.10: User Interface for the RawStock Database Editor

With reference to Figure 7.11, the class **ToolDatabaseEditor** has a single instance that organizes and constructs the instances of the class **Tool** or subclasses of **Tool**. Four radio buttons are implemented so far. These are for drills, turning tools, end mills, and face mills. **Datapane** is used for displaying all attributes of the selected tool. With reference to Figures 7.9 and 7.11, eight

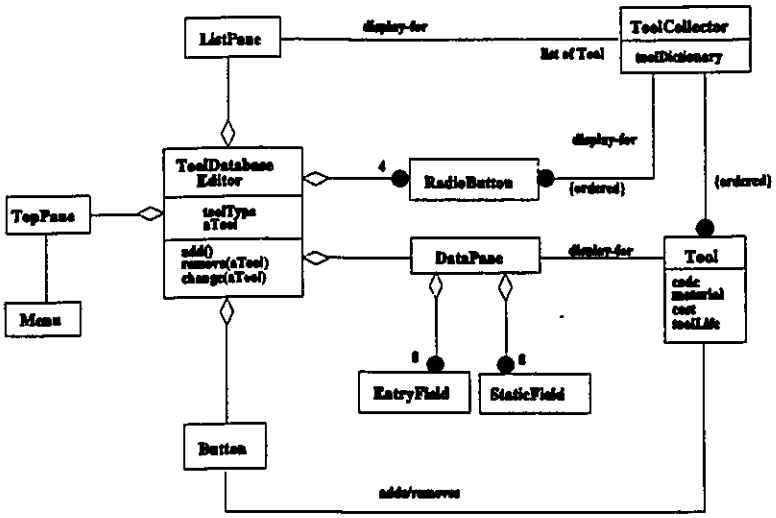


Figure 7.11: Object Model for the Tool Database Editor

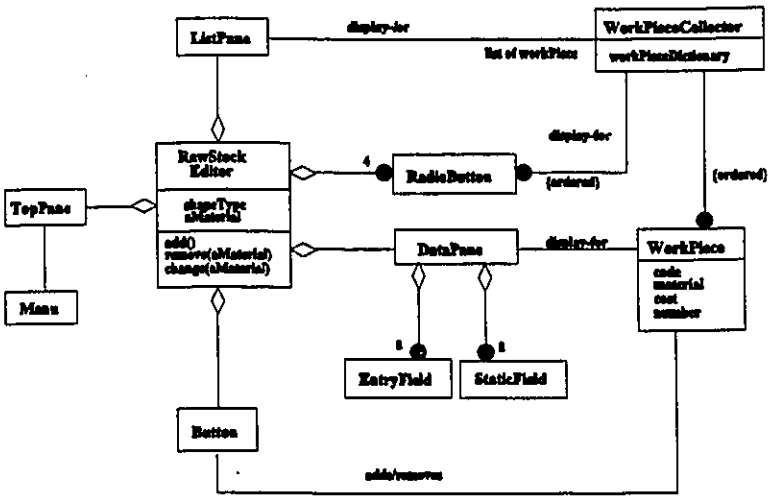


Figure 7.12: Object Model for the RawStock Database Editor

entry fields and eight static fields are provided. Furthermore, two buttons are provided for removing tool objects from the database, and for adding new ones.

7.4 Neural Networks Builder

Generally it takes time to train BPNNs. So far three BPNNs have been implemented in the UniSet Environment. If the design of the environment is such that any of the neural networks are trained within it, the UniSet Environment could not be used for any other purpose during the training period. For this reason, the design is such that neural networks are trained outside the UniSet environment. After being trained, the structure of the neural networks is imported into the Environment and this Neural Network Builder reconstructs the neural network.

Figure 7.13 captures the design decisions for his editor as a Functional Model. With reference to the figure, the data store **External trained Neural Network** holds the bias values and the weights for externally trained networks in the form of a data stream. The *load neural network*

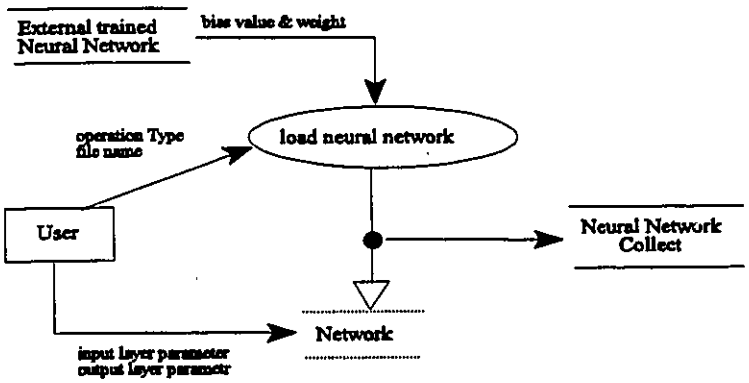


Figure 7.13: Functional Model for the Neural Network Builder

process imports these data into the environment. These data are used to create a network object in the transient store **Network**, which is then added to the **Neural Network Collect** database.

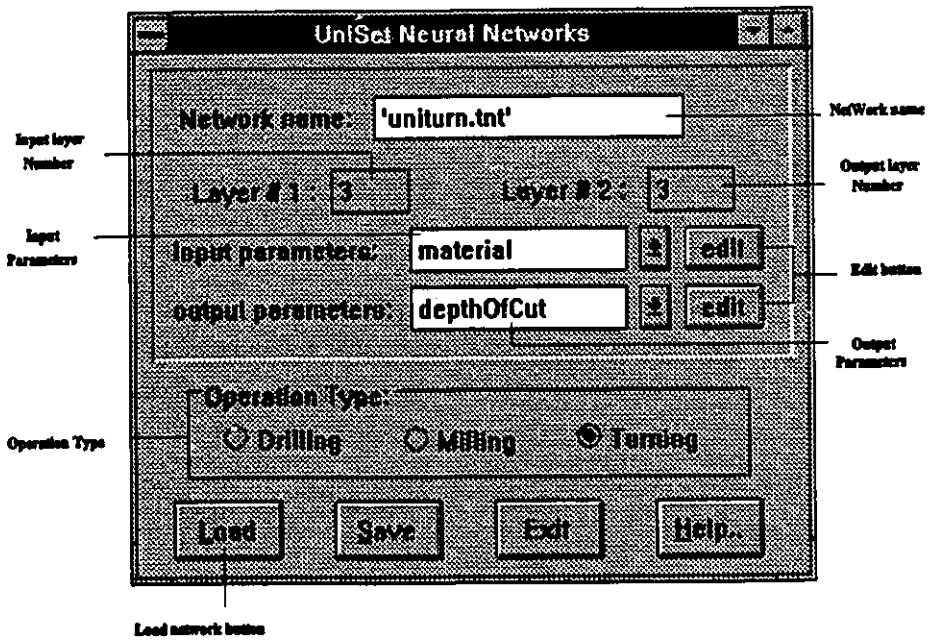


Figure 7.14: User Interface for the Neural Network Builder

The user interface and the Object Model for the Neural Network Builder are shown in Figures 7.14 and 7.15. With reference to the figures, The Instance of the class **NeuralNetworkBuilder** constructs neural networks corresponding to the different machining operations. So far three networks for drilling, milling, and turning have been implemented. Three radio buttons are associated with these networks. The editor also contains the subeditor **NetworkLoader**. This subeditor is used to load the data for externally trained neural networks from file streams. The input parameter and output parameter panes are used to display input parameters and output parameters, respectively, for the selected neural network. The user interface for this

editor is simple in structure. Consequently, its Dynamic Model is rather straightforward and will not be shown.

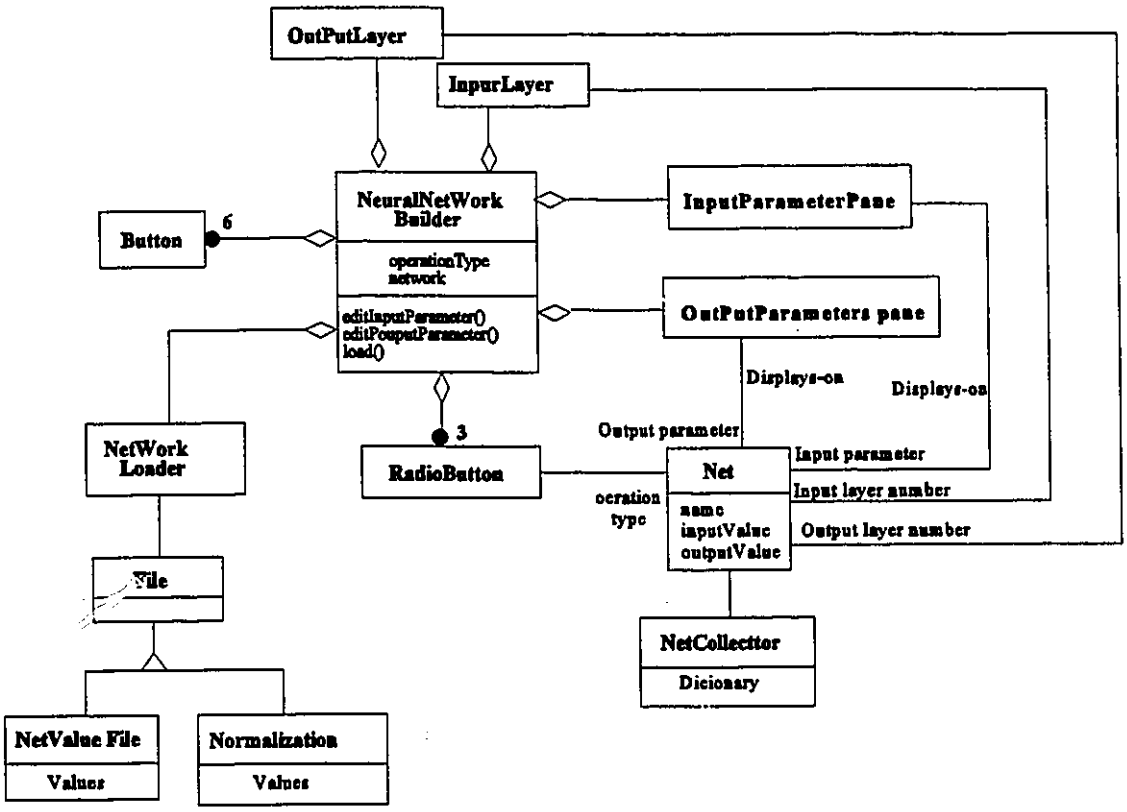


Figure 7.15: Object Model for the Neural Networks Builder

CHAPTER 8
PROGRAMMING MODULE

This module is used for developing cell programs. A cell program contains part programs and operation programs. The latter are motion sequences of different devices under normal conditions. Programming an FMC is an inherently knowledge-intensive task, because the constituent devices may include programmable equipments which use different machine languages, and non-programmable equipment such as feeders, which may require logic signals to activate them.

In this module, the cell program can be created in any combination of basic level UniSet, in Task level UniSet, and in native languages of the cell constituents. If users develop a program using Task level UniSet, device-specific details about the cell equipment are hidden from users by a virtual device control layer which translates generic commands into device-specific ones. The program development module and the automatic generation module will assume the task of carrying out the translation.

8.1 Program Development Module

Programming methodology includes task descriptions, Task level UniSet, or motion

descriptions for specific machines, UniSet, using the system GUI. Figure 8.1 shows the interface to the programming module. In the example shown in Figure 8.1, a cell program is developed in basic level UniSet and Task level UniSet for the manufacturing cell configured in Figure 7.4. This program is for producing two different parts which are processed by the two NC machines Mill1 and Mill2. As can be seen in the figure, the programming editor's GUI provides radio buttons for switching the programming language to UniSet, Task level UniSet, or automatically to the specific native language of the machine whose code fragment is being developed.

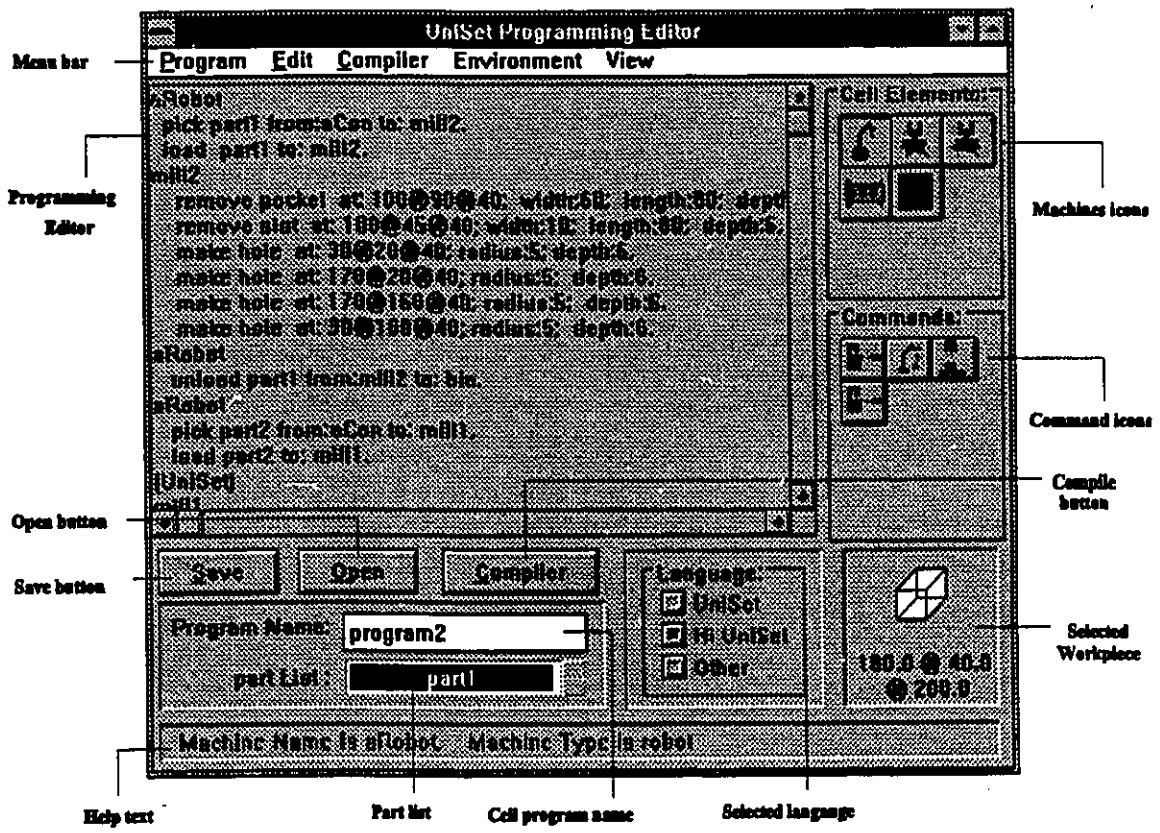


Figure 8.1: User Interface for the Cell Programming Editor (UniSet Editor)

Two levels of program integrity checks are carried out in this mode. The first occurs

during program entry, and is intended to detect syntax errors and to verify the validity of the commands and the parameters for the intended machine. The command interpreter embedded in the program editor gives the cell programmer on-line information and notifies the cell programmer immediately of the error. The second integrity check is carried out at the end of the programming stage and is intended to detect logic or sequence errors.

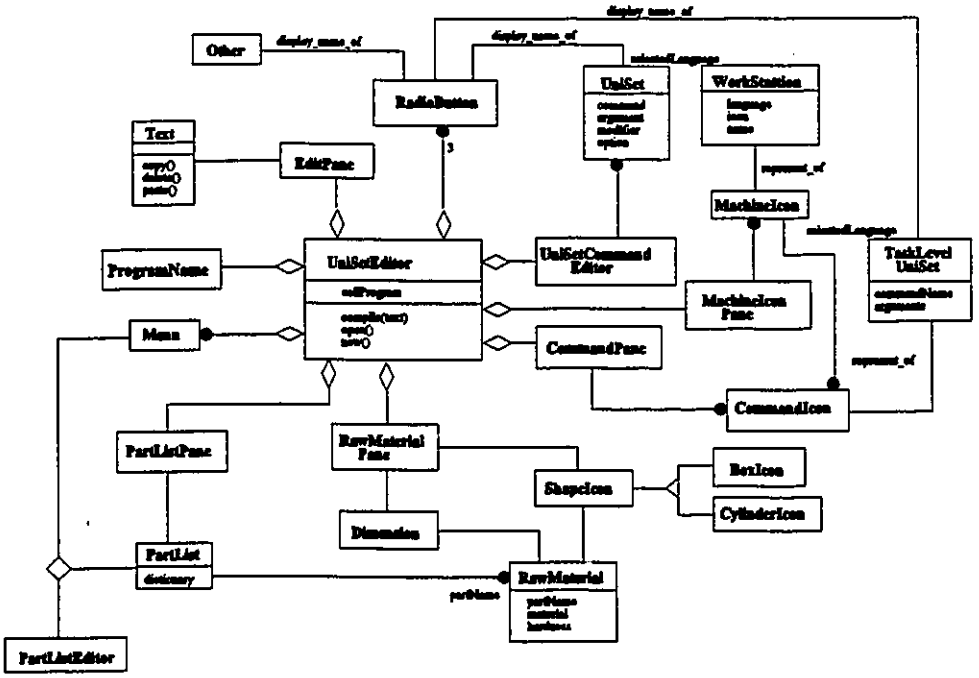


Figure 8.2: Object Model for the UniSet Editor

Since the UniSet Editor manages only text data of the program being developed, its Functional Model is relatively simple and will not be discussed here. The result of a study on the amount and type of the information required for a cell programming task, and on the most convenient mode of relaying this information to the programmer, leads to the contents and arrangement of panes of the UniSet Editor shown in Figure 8.1. The Object Model for the editor

is illustrated in Figure 8.2. With reference to the figure, the instance of the class **UniSetEditor** provides several panes for displaying different format views and facilities for a cell program. Each cell program has its name. This name is used as the identifier of the program in the database. The object **Menu** opens **PartListEditor** which is used for selecting work pieces from the part list dictionary, and the results are displayed in the **PartListPane**. This pane is associated only with the objects of the class **RawMaterial** whose role is to provide the selected material part name, *partName*.

The shape and dimensions of the selected raw material are displayed using **RawMaterialPane**. The information for display is provided through the association of **Dimension** and **ShapeIcon** to **RawMaterial**. The model has three radio buttons associated with the object **UniSet**, the object **TaskLevelUniSet**, and **Other** respectively. The first two language objects provide through their association with the radio buttons the name of the selected language, *selectedLanguage*.

Object on the other hand provides the third button with the name of the language corresponding to the selected machine through its association with the object **WorkStation**. The **UniSetCommandEditor** is associated with many objects of the class **UniSet**, and uses the objects of **UniSet** to prompt the programmer for the different syntax components of the UniSet commands described earlier in Chapter 4.

The object **MachineIcon** is used to graphically represent the objects of the subclasses of **WorkStation**. Each object of the class **MachineIcon** is associated with many objects of the class **CommandIcon** which in turn represents graphically an object of the class **TaskLevelUniSet**. The

objects of **TaskLevelUniSet** are used to select the task commands appropriate for the selected machine, and prompts the user for the required arguments.

This user interface is modular, and is provided with event-ordered behaviour. Figure 8.3 shows the Dynamic Model for the UniSet Editor. As shown in the figure, there are two basic states, *Setting up,* and *Programming*. In the *Setting up* stage, a programmer prepares the editor for a cell program by choosing among other things the name of the program, the raw stock. The state *Programming* is evoked for editing a cell program.

After set up, the event **edit** causes the transition from the state *Setting up* to the state

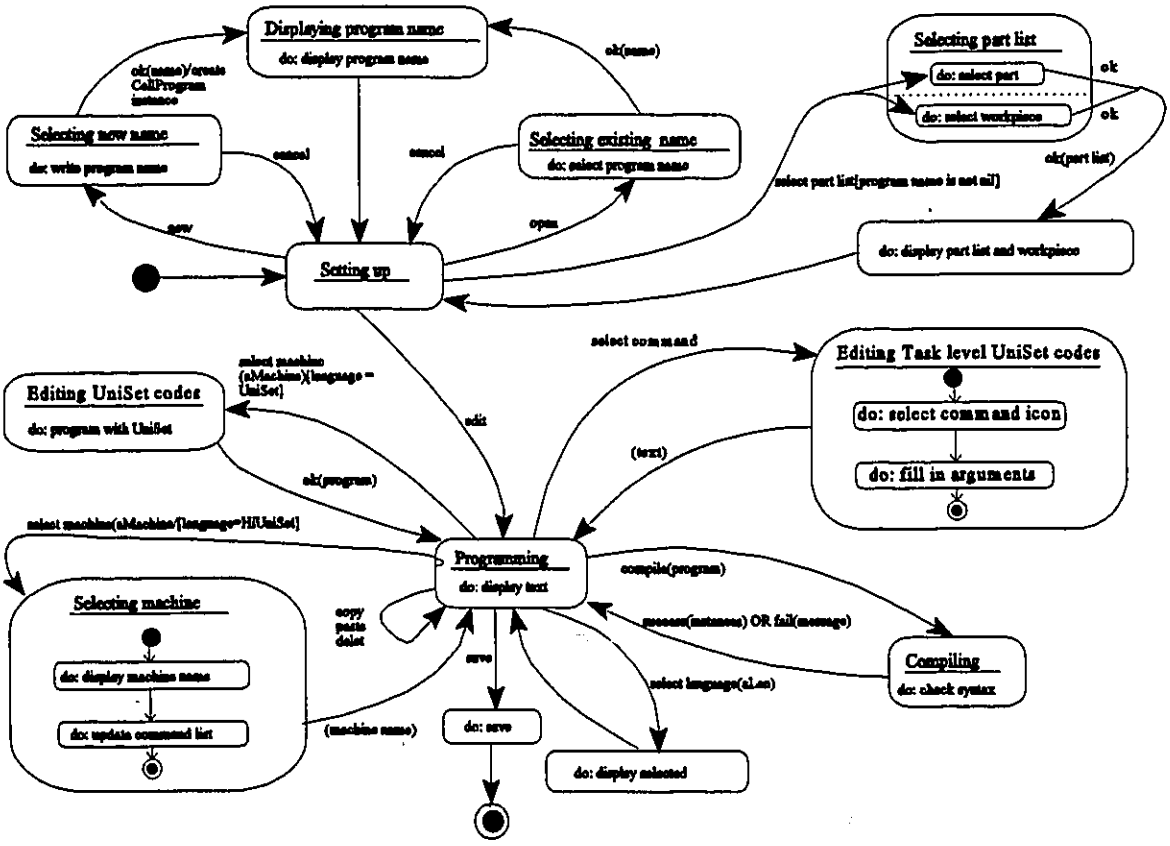


Figure 8.3: Dynamic Model for the UniSet Editor

Programming. In *Setting up*, the events **new** and **open** cause transitions to the states *Selecting new name* and *Selecting existing name*, respectively. The state *Selecting part list* requires two decisions, selecting a part name and selecting the raw material. The two selections are independent. The route from *Selecting part list* back to *Setting up* occur via the transient state *do:display part list and workpiece*.

The event **select machine** can cause transitions to the two possible states, *Editing UniSet codes*, and *Selecting machine*. The state selection depended on whether *language* is UniSet or Task level UniSet. The state *Editing Task level UniSet codes* can be entered only after the state *Selecting machine*. Once a cell program is developed, the event **compile** can be evoked to evaluate the integrity of the program.

8.2 Automatic Generation of Machine/Robot Programs

The UniSet Environment includes an automatic program generation module in addition to the general programming module. This module plays an important role in developing the UniSet Cell Operation Model (UCOM).

As shown in Figure 8.4, the objective of the generation module is to translate the text programs into target machine commands and checks for input errors. The module architecture consists of three passes that step through a series of internal models (i.e., a set of Task level UniSet, a set of UniSet, and a set of target codes). These passes are performed by three main objects, the **UniParser**, the **UniGenerator**, and the **UniTranslator**. Each of these objects have

their own functions and knowledge bases to carry out their task.

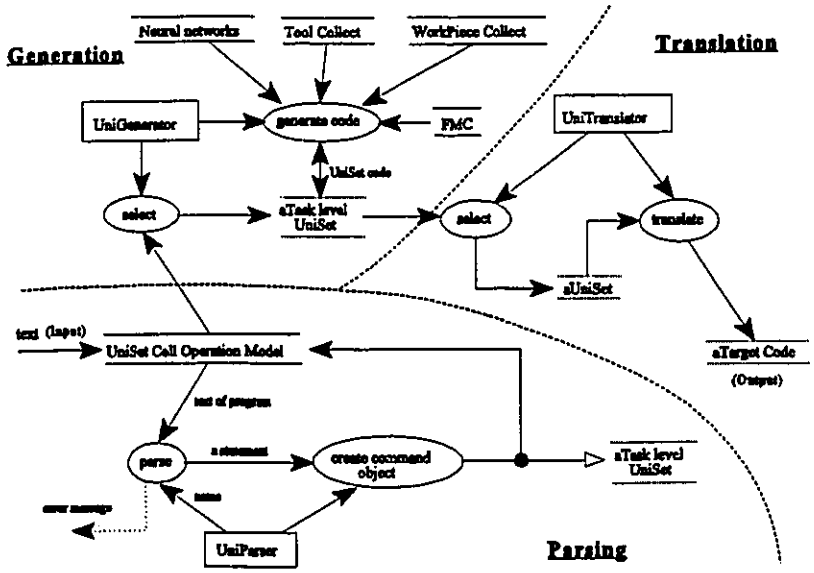
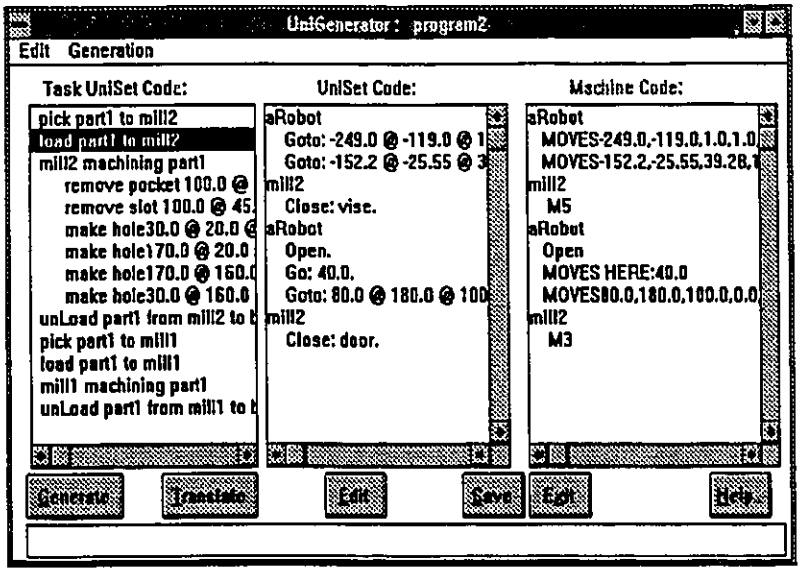


Figure 8.4: Functional Model for the Generation Module

Figure 8.5 shows the Task level UniSet program, which has been developed in Figure 8.1, and the equivalent basic level UniSet and native language programs, for the selected task "load part to mill2". As shown in the example, the selected task uses two machines, **aRobot** which is programmed in VAL II, and **mill2** which uses Word Address.

The segment of UniSet code for each machine is generated automatically by the UniGenerator, and displayed in the UniSet window. In turn each UniSet code fragment for a machine is translated into the corresponding machine's native code fragment by the UniTranslator. If a user does not accept the automatically generated code, ir can be edited directly in this editor.



8.5: Generation and Translation Editor

8.2.1 UniParser

The major task of the **UniParser** is the recognition of sentences that are generated by the programmer, and the semantic interpretation of recognized sentences into a collection of instances of the related classes in **UniSet** and **TaskLevelUniSet**. The **UniParser** must detect input errors but need not correct errors. For instance, an argument type must be satisfied by corresponding command objects. The UniParser must abort when it encounters user errors, and shows error information.

Since the user program may contain any mix of languages, objects had to be created to deal with each of these languages. As shown in Figure 8.6, the object of class **HiParser** deals with commands of Task level Uniset, while the object of class **LowParser** deals with commands of basic level UniSet. These two objects use the object of class **ExpertParser** which contains

knowledge about syntax grammars.

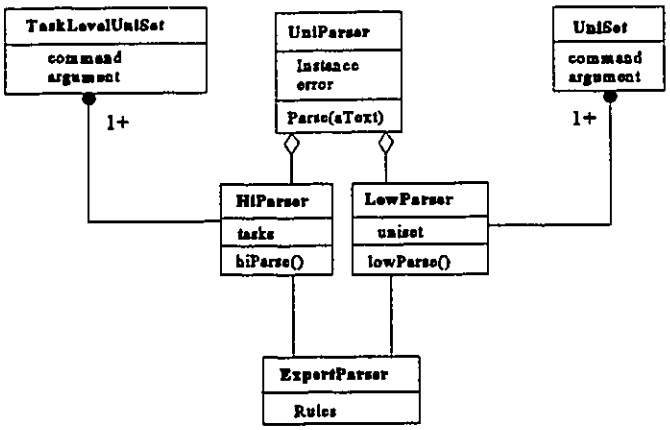


Figure 8.6: Object Model for UniParser

To formulate syntax grammars, two approaches are considered. The first one uses a parsing tree. This approach is appropriate for commands of UniSet, because there are many options and the history of the syntax is important. The parsing trees for each UniSet command are illustrated in Chapter 4.

The second approach is to use BNF. This approach is appropriate for commands of Task level UniSet. This BNF is created during the development of class Task level UniSet, and presented in Chapter 5.

Prolog is chosen as the vehicle language for implementation of these approaches into an object of the **ExpertParse** class. The grammar rules are expressed as predicates based on grammar provided by each language. The argument list of the prolog predicates contain the specification for the primary syntactical categories for the sub components of the currently specified predicate. For example, one form of a sentence in the grammar consists of a <Verb> followed by an

<Object>. This requires that two arguments in the sentence recognition predicate include a specification of the structure of a verb phrase and an object phrase. This appears can be presented in abbreviated form as:

sentence(noun,object)

sentence(noun,object,arg)

Thus, the phrase sentence (noun, verb) specifies that a single prolog argument to this predicate call requires a difference list that matches the structure of a sentence, which is declared in the prolog domain 'sentence', followed by its two variable arguments, *noun* and *verb*. If the single argument consists of a *noun* phrase followed by a *verb* phrase, then the first predicate succeeds. If the first predicate fails, the original argument list is passed to the second predicate.

If the argument consists of a noun phrase followed by a *verb* phrase, followed by an *arg* phrase, then the second predicate succeeds, and so on. If the appropriate difference is not determined, then recognition of the sentence fails, and results in the error statement "syntax error". After the syntax decomposition is successful, the instantiation process starts. The procedure knowledge is also implemented in the class **UniExpertParser** in the form of prolog predicate rules.

8.2.2 UniGenerator

The UniGenerator operates on tasks expressed as instances of the subclasses **TaskLevelUniSet**. Each task object is decomposed into the detailed actions that achieve spacial

relations among objects involved in the task. The process of synthesis is illustrated in Figure 8.7. With reference to the figure:

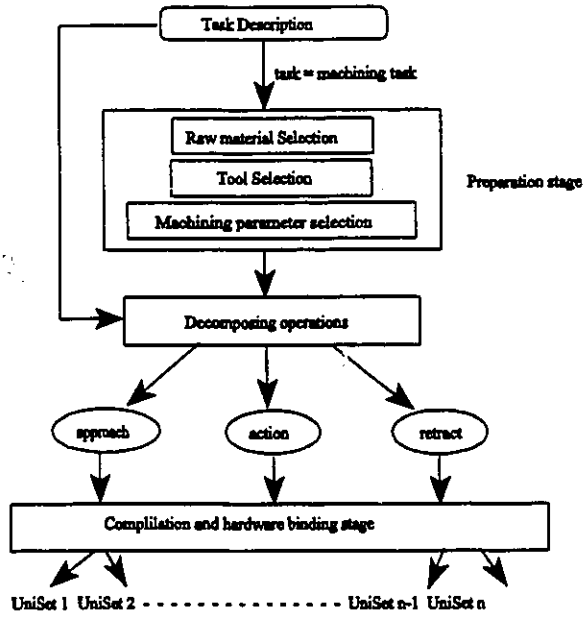


Figure 8.7: Synthesis Process of UniSet Codes Gemeration

8.2.2.1 Preparation Stage

This stage is needed only for the case of tool path generation. Three sub-modules are involved here, namely; the Raw material selection module, the Tool selection module, and the Machining parameter selection module. Discussion of these sub-modules follows.

Raw material selection

The raw material geometry can take a number of shapes. The most common of which are rectangular bar stock, plate, and rod stock. The selection of raw stock requires some study to

ensure its specified engineering properties and economical viability. With these raw materials as a base, the sequence of operation is created to convert this general material shape into a predetermined final shape. When a part program is generated by a CAD system, the raw material should be retrieved from the information in the program. The database information is also used for grasping and transportation of the raw material.

Tool selection

The most appropriate cutting tools for a particular machining task are selected according to a particular manufacturing feature. Tool selection rules are used to synthesize database queries. The query returns a set of feasible tools. The tool database uses an object-oriented scheme and is developed in the set up module.

Tools are classified into two groups: those which are currently in the tool magazine, and those which are available in the tool storage bin. When tools are selected from both groups, priority is given to tools in the magazine.

The rationale is that tool preparation time should be minimized in order to reduce the lead time for the part manufacturing. If an exact tool cannot be found, the next tool on the list is selected. The system also tries to use the same tool for several features. Although one cutter may not be the best for all of the features, reducing the number of tools needed for the job reduces the number of tool changes.

In a machine shop, if only the optimum cutters are used, hundreds of cutters need to be purchased and maintained. This represents a large amount of capital investment as well as

tremendous set up and storage management problems. Also, because the tool magazine can hold only a limited number of tools at any given time, the loading and unloading of additional tools create undesirable idle time.

The actual tool selection involves the determination of the following: tool material (HSS . . .), tool geometry (i.e., Rake . . .), tool type (i.e., milling . . .), and tool dimensions (i.e., length . . .). The tool material selection is based on the raw stock material and its hardness. Suggested tool material data are taken from the "Machining Data HandBook [189]."

The tool type is determined by the machining process. Tool geometry is selected based on the feature geometry, raw material condition, and tool material. As example, the flute length of a tool is selected to be adequately larger than the actual height for the feature. A tool diameter is determined by the feature dimension. For hole features the diameter is easy to determine. For slots, the width of the feature is the upper bound of the tool diameter. Although it is desirable to use the width as the tool diameter, in practice such a tool may not be available. Therefore, the diameter is the largest available tool which is smaller that the width of the slot.

Rules have been implemented to generate tool selection criteria. These rules are divided into three categories: material selection, geometry selection, and cutter type selection. A forward chaining mechanism is used in a tool selection knowledge base. The result of the tool selection rule is a query which is sent to the tool database management system. The selected tool is returned. Currently 35 tool selection rules are implemented in the UniSet Environment of which some rules are listed in the Figure 8.8.

As an example of how to interpret the rules in this list: rule 33 specifies the following:

condition: process is *profiling* and width is d
result (assumed): correct
action: maximum tool diameter is d

If several tools are available, then a number of heuristic approaches can be used to select the best one. Some of these heuristic approaches include choosing the minimum cost tool, choosing the tool with more tool life left, choosing the tool based on the location, and choosing the tool with the maximum number in stock.

```
Expert add:(ToolRules
    number:1
        condition:[ #(process drill) isRight]
        action:[#(cutType drill) doTool]
        description:'process is drilling').
Expert add:(ToolRules
    number:2
        condition:[ #(process turn) isRight]
        action:[#(cutType turn) doTool]
        description:'process is turning').
Expert add:(ToolRules
    number:3
        condition: [#(process profiling) isRight]
        action:[#(cutType endMill) doTool]
        description:'process is profiling').
Expert add:(ToolRules
    number:4
        condition: [#(process facing) isRight]
        action:[#(cutType faceMill) doTool]
        description:'process is facing').
" Rules for tool material selection based on the raw stock material and its hardness."
Expert add:(ToolRules
     number:5
        condition:[ #(material 1) isRight]
        action:[#(toolMaterial 1) doTool]
        description:'material is #1').
.....

    Expert add:(ToolRules
      number:33
        condition: [#(process profilling) isRight & #(width d) isAssume ]
        action:[#(maxToolDiameter d) doAssign]
        description:'profiling process and width d').! !
```

Figure 8.8: Tool Selection Rules

The Functional Model showing the design decisions is presented in Figure 8.9. The *buildfact* process uses information in the **Machining Task** data store to generate a set of facts about the required operation. The object **ToolExpert** uses these facts and rules from the **ToolRules** data store to generate the reasoning for tool selection. The process *find tool* selects the

best tool in the database **ToolCollector** that conforms to the generated reasoning.

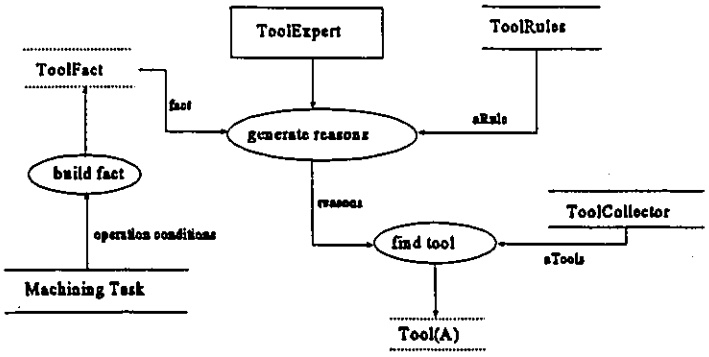


Figure 8.9: Functional Model for Tool Selection

Machining parameter selection

In machining operations, the surface finish, force, and power consumption are directly affected by the machining parameters, feed rate, speed, and depth of cut. The selection of the machining parameters affects the quality, time, and cost to produce a part. These parameters are not arbitrary, nor are they constants for different machining processes.

Earlier studies on economical selection of machining parameters are grouped into three categories: optimal techniques, handbook databases, and machinist experience. The optimal technique approach requires quantifying the result of using a set of parameters in the form of mathematical relationships that can be optimized for a desired result. Due to the complexity of interrelationships between the parameters, such techniques are not widely used in practice. Machinists can through experience select a good set of parameters for a required outcome. However, experience varies widely, and is very difficult to capture in a machine useable format.

Handbooks of test results contain recommended machining parameters for efficient machining. The data set contained in these handbooks is very large. While it may be possible to house it in a database, and use search engines for parameter determination, such an approach would be wasteful.

As has been mentioned earlier, neural networks have been implemented and used in this work for parameter selection. This approach saves computing time and storage space. In addition, it provides easy extendability as new data become available. Figure 8.10 show the Functional Model of machine parameter selection module.

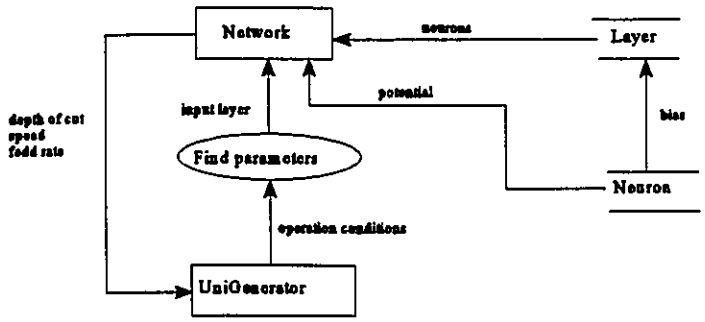


Figure 8.10: Functional Model for Machining Parameters Selection

8.2.2.2 Decomposing Operations

A task involving more than one component of the cell, like a machine and a robot, is mapped into a set of operations, that the individual components have to carry out. These operations would involve hardware independent operations as well as information about the coordination required between the components involved in the task.

At this level, a device, robot, or machine, can perform only one operation at a time. Parallelism is achieved by employing more than one device. The mapping relations of the operation are represented by the transition diagrams and tool movement diagrams in Task level UniSet as has been previously discussed in chapter 4.

As in an example of a robot feeding raw material to a machine, the task is decomposed intc three operations: approach operation, action (feed) operation, and retract operation. In the approach operation, the robot positions the workpiece into the machine chuck. This operation involves several interlock signals to ensure that the machine is idle, ready to receive the part, and have its chuck open. Furthermore, the operation requires assessment of possible collision between the workpiece and the chuck. If collision is detected, collision avoidance is ensured by automatically defining a via position. The feed operation also entails an interlock to ensure that the chuck has closed on the part before the robot releases it. In the retract operation, the robot moves back in preparation for a next task. Again here collision avoidance consideration is needed.

8.2.2.3 Compilation and Hardware Binding Stage

In this stage, the required operations must be transformed from the ideal task domain into the inaccurate real-world domain. Compilation involves decomposing every operation into a set of machine/robot motions. The sequence of the NC machine tool path is directly obtained from the tool path diagram. Operations are related to a geometric model of the world. A simple model where each location of interest is presented with frames and transforms from one frame to another is developed in the set up module and used here. The interest locations are the robot base

coordinates, the machine base coordinates and rest location for a machine with respect to the world coordinates. Via location and a loading reference frame are defined with respect to the machine base coordinates. The orientation of the via location should be the same as one of the loading reference frame. The homogeneous coordinates system is utilized.

The locations of parts are defined with respect to the coordinates of their origin which are supposed to match defined reference frames (such as loading and picking reference frames). The geometrical shapes of these part have to be described, as well as salient positions on them. The system allows for the computation of the grasping position of each object, and its position after it has been moved. All these geometrical information is handled by the UniSet Cell Facility Model and UniSet Cell Operation Model.

As an example, the task "load a part to a machine" is considered in Figure 8.11. The sequence of locations of the end effector of the robot with respect to the robot coordinates (**R**) are used in the generated UniSet codes as arguments. The locations of the end effector of the robot are obtained as follows:

For the task "load a part to a machine," the end effector of the robot while holding a part is assumed to be located near a machine (rest location **r**). The transformation matrix for this rest location (**r**) is expressed as follows:

$$T_G^W = T_R^W \, T_G^R \, T_r^W$$

where, T_R^W is the transformation matrix of the robot base coordinates with respect to the world coordinates, T_G^R is the end effector transformation matrix with respect to the robot coordinates, and T_r^W is the rest location transformation matrix with respect to the world coordinates.

The transformation matrix of the end effector with respect to the robot coordinates (**R**), T_G^R is given by:

$$T_G^R = (T_R^W)^{-1} T_r^W$$

On its path to load a part the robot passes by the *via location* defined in such way so that the robot avoids the collision with obstacles in the machine space. The transformation matrix in this case is given by

$$T_G^W = T_R^W T_G^R = T_M^W T_v^M$$

where, T_M^W is the machine base coordinates transformation matrix with respect to the world coordinates, and T_v^M is the via location transformation matrix with respect to the machine base coordinates.

At the via location, the matrix of the end effector with respect to the robot coordinates is given by:

$$T_G^R = (T_R^W)^{-1} T_m^W T_v^M$$

The final step in loading the part in the vice is to match the part origin frame (**p**) is with the loading reference frame (**L**). At this location the transformation matrix is represented as follows:

$$T_G^W = T_R^W T_G^R = T_M^W T_L^M T_g^L$$

where, T_L^M is the transformation matrix of the loading reference frame with respect to the machine, and T_g^L is the transformation matrix for the grasp location on the part with respect to

the loading reference frame. The transformation matrix for the end effector at the loading position with respect to the robot coordinates is given by:

$$T_G^R = (T_R^W)^{-1} T_m^W \; T_L^M T_g^L$$

The robot deproach transformation matrices are generated in a similar fashion. This collection of transformation matrices of the end effector with respect to the robot coordinates are used for generating the arguments for the task code.

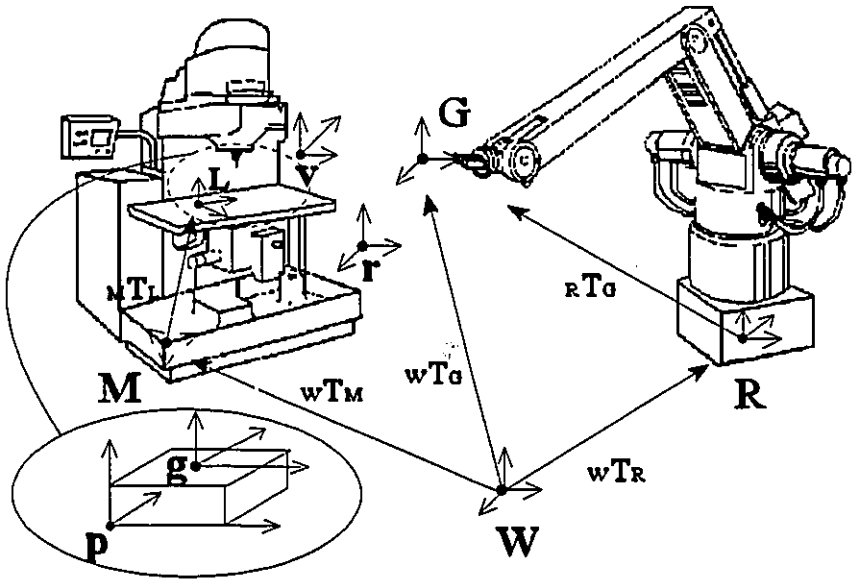


Figure 8.11: Coordinates for the task "load a part to a machine"

8.2.3 UniTranslator

Before a UniSet program can be used to control cell operation, it must be converted into a format that the machines can utilize. This translation is performed by the UniTranslator. In translation, the commands and the arguments are considered separately. The Prolog is again used here for the translation of the commands between UniSet and machine native codes. The equivalences are expressed as arguments of the predicates in Prolog. Figure 8.12 illustrates some predicate rules. With reference to the figure, commands whose format is **UnaryCommand** and **Command** are relatively simple process since a one-to-one equivalence exist between UniSet and machine's native codes. For example, UniSet code **close** indicating a robot closes a gripper, is translated into VAL II code **CLOSE**. However, in the contrast the translation, commands having modifiers, whose format is **ModifiedUnaryCommand**, **UnaryModifiedCommand**, and **BinaryModifiedCommand**, are dependent on the context of the modifier. For instance, the UniSet commands **coolant** depends on the **on** modifier result in the translation Word Address code **M08** while the **off** modifier result in the translation Word Address code **M09**.

The arguments format expressed in UniSet is different from those of natives' machine language due to the use of OOP syntax. Consequently, arguments recasting should be carried out by the UniTranslator. If the argument of the coded command differs in format from thc target language command, a transformation is carried out to harmonize the user supplied arguments with the target language requirement.

```
!ExpertUniTranslator methods !

trans('goto','rapid:','MOVE','val2','UnaryModifiedCommand').
trans('goto','rapid:','G90;G00','word','UnaryModifiedCommand').
trans('goto:','POS','asea','Command').
trans('go:','G91;G01','word','Command').
trans('go:','GODLTA/','apt','Command').
trans('go:','MOVES HERE','val2','Command').
trans('go:','POS OFFSET','asea','Command').
trans('gocircle','G17G02','word','BinaryModifiedCommand').
trans('spindle','ccw:','M04S','word','UnaryModifiedCommand').
trans('spindle','ccw:','SPINDL/,CCW','apt','UnaryModifiedCommand').
trans('coolant','off','COOLNT/OFF','apt','ModifiedUnaryCommand').
trans('coolant','off','M09','word','ModifiedUnaryCommand').
trans('coolant','on','M08','word','ModifiedUnaryCommand').
 -----
trans('close','M22','word','UnaryCommand').
trans('close','Close','val2','UnaryCommand').
trans('close','GRIPPER','asea','UnaryCommand').
trans('close:','close','val2','Command').
trans('tool:','G10H<num>Z<num>','word','Command').
```

Figure 8.12: Prolog based Translation Rules

The unique characteristics of OOP allow this task of translation to be performed by the object of the class of the individual UniSet code rather than by the prolog translation mechanism. This methodology allows each argument associated with a command to translate itself based on knowledge encapsulated in the UniSet command object. As an example, Smalltalk code for the translation of arguments associated with the command **Goto**, is as follows:

```
Command subclass: #Goto
 instanceVariableNames: ''
 classVariableNames: ''
 poolDictionaries: ''  !

        !Goto methods !

apt
 |return x y z|
 (argument isString) ifTrue:[^argument].
 x := argument valueX.
```

```
 y := argument valueY.
 z := argument valueZ.
return :=''.
return := return,x asString,',',y asString,',',z asString.
^return!

asea
^argument aseaFormat!

val2
^(argument valueX argument valueY,argument valueZ!

wordAddress
  |return x y z|
x := argument valueX.
y := argument valueY.
z := argument valueZ.
 return :=''.
 return := return,'X',x asString,'Y',y asString,'Z',z asString.
^return! !
```

For the case of VAL II, APT, and Word Address, the translation of the arguments is relatively straightforward. However, as mentioned earlier, UniSet uses Euler angles referred to the robot coordinate frame to describe orientation, while the ASEA robot uses Quaternions to define orientation with respect to a wrist coordinate frame attached to the tool mounting plate. Consequently, a transformation procedure from Euler angles to Quaternions is performed by a Smalltalk object based on the relations outlined in Appendix B. The example programs generated by these facilities are presented together with a comparison between them and programs developed by a user is illustrated in Appendix C.

CHAPTER 9
Compilation, Simulation and Control Modules

Cell programs generated using the Task Level UniSet in the programming module form the static structure of the UniSet Cell Operation Model (UCOM). This static structure does not contain succession and interlock logic required for cell dynamics, and to ensure proper and safe operation of the FMC. In the simulation module, the environment uses the cell program to generate the dynamic structure, necessary to complete the UCOM, following specific cell process rules. Cell processes are represented by a Task Initiation Diagram using an object-oriented approach. The module constructs the succession of events and the interlock logic automatically based on current states of the machines and the cell. The UCOM for the cell program can then be simulated and executed in the simulation and control modules.

9.1 Task Initiation Diagram Development Module

This module provides the user with a mechanism to edit the Task Initiation Diagram (TID) for the cell program. Figure 9.1 shows the user interface for this module. The TID can be generated automatically from the process rules and the task description, or can be created by the user. In either case the diagram can be modified and edited by the user. The UniSet TID Builder

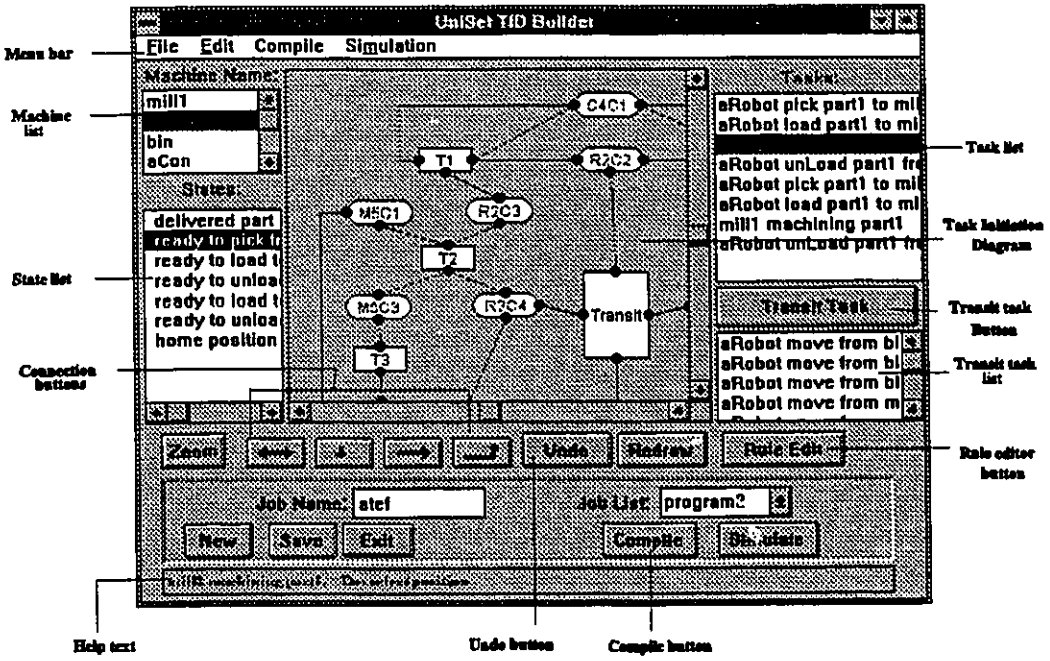


Figure 9.1: User Interface for the Task Initiation Diagram

GUI provides the user with all the information required to accomplish the task of completing the

UCOM. As an example, consider the program shown previously in Figure 8.1, all the tasks involved are shown in the **Task list pane** of Figure 9.1. The TID for the program is shown in Figures 9.1 and 9.2. With reference to the diagrams, boxes represent tasks, ellipses represent states. Symbols in boxes and ellipses are automatically determined by the system. A solid bullet at the end of a line indicates the source, while a gray bullet indicates a destination.

The Object Model for the **UniSet TID Builder** is shown in Figure 9.3. With reference to the figure and to Figure 9.1, the instance of the class **UniCompiler** provides the GUI with several panes and buttons. The panes most worthy to describe are the Machine list, the State list, the Task list, and

the TID panes. Description of these follow:

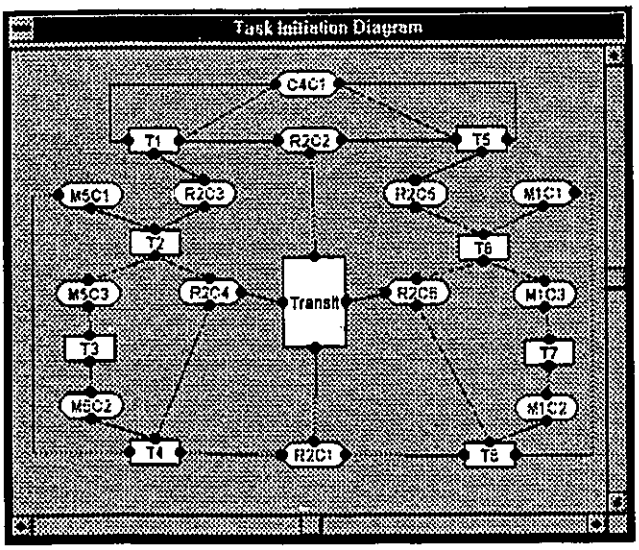


Figure 9.2: Task Initiation Diagram for the job *atef*

The Task list pane contains the list of tasks in the user cell programs. This pane is created by the class **TaskListPane** class. **Task** class is responsible for listing the tasks in the cell program. The Machine list pane is created by the class **MachineListPane** and the cell components are provided by class **MachineList**. The TID pane is created by **TaskInitiationPane** and **TaskBox**, **Connection**, **StateBox**, and **TransitBox** classes are responsible for generating the symbols of the diagram. The TID itself is created by the compilation module that will be described later.

A highlighted task in the Task list pane causes through the association of **Task** with **MachineList** the machine names involved in the task to be highlighted. The highlighted machines evaluate their states required for proper execution of the task according to the described operational rules. Each machine is an instance of the subclasses of the abstract class **WorkStation**. The states are listed in the State list pane created by the class **StateListPane**. Five buttons are created by class

Button and are used for graphical editing purposes.

Because the **UniSet TID Builder** is used to manage graphical primitives and information, its Dynamic Model should be evaluated carefully. The dynamic interaction model is represented in Figure 9.4. In this figure, all states involved in constructing and editing the TID are associated with activities.

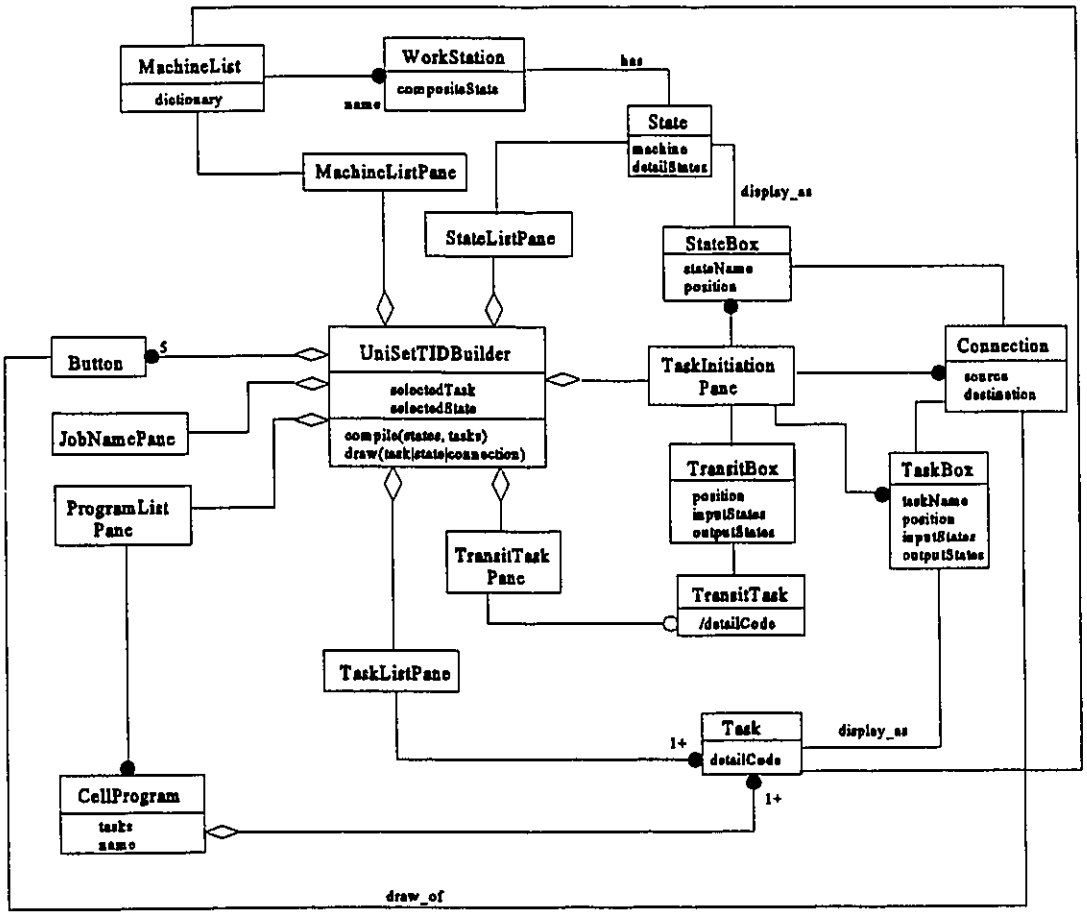


Figure 9.3: Object Model for the UniSet TID Builder

The initial state of the builder is *Editing*. The event **Select task** causes the transition to the state *Selecting task* followed by the state *Selecting machine* automatically. This is again followed automatically by the state *Selecting state*. Another entry point to the states *Selecting machine* followed by *Selecting state* is through the event **Select machine**. For the cases where the user wishes to modify the TID, the user enters the *Selecting symbol* state through the event **Select**. In this state the user can modify the state of a machine, the connections on the TID, and the locations of the symbols on the TID.

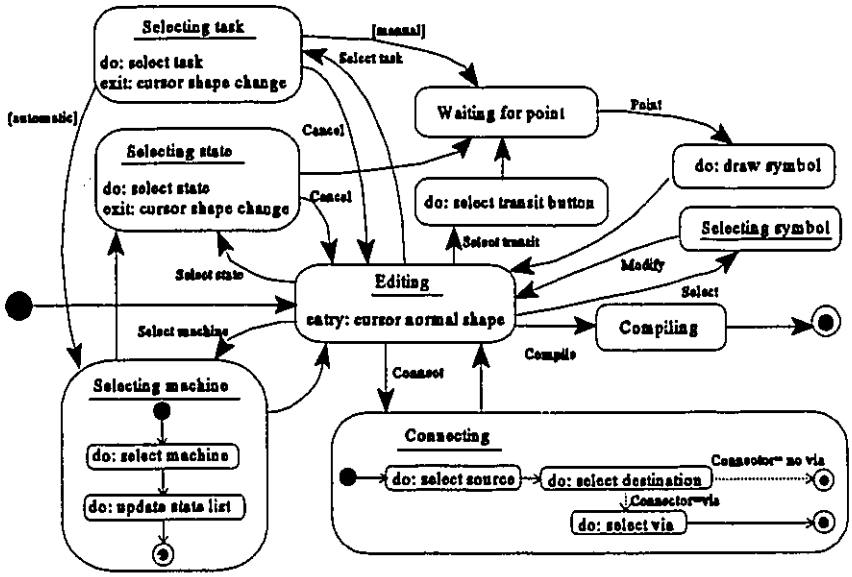


Figure 9.4: Dynamic Model for the UniSet TID Builder

The user can override the automatic generation of the TID, and construct the diagram manually. The sequence for this process starts by selecting a task from the Task list pane. After selection the state *Waiting for point* is entered and the cursor is highlighted in the TID pane. If user input is an appropriate point, the activity of the draw symbol is performed. The state returns to

Editing. The *Selecting state* activity is accomplished in the same manner. When all states and tasks are located in the TID pane, the *Connecting* state is entered through the event **Connect**. The user selects the source and destination symbols, and if required via points to avoid crossing of lines. The state *Compiling* is entered using **Compile** to generate rules according to the Task Initiation Diagram and terminates this builder.

9.2 Compilation Module

This **Rule Compiler** module is responsible for creating process rules from a Task Initiation Diagram developed in the previous module. The rules take the form "If - Then". The **Rule compiler** Functional Model is shown in Figure 9.5.

The three main processes in compilation are: the *extract connection* process, the *generation rule* process, and the *generation transit tasks* process if required. The **Task-state connection model** is an intermediate data store.

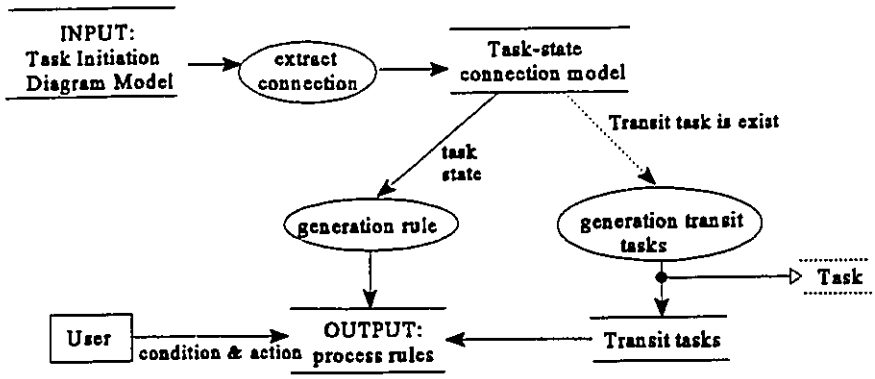


Figure 9.5: Functional Model for the Rule Compiler

9.2.1 Task Initiation Diagram Model

Figure 9.6 illustrates the TID Object Model. The geometry and identifying text primitives of the TID are created by graphical objects in the model. These objects also capture the semantic meanings of the components of the TID.

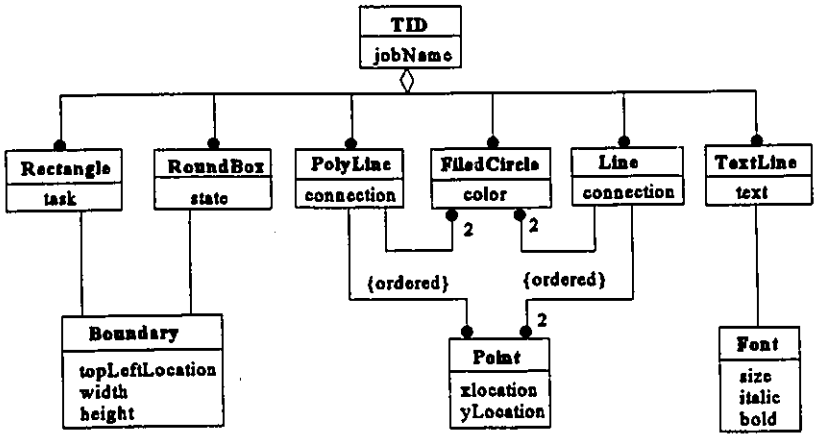


Figure 9.6: Object Model of the Task Initiation Diagram Model

9.2.2 Extract Connection Process

This process consolidates raw geometries and detects attachments of geometric primitives. The output of this process is the Task-state connection model. Figure 9.7 represents the functional model for this process. Classes **Rectangles**, **RoundBox**, **PolyLine**, **Line**, and **TextLine** from the TID model are the input data stores in the functional model. Every task, transit task, state, and connection in the Task-state connection model is the output of one or more functional model processes.

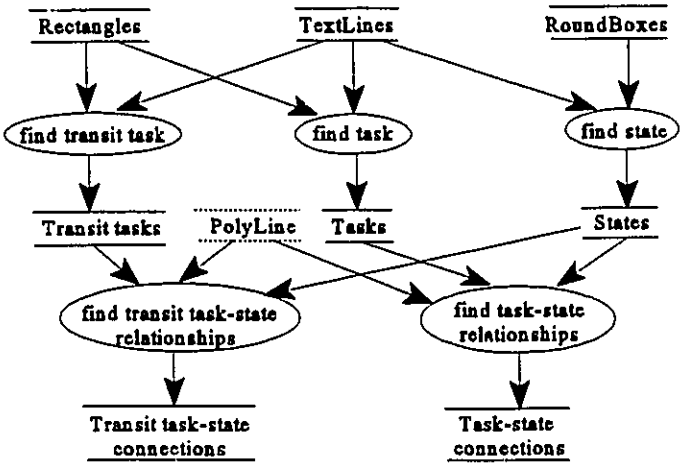


Figure 9.7: Functional Model for the Extract Connection Process

9.2.3 Task-State Connection Model

This model regards the TID as a topology, a collection of connections made between a state

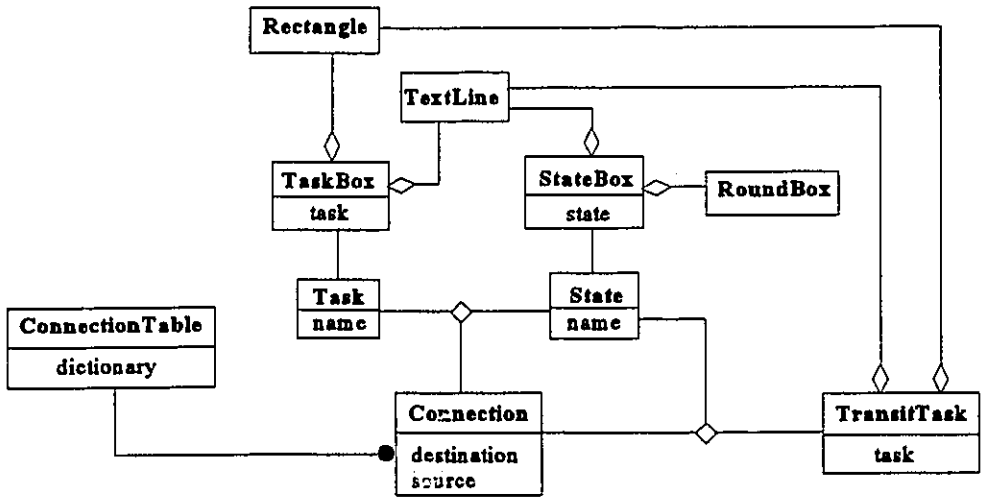


Figure 9.8: Object Model for the Task-State Connection Model

and a task. This connection, task, and state have attached geometry and text. The Object Model for the Task-state connection is shown in Figure 9.8. The **Task**, **State**, and **Connection** classes form the core of this model. A connection corresponds to exactly one state and one task. A transit task may be connected to many states.

9.2.4 Transit Task Generation Process

For FMC's where a single robot serves a single machine, Transit tasks are not required. However, when a single robot serves several machines, or several robots serve several machines, the user programs each cycle separately in Task level UniSet. In this case, in order to decrease wait time for machines the UniSet Environment determines the most suitable cycle transfer events.

The Transit Task is the mechanism used to arbitrate the succession of cycles. These tasks are generated automatically and the code sent to the robot controllers. The Cell Controller determines in real time which of this transit task is to be executed depending on the states of the cell. As an example, Figure 9.9 magnifies the Transit Task in the TID of Figure 9.2. The seven dotted lines represent the possible paths between the two cycles.

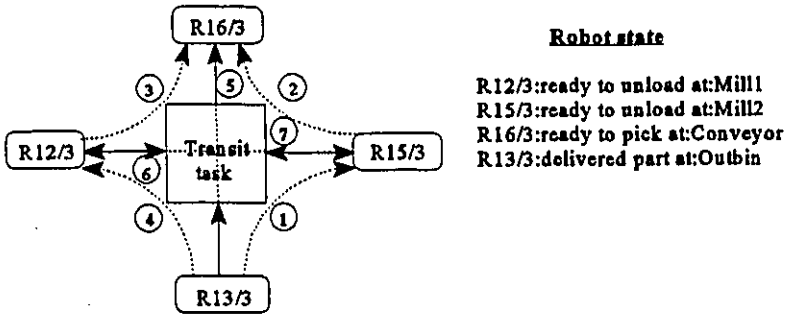


Figure 9.9: Example for the part of TID

These paths, in turn, are transformed into tasks in Task level UniSet. Table 9.1 illustrates the equivalent Task level UniSet commands to the state transitions.

Table 9.1 Generation of the Transit tasks

Transit number	State transition	Task Level UniSet Command
①	R13/3 → R15/3	move to:Mill2
②	R15/3 → R16/3	move to:Conveyor
③	R1/2/3 → R16/3	move to:Conveyor
④	R13/3 → R12/3	move to:Mill1
⑤	R13/3 → R16/3	move to:Conveyor
⑥	R12/3 → R15/3	move to:Mill2
⑦	R15/3 → R12/3	move to:Mill1

9.2.5 Rule Generation Process

This process creates the process rules which are the symbolic format of the production rule, "If ... Then". The tasks, states, and connections are the input for this process. The output is the production knowledge database which contains the production rule set.

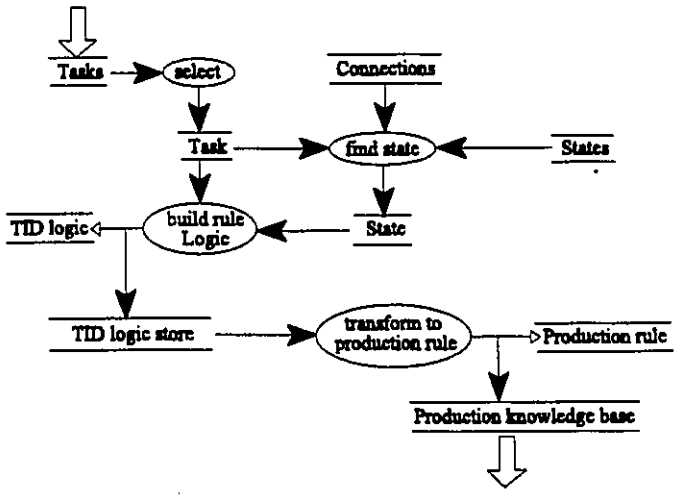


Figure 9.10: Functional Model for the Rule Generation Process

The Functional Model for this process is shown in Figure 9.10. The process *build rule logic*

takes each task and the associated states and generates the TID logic data which is stored in the **TID logic store**. These elements of the **TID logic store** are transformed by the process *transform to production rule* into production rules.

The User interface for this process is shown in Figure 9.11. This Figure shows a part of production rules for the Task Initiation Diagram developed in Figure 9.2. This editor consists of three main panes: the **Condition** pane, the **Action** pane, and the **Description** pane. The condition consists of three parts: argument1, which is the name of the component, argument2, which represents the name of the state, and operator, which performs the function of a relationship. The format of an action is similar to a command of the Task level UniSet. The description helps in understanding a rule. The rule number is used as identifier to retrieve a rule from the rule database. The buttons are used to edit a rule, for example, delete a rule from a rule database.

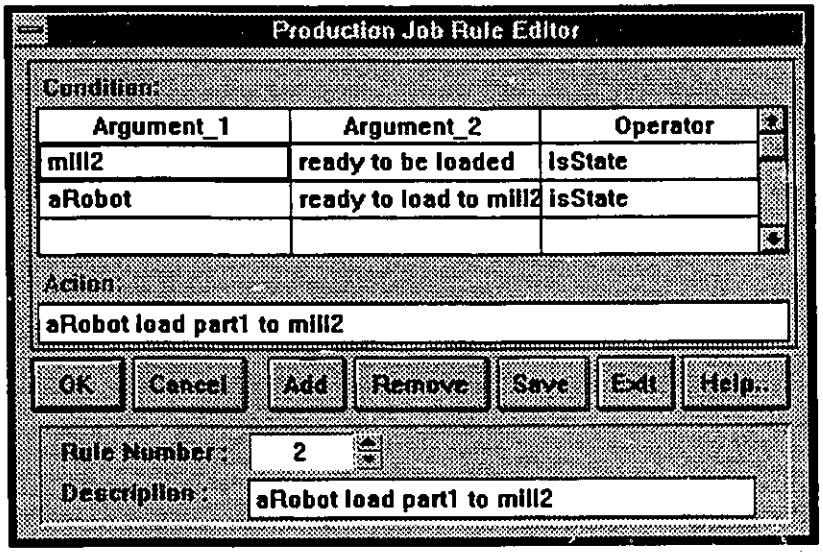


Figure 9.11: User Interface for the Rule Generation Editor

9.3 Simulation and Control Modules

Before implementing the cell program, this module can be used to simulate and evaluate it. However, the major function of this module is to schedule tasks, machines, and other resources in the cell. In addition, this module may be equipped to handle exception conditions and deal with them using recovery procedures.

9.3.1 Cell Controller Operation In UniSet

Cell operation involves tasks that are performed on single machines independent of others, and tasks that require the cooperation of two or more machines. Where that task calls for the coordination of two or more machines, the cell controller has to be involved to ensure proper execution of the task. For tasks involving a single machine, the controller's main task is to schedule the start of the task, and waits for its completion in order to command the nest task.

In order to accomplish both functions, the UniSet controller is designed as a hybrid structure of both hierarchal controller and decentralised controllers as shown in Figure 9.12. The cell controller consists of three different layers. The Task Scheduler, the Decentralized Control layer, and the Virtual Device layer. In the figure, dark lines indicate physical connections and light lines indicate logical connections. Information and message passing are indicated by arrows.

The Task Scheduler is a core component which receives the states of all the machines in the cell from the Decentralized Control layer, and decides the next task. It then dispatches the nest task to be executed to the Decentralized Control layer. It uses the process knowledge bases that contain the routine cell task rules which are generated from the TID. As has been discussed earlier, these rules are in the form of IF-THEN statements. The arguments of the IF part indicate the cell states

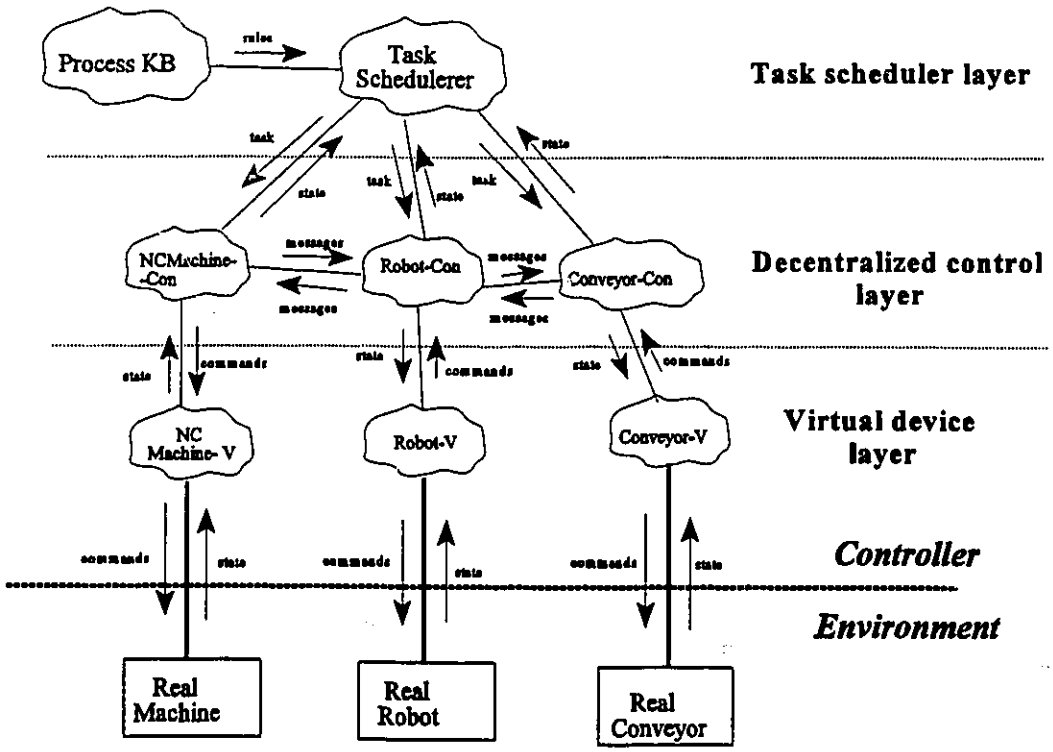


Figure 9.12: Control Structure of the Cell Controller in UniSet

that must prevail in order for a task to be started. The arguments for the THEN part contains the required task. It is an inference engine that will select the required rules to handle the cell operation based on a forward chaining reasoning procedure. The cell state describes, at any instance in time, is given by the individual states of the cell constituents obtained from the Virtual Devices.

The Decentralized Control layer consists of Virtual Controllers for the physical machines. Their main role is to perform the harmonization and the cooperation between the cell components in order to carry out the task called for by the Task Scheduler layer. They provide a device independent interface to the actual cell components by translating the generic commands and error

messages of the corresponding machine. The Virtual Controllers in the layer communicate and pass messages with each others. Each is correlated to a dedicated Virtual Device and has no connections with other Virtual Devices. A Virtual Controller sends commands to the corresponding physical machine, and receives the state of that machine, through that machine's Virtual Device in the Virtual Device layer. If a give task includes an interaction between machines, the Virtual Controllers concerned exchange the necessary information to coordinate the interaction between the machines. These controllers utilize the OID to carry out their function.

The lowermost layer of the controller consists of the Virtual Devices which monitor and continuously mirror, in real time, the state of the physical machine they represent. Each machine's state is analysed by its Virtual Device and reported to the corresponding Virtual Controller as required. The Virtual Devices also act as conduits for commands from the Virtual Controllers to the physical machines.

If the cell does not experience machine or tool failures, the cell control will not need to intervene to stop the production cycle, and to carry out error recovery procedures. In practice such situations are very likely to occur during cell operation. While error detection and handling can be tailored for different cells by implementing fault sensors and declaring them during the cell setup stage, errors can also be detected when a device or machine fails to report operation completion after a specified time. This time is a function of the expected duration for performing the task. The error is detected by the Virtual Controllers. The current mechanism implemented for dealing with errors, call on the Virtual Controllers to immediately abort operation of the cell, and to report to the topmost layer in the hierarchy, the Cell Scheduler, for action.

The error knowledge that is contained in the Task Scheduler layer, includes different rules

for dealing with different problems that occur during cell operation. These rules are contained in workstation, cutting tool, and cell operation knowledge databases. The rules consist of antecedent and consequence parts, which have the form of IF-THEN clauses. The problem solving system needs to diagnose the cause of the breakdown when it occurs. All the facts are related to the conclusions of the rules (the THEN part of the rules). This system tries to find out the causes of the breakdown through these conclusions and to correct the causes. Backward chaining is used to search rules to solve the problem. The error state stores the status of workstations, robot, and part. After receiving the status of the cell, the status of the error state will be updated.

9.3.2 User interfaces for the Simulation andControl Modules

As an example, the GUI for the Control Module for the cell configured in Figure 7.4, is shown in Figure 9.14. This interface consists of several windows. Each cell component has a corresponding window. An additional window is used for the Cell Scheduler. For example, the window named **aRobot** contains information about a real robot, so that the window is associated with a **virtual robot**. This window consists of two panes: the pane for the state, and the current operation. This individual task may be changed after the operation is performed. The other three windows have the same structure.

The window for the **UniSet TaskScheduler** consists of three main panes. These contain dynamically updated information about the cell components' states, the reasoned task, and the explanation of the reason.

The states of the cell components can be initialized by the main window before beginning

operation. If the job is assigned by a user, the main window retrieves all information including process rules and sets up the cell controller. By pushing button **start**, this user interface executes its own process.

atef

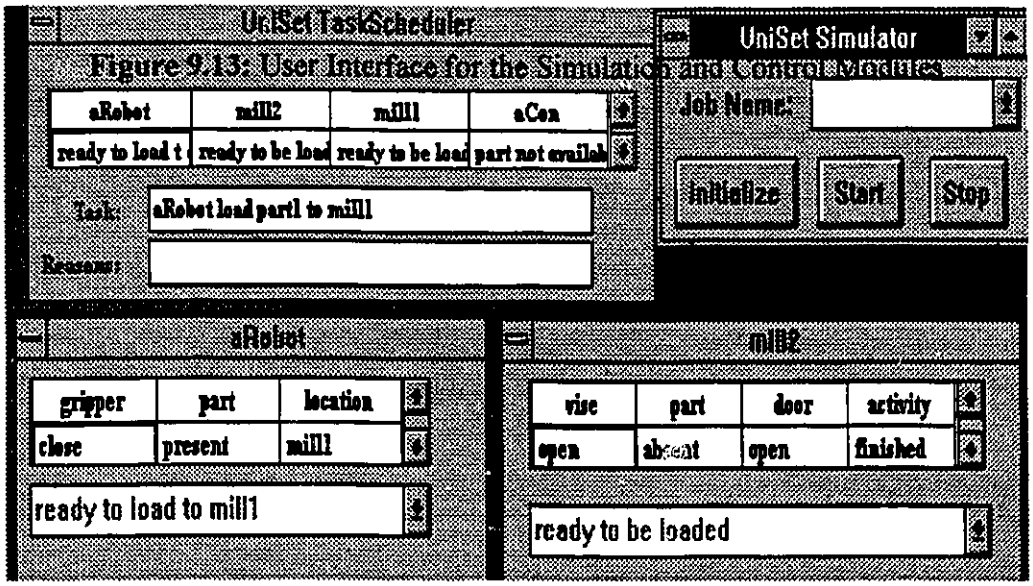


Figure 9.13: User Interface for the Simulation and Control Modules

CHAPTER 10
CONCLUSIONS

The result of the research work reported in this thesis is a fully functional software environment for dealing with Flexible Manufacturing Cells. The UniSet environment provides the facilities for setting-up the cell, for programming it, for simulating its operation, and for controlling it. The environment is easy to use, and measures have been taken in its design so as to guard against users' errors. The environment has been developed using knowledge-based approaches and the Object Oriented Paradigm.

In the setup module, three main object data models are built to captures information involved in manufacturing applications using FMCs. These models are created from the generic models which have been developed for various application domains using Object Modeling Technique (OMT) and reported in Chapter 6. These models are the UniSet Cell Facility Model (UCFM), the UniSet Tool and the RawStock Model (UTRM), and the UniSet Machining Parameters Model (UMPM).

UCFM deals with physical locations of the cell constituents, their characteristics and programming languages. UTRM contains a collection of tools and raw stocks available. UMPM contains knowledge bases regarding the selection of the optimal machining parameters, which are represented as neural networks. Three neural networks have been trained and implemented into the

current environment, and can be extended and made installation specific. Since these models are integrated and shared with other modules, the information contained in the models is used later during programming, simulation, and control.

In order to deal with building these object data models, four main editors have been designed. Details of the design are presented in Chapter 7. These editors are the Cell Configuration Editor, the Tool Database Editor, the RawStock Database Editor, and the Neural Network Builder. Each editor is associated with a particular object data model. These, as well as other editors used in the environment are designed based on the Model-View-Controller (MVC) concept. To facilitate user interaction all the editors use a GUI.

Since the information in these object data models are vital for proper operation and safety, and because it requires a high level of expertise, the editors for the setup module are not available to a cell operator. A cell designer however, have access to this module.

The programming module is designed to facilitate a cell programmers job. UniSet has been developed as a manufacturing instruction set. It has an instruction set that caters for all the cell constituents. UniSet has been developed based on comparison of existing automation languages presented in Chapter 4. Its commands are classified into six formats to adhere to a consistent quasi-English syntax.

In addition, the environment provides Task level UniSet. The development of this high level goal oriented language has been presented in Chapter 5. Using Task level UniSet cell programmers would not have to deal with the minute details of programming the different cell constituents. Rather, the programmer concentrates on describing the tasks that need to be performed by the cell constituents. Task level UniSet instructions are classified into two groups: the cell configuration

dependant tasks, and the cell configuration independent tasks. The cell configuration dependant tasks are those that require some coordinations among cell components to carry out the task. The sequential logic interlock mechanism necessary for proper and safe execution of the tasks are represented graphically by the Operation Initiation Diagram (OID). The cell configuration independent tasks are those that require only a specific cell component to perform the task. Task level UniSet is implemented as objects. These objects of Task level UniSet encapsulate data and knowledge that are required to carry out these tasks successfully.

The Unified Instruction Set, and the Task level UniSet are non exclusive. A programmer can use UniSet instructions, native machine instructions, or the Task level UniSet in any mix. To facilitate programming, the programming module is context sensitive. The user chooses the commands from lists that are tailored for the machine being programmed. The system queries the programmer for the modifiers and arguments as required. Details of the design of the UniSet Editor for program development are presented in Chapter 8.

The programming module also includes the automatic generation and translation facilities. Programs developed in Task level UniSet are translated into basic level UniSet codes by the automatic generation facility. Basic level UniSet is then translated into the different machines native code before being dispatched to the machine controllers. These facilities perform their duties using knowledge bases, such as tool selection rule base, and prolog-based translation rules.

A new methodology of representing the interaction between machines and the interlock signals required in FMCs has been developed. The Task Initiation Diagram (TID) and the Operation Initiation Diagram (OID) allow the user to visualize graphically the task programmed, and the prevailing conditions required so they can be initiated. In the compilation module presented in

Chapter 9 the TID is generated automatically from the user program. The generation process uses a set of rules provided during setup, to ensure safe and proper operation. This TID is a core element of the UniSet Cell Operation Model (UCOM). UCOM contains static and control structures of the cell operation. The static structure is in essence a compilation of the tasks required to carry out the production job, and is derived from the user program. The control structure logic required by the controller to coordinate the cell is extracted from the TID and the OID.

A new cell controller structure has been implemented in the UniSet Environment. The Controller has a hybrid architecture, partly hierarchical and partly decentralized. It consists of a high level controller that oversees the operation of a set of logically decentralized controllers. This approach provides benefits embodied in both architectures. It consists of three layers. The lowest layer contains machine representations, and called here Virtual Device Layer. These virtual devices hold an exact, up to date logical information of the machine they represent. The middle layer is the decentralized control layer. Here each machine has a virtual controller. Tasks and conditions for initiating them are relayed from the uppermost layer, the Task Scheduler, to the decentralized controllers in this layer. The decentralized controllers interact together to harmonize the cell operation.

In order to verify and evaluate the operation of a cell based on a cell program, the UniSet Simulator has been developed. It displays dynamic information from the different controller layers, and allows user interaction. The Simulator GUI is also used by the Cell Controller to display salient information during cell operation.

All object data models are developed explicitly in different modules. However, since the UniSet Environment integrates all these modules and models together, the models can be retrieved

and modified in any associated module.

The UniSet Environment has not been tested to date on a physical FMC. Numerous logical test runs have been conducted. The environment has reached a level of maturity believed to be suitable for a physical test.

In the current work, UniSet has the translation facility dealing with only four existing automation languages: Word Address, APT, VAL II, and ASEA. But there exist many automation languages used in industry. This translation facility should be extended to implement other relevant automation languages.

The current version of Task level UniSet covers the basic operations performed by cell constituents. The roster of tasks should be expanded to make the environment easier to use.

REFERENCES

1. Gruver, W.A., Soroka, B.I., "Industrial robot programming languages: a comparative evaluation", IEEE Transactions on Systems, Man, and Cybernetics, 1984, Vol.SMC-14, No.4, pp.560-565.

2. Lozano-Perez, T., "Robot programming", Proceedings of the IEEE, 1983, Vol.71, No.7, pp.821-841.

3. Grossman, D.D., Short, M.M., "AML- much more than robot language", Technical paper presented at Robots 9 Conference, June 2-6, 1985, SME Technical paper No. MS85-612.

4. Grossman, D.D., "AML as a plant floor language", Robotics & Computer Integrated Manufacturing, 1985, Vol.2, No.3/4, pp.215-217.

5. Sharon, A., Carey, E., "AML+PC=CIM?" Computers in Mechanical Engineering, July 1985, pp.30-34.

6. Unimation, Inc., Unimate Industrial Robot, Programming Manual- User's guide to VAL II, Document 398T1, Ver.1.4b, 1985.

7. Chang, C.H., Melkanoff, M.A., NC machine programming and software design, Prentice Hall, 1989.

8. Wright, P.K., Bourne, D., Colyer, J.P., Schatz, G.S., "A flexible manufacturing cell for swaging", Mechanical Engineering, October 1982, pp.76-83.

9. Bourne, D., "Designing programming languages for manufacturing cells", Robotics Institute, Carnegie-Mellon University, Pittsburgh, Pa, 1982.

10. Bourne, D., "A numberless, tensed language for action oriented tasks", Robotics Institute, Carnegie-Mellon University, Pittsburgh, Pa, 1982.

11. Bourne, D., "A multi-lingual database bridges communication gap in manufacturing", Robotics & Factories of the future(Dwiveldi,S.N.,Editor), Springe-Verlay, 1984, pp.545-552.

12. Bourne, D.A., "CML: a meta-interpreter for manufacturing", AI magazine, Fall 1986, Vol.7, No.4, pp.86-96.

13. Bourne, D., Fox, M.S., "Autonomous manufacturing: automating the job-shop", Computer, September, 1984, pp.76-86.

14. Wright, P.K., Bourne,D.A., Manufacturing Intelligence. Addison-Wesley Publishing Company, Inc. 1988.

15. Pinson, L.J., Wiener, R.S., "An Introduction to object oriented programming and Smalltalk", Addison-Wesley Pub.. Inc., 1988.

16. LaLonde, W., Pugh, J.R., Inside Smalltalk, Vol.1, Prentice Hall, 1990.

17. Booch, G., Object oriented design with applications, Benjamin/Cummings Pub. Co., Inc., Redwood city, CA, 1991.

18. John, H.S., "A survey of object-oriented programming languages", Journal of object oriented programming, March/April 1989, pp.5-11.

19. Smalltalk v object oriented programming system, Tutorial and programming Handbook, window series, Digitalk Inc., 1992.

20. Thomas, D., Best, R., "Smalltalk+C The power of two", Dr.Dobb's Journal, August, 1989, Vol.14, pp. 16-18.

21. Bonney, M.C., Young, K.Y., "Off-line programming using the GRASP robot simulation system", Proceedings of 2nd International Conference on CAPE, 1987, pp.67-70.

22. Wright, E.J., "Some experiences of off-line programming on ASEA robots", Proceedings of 4th International Conference on Computer-aided production Engineering, 1988, pp.135-140.

23. Miller, D.J., Lennox, R.C., "RIPE: A robot independent programming environment", Proceedings of Intelligent robots and computer vision X: Algorithms and Techniques, 1991, Nov. 11-13, pp.518-529.

24. Petriu, E.M., Goguen, J.P.T., Karoui, B., Basran, J.S., "Robotic-assembly architecture for a task-level programming language", Proceedings of Symposium On Manufacturing Application Languages, June 20-21, 1988, pp.41-48.

25. Gini, M., "From goals to manipulator programs", Computers in industry, 1986, vol.7, pp.263-273.

26. Mettala, E.G., Strong, D.B., Egbelu, P.J., "Development of an artificial intelligence interface for robotic assembly in a flexible manufacturing system", International Journal of Computer Integrated Manufacturing, 1989, Vol.2, No.1, pp.48-58.

27. Tseng, C.S., Crane, C., Duffy, J., "Generating collision free paths for robot manipulators", Computers In Mechanical Engineering, September/October, 1988, pp.58-64.

28. Pau, L.F., Paus, D., Stokka, T., "Knowledge-based order specific NC drilling system", Conference Proceedings of Simulation and Artificial Intelligence in Manufacturing, 1987, pp.145-152.

29. Kramer, W., "Object-oriented NC programming with CAD/CAM", Dearborn, MIch. Society of Manufacturing Engineers, 1989.

30. El-Midang, T.T., El-baz, M.A., "Auto CNC part programming through machinability data base system", PD-Vol 43, Computer Applications and Design Abstraction, 1982, pp.1-7.

31. Zhang, D.T., Chen, J.W., "An NC lathe simulation for part programming and machine operation training", Computers in Industry, 1993, vol.21, pp.139-148.

32. Conde, D., Mediavilla, E., Granda, M., "Job programming language for multi processor control systems", International Journal of Robotics and Automation, 1990, Vol.7, No.3, pp.126-132.

33. Volz, R.A., Mudge, T.M., Gal,D.A., "Using Ada as a Programming language for robot-based manufacturing cells", IEEE Transactions on Systems, Man, and Cybernetics, 1984, Vol.SMC-14, No.6, pp.863-878.

34. Levas, A., Jayaraman, R., "WADE: an object-oriented environment for modeling and simulation of workcell applications", IEEE Transaction on robotics and automation, 1989, Vol.5, No.3, pp.324-335.

35. Kramer, T. R., "Process planning for a milling machine from a feature-based design", Proceedings of Manufacturing International '88, Atlanta, Georgia, April 17-20, 1988, Vol.69, Pt.4, pp.95-101.

36. Matsuda, M., Inoue, K., "Software Architecture of autonomous manufacturing cells", PED-Vol.53, Design, Analysis, and Control of Manufacturing cells, ASME 1991, pp.173-183.

37. Ranky, P.G., "Intelligent planning and dynamic scheduling of flexible manufacturing cells and systems" JAPAN/USA symposium on Flexible Automation, 1992, Vol.1, pp.415-422.

38. Caselli, S., Papaconstantinou, C., Doty, K.L., "A structure-Function-Control paradigm for knowledge based modeling and design of manufacturing cell", Journal of Intelligent Manufacturing, 1992, Vol.3, pp.11-30.

39. Rembold, U., Levi, P., "The factory of the 90s", Computers in Mechanical Engineering, May/June, 1988, pp.30-47.

40. Kishinami, T., Kanai, S., Saito, K., "An integrated approach to CAD/CAPP/CAM based on cell-constructed-geometric model(CCM)", Robotics & Computer-integrated Manufacturing, 1987, Vol.3, No.2, pp.215-220.

41. Milacic, V,R., " Factory environment and CIM", Robotics & Computer-Integrated Manufacturing, 1990, Vol.7, No.3/4, pp.213-228.

42. Zeng, l., Wang, H.P., "An expert system model for manufacturing planning", Proceedings of the Third International Conference On Industrial & Engineering Applications of Artificial Intelligence & Expert Systems, 1990, Vol, pp.86-90.

43. Chang, T.C., Expert process planning for manufacturing, Addison-Wesley Publishing Company, 1990.

44. Frank, W., Sauve, B., "Expert systems for manufacturing and testing applications", International Journal of Computer Applications In Technology, 1990, Vol.3, No.3, pp.262-268.

45. Chaudhury, A., Rao, H.R., "A knowledge-based approach to CIM modelling", Journal of Intelligent Manufacturing, 1991, Vol.2, pp.232-234.

46. Cha, J., Rao, M., Zhou, J., Zhao, Z., Guo, W., "New process on integrated environment for intelligent manufacturing automation", Proceedings of the 1991 IEEE International Symposium On Intelligent Control, August 13-15, 1991, Arlington, Virginia, pp. 7-12.

47. Raggenbass, A., Reissner, J., "An expert system as a link between computer aided design and combined stamping-laser manufacture", Journal of Engineering Manufacture, 1991, Vol.205, No.B1, pp.25-34.

48. Widmer, N.S., Nof, S.Y., "Design of a knowledge-based performance process control", Computers and Industrial Engineering", 1992, Vol.22, No.2, pp.101-114.

49. Taylor, A., "How intelligent KBS, expert, real time systems will provide the key to integrating information and flow of factory 2001", IEEE Conference Publication n323, 1990, pp.61-65.

50. Mulder, M.C., "The barriers to widespread use of intelligent robots and manufacturing machines", Robotics & Computer-Integrated Manufacturing, 1990, Vol.7, No.3/4, pp.229-242.

51. Grieve, R.J., Smith, G.W., "Machine and component specification for flexible manufacturing systems for metal cutting processes", Flexible Manufacturing Systems, method and Studies (Editor: A.Kusiak), 1986, Elsevier Science Publishers B.V..

52. Ben-Arieh, D., Moodie, C.L., Chu, C.C., "Knowledge-based scheduling under unpredicted conditions: two approaches", International Journal of Production Research, 1989, Vol.27, No.5, pp.869-882.

53. Merchant, M.E., "World trends of importance to intelligent robotic and CIM systems", Robotics & Computer-Integrated Manufacturing, 1990, Vol.7, No.3/4, pp.255-261.

54. Nitschke, N., Tseng, M., "Overview and examples of digital's AI applications in manufacturing", Proceedings of 1st Conference on Artificial Intelligence and Expert Systems in manufacturing, March, 1990, pp.45-56.

55. Larner, D.L., "Factories, Objects, & Blackboards", AI EXPERT, April, 1990, pp.38-45.

56. Megale, A., Martins, F., Sakamoto, F., Bueno, A.L., Rodrigues, V., "Using a blackboard architecture in CAD-CAM systems integration", Proceedings of the Third International Conference On Industrial & Engineering Application of Artificial Intelligence & Expert Systems, 1990, Vol.1, pp.131-140.

57. Eloranta, E., Syrjanen, M., Torma, S., "Knowledge-based tool for manufacturing systems design", Computer-Integrated Manufacturing Systems, 1990, Vol.3, No.3, No.163-170.

58. Holloway, L.E., Paul, C.J., Strosnider, J.K., Krough, B.H., "Integration of behavioural fault-detection models and an intelligent reactive scheduler", Proceedings of the 1991 IEEE International Symposium on Intelligent Control, August 13-15, 1991, Arlington, Virginia, pp.134-139.

59. Schaefer, P., 'Using qualitative reasoning for robot task planning", SPIE Vol.1612 Cooperative Intelligent Robotics in Space II, 1991, pp.139-148.

60. Wang, J., "Multiple-objective optimization of machining operations based on Neural Network", International Advanced Manufacturing Technology, 1992, Vol.7, pp.1-9.

61. Barschdorff, D., Monostori, L., "Neural networks - Their applications and perspectives in intelligent machining", Computers in Industry, Vol.17, 1991, pp.101-119.

62. Franklin, J.A., "Learning a nonlinear model of a manufacturing process using multi layer connectionsist networks", Proceedings of 5th IEEE International Symposium on intelligent Control, 1990, sept. 5-7, Philadelphia, Pen. USA, pp.404-409.

63. Sutton, J.C., "Manufacturing applications of neural networks for the 90s", Proceedings of manufacturing international- 1992, Dallas, TX, USA, 1992, March 29-April 1. pp.177-189.

64. Hillion, H.P., Proth, J.M., "Using timed petri nets for the scheduling of job-shop systems", Engineering Costs and Production Economics, 1989, Vol.17, pp.149-154.

65. Jockovic, M., "An application of petri nets in the control system of the FTC", Microprocessing and Microprogramming, 1988, Vol.24, pp.681-686.

66. Egilmez, K., Kim, S.H., "Zone logic for knowledge-based control: formalization and application", IEEE, 1989, pp.607-613.

67. Egilmez, K., Kim, S.H., "A logical approach to knowledge based control", Journal of Intelligent Manufacturing, 1990, Vol.1, pp.59-76.

68. Willson, R.G., Krogh, B.H., "Petri net tools for the specification and analysis of discrete controllers", IEEE Transactions on software Engineering", 1990, Vol.16, No.1, pp.39-50.

69. Jockovic, M., Vukobratovic, M., "An approach to the modeling of the highest control level of flexible manufacturing cell", Robotica, 1990, Vol.8, pp.125-130.

70. Jockovic, M., Vukobratovic, M., Ognjanovic, Z., "A contribution to the organization of an expert system for process control of FMC", Robotics & Computer-Integrated Manufacturing, 1990, Vol.7, No.3/4, pp.297-302.

71. Teng, S.H., Black, J.T., "Autonomous cell control system design with petri nets", PED-Vol.53, Design, Analysis, and Control of Manufacturing cells, ASME 1991, pp.157-172.

72. Harhalakis, G., Nagi, R., Proth, J.M., "Design of manufacturing systems: a bottom-up approach based on petri nets", IFAC-INCOM'92, pp.678-682.

73. Besombes, B., Alla, E.M., "Application of generalised bond-graphs and continuous petri nets to modeling industrial processes and manufacturing systems", IFAC-INCOM'92, pp.617-622.

74. Harhalakis, G., Lin, C.P., Muro-Medrano, P.R., "UPN: a Petri Net based graphical representation of company policy specifications in CIM", IFAC-INCOM ,92, Toronto, Canada, 1992, pp.623-627.

75.. Blanchfield, P, Benzeltout, B., Rhodes, D.J., "Multi-Robot cell controller using an expert system approach", University of Nottingham, England, 1985, pp.229-235.

76. Alexander, S.M., "A hierarchical framework for expert manufacturing planning and control systems", 1989, PED-Vol.2, pp.205-209.

77. Hirsch, P., Katke, W., Meier, M., Stillman, R., "Interfaces for knowledge base builder's control knowledge", IEEE, 1986, pp.78-87.

78. Johnston, D.F., "Small batch, precision manufacturing using a prototype automated cell", Proceedings of the 2nd International conference on computer-aided production Engineering, 1987, pp.145-153.

79. Costa, A., Garetti, M., "Design of a control system for a flexible manufacturing cell", Journal of Manufacturing System, 1985, Vol.4, No.1, pp.65-84.

80. Spur, G., Seliger, G., Viehweger, B., "Cell concepts for flexible automated manufacturing", Journal of Manufacturing Systems, 1986, Vol.5, No.3, pp.171-179.

81. Gupta, M.C., Judt, C., Gupta, Y.P., Balakrishnan, S., "Expert scheduling system for a prototype flexible manufacturing cell: a framework", Computers & Operations Research, 1989, Vol.16, No.4, pp.363-378.

82. Petriu, E.M., Basran, S., "Resource Management for a multi-arm robotic assembly cell", Canadian Programmable Control & Automation Technology Conference & Exhibition, October 12-13, Toronto, Canada, 1988, pp.4.1-4.5.

83. Farhoodi, F., "A knowledge-based approach to dynamic job-shop scheduling", International Journal of Computer Integrated Manufacturing, 1990, Vol.3, No.2, pp.84-95.

84. Fussell, P.S., Wright, P.K., Bourne, D., "A design of a controller as a component of a robotic manufacturing system", Journal of Manufacturing Systems, 1987, Vol.3, No.1, pp.1-11.

85. Harhalakis, G., Nagi, R., Proth, J.M., "Hierarchical modeling approach for production planning", IFAC-INCOM '92, Toronto, Canada, 1992, pp.21-26.

86. Cho, H., Wysk, R.A., "A robust adaptive scheduler for an intelligent workstation controller", International Journal of Production Research, 1993, Vol.31, No.4, pp.771-789.

87. Seidmann, A., "Intelligent control schemes for automated storage and retrieval systems", International Journal of Production Research, 1988, Vol.26, No.5, pp.931-952.

88. Rueb, K.D., Wong, A.K.C., "Knowledge-based visual part identification and location in a robot workcell", International Journal of Mechanical Tools Manufacturing, 1988, Vol.28, No.3, pp.235-249.

89. Larin, D.J., "Cell control: what we have, what we will need", Manufacturing Engineering, January, 1989, pp.41-48.

90. Newman, L., Bell, R., "The modeling of Multi-cell flexible manufacturing facilities", IEEE Conference publication n359, 1990, pp.285-290.

91. Santamarina, G., Chen, C., Lee, S., "An application of C++ to manufacturing system control", Computers and Industrial Engineering, 1991, Vol.21, No.1-4, pp.565-570.

92. Pang, G.K.H., "An expert adaptive control scheme in an intelligent process control system", Proceedings of the 1991 IEEE International Symposium on Intelligent Control, August 13-15, 1991, Arlington, Virginia, pp.13-18.

93. Moore, R.L., "G2: a software platform for intelligent process control", Proceedings of the 1991 IEEE International Symposium on Intelligent Control" August 13-15, 1991, Arlington, Virginia, pp.1-5.

94. Sanii, E.T., Davis, R.E., "Computer aided process planning in a hierarchically controlled manufacturing environment", Proceedings of Manufacturing International '90, Vol.1, pp.233-241.

95. Chen, H.G., Guerrero, H.H., "A rule based robot scheduling system for manufacturing cells", Journal of Intelligent Manufacturing, 1992, Vol.3, pp.285-296.

96. Wylie, R.H., Kamel, M., 'Model-based knowledge organization: a framework for constructing high-level control systems", Expert systems with applications, 1992, Vol.4, pp.285-296.

97. O'Grady, P.J., Bao, H., Lee, K.H., "Issues in intelligent cell control for flexible manufacturing systems", Computers in Industry, 1987, pp.25-36.

98. Johnston, D.F., Brunelle, D., "CNC servant low cost DNC for small shops", Canadian CAD/CAM Conference, 1989, Toronto, Canada, pp.125-129.

99. Pham, D.T., Hu, H., Dote, J., "A transputer-based system for locating parts and controlling an industrial robots", Robotica, 1990, Vol.8, pp.97-103.

100. Lombardi, F., Boer, C.R., "A knowledge based on-line control procedure for flexible manufacturing systems", 1989, PED Vol.2, pp.369-381.

101. Erdelyi, F., Santha, C., "Monitoring tasks on boring and milling production cells", Computers in Industry, 1986, Vol.7, pp.65-71.

102. Manivannan, S., Cotton, R., "A rule based approach to robotic collision avoidance in a flexible assembly cell", PED-Vol.53, Design, Analysis, and Control of Manufacturing cells, ASME 1991, pp.235-244.

103. Rege, A., Agogino, A.M., "Sensor-integrated expert system for manufacturing and process diagnostics", 1989, PED-Vol.2, pp.67-83.

104. Rowen, R., "Diagnostic systems for manufacturing", AI EXPERT, April, 1990, pp.28-37.

105. Petriu, E.M., "Absolute position recovery for automated path-guided vehicles", Robots 12 and Vision '88 Conference Proceedings, June 5-9, Detroit, Michigan, 1988, Vol.1, pp.35-46.

106. Monostori, L., Bartal, P., Zsoldos, L., "Concept of a knowledge based diagnostic system for machine tools and manufacturing cells", Computers In Industry, Intelligent Manufacturing Systems-IMS'89, pp.95-102.

107. Jamshidi, M., "Sensors and interfacing in robotics and manufacturing", Robotics & Computer-Integrated Manufacturing", 1990, Vol.7, No.3/4, pp.143-253.

108. Pun, L., Archimede, B., Berard, C., "Intelligent learning aid for an intelligent FMS-Monitoring process", Computers in Industry, 1991, Vol.17, pp.225-235.

109. Hermann, G., "The evolution of numerical control units in the light of integration", Computers in Industry, 1991, Vol.17, pp.341-347.

110. Ward, T., Ralston, P.A.S., Karwowsk, W., "ITONNUS: expert system for machining on a lathe", Journal of Intelligent Manufacturing, 1991, Vol.2, pp.253-363.

111. Sturges, R.H., "Monitoring milling processes through AE and tool/part geometry", Journal of Engineering for Industry, 1992, Vol.114, pp.8-14.

112. Carvalho, S.V., Sahraoui, A.E.K., Serrano, A., "Design and implementation of a monitoring system for NC machines", Computers in Industry, 1992, Vol.18, pp.59-66.

113. Famili, A., Turney, P., "An intelligent supervisory system for process optimization and knowledge refinement of an industrial process", NRC 33199.

114. Schmucker, Kurt., "Using Objects to package user interface functionality", Journal of Object Oriented Programming, April/May, 1988, pp.40-45.

115. Belaid, F., Herin-Aime, D., "Explanations for an object-oriented knowledge base", Fourth International Expert System Conference, London 7-9, June, 1988, pp.61-72.

116. Pope, S.T., "Building Smalltalk-80-based computer music tools", Journal of Object-Oriented Programming, April/May 1988, pp.6-10.

117. Dewan, P., "Object-Oriented Editor Generation", Journal of Object Oriented Programming, July/August, 1990, pp.35-49.

118. Johson, E.R., Foote, B., "Designing Reusable Classes", Journal of Object-Oriented, June/July, 1988, pp.22-35.

119. Christodoulakis, D.N., "Petri net semantics of Smalltalk-80", Microprocessing and microprogramming, 1988, Vol.24, pp.267-272.

120. Ege, R., "Improving object-oriented user interfaces with constraints", Information and Software Technology, 1991, Vol.33, No.2, pp.143-149.

121. Kusiak, A., Szczerbick, E., Vujosevic, R., "Intelligent design synthesis: an object-oriented approach", International Journal of Production Research, 1991, Vol.29, No.7, pp.1291-1308.

122. Goerner, A.A., Hines, M.L., "Instrumenting Intelligent Manufacturing applications: an object oriented approach using probes and generalized daemons", Proceedings of the Fourth International Conference on Industrial & Engineering applications of Artificial Intelligence & expert systems", June 2-5, 1991, pp.192-199.

123. Wang, M.T., Chang, T.C., "Feature recognition for automated process planning", Proceedings of Manufacturing International'90, Vol.2, pp.49-64.

124. Onosato, M., Iwata, K., "Virtualworks: Building a virtual factory with 3-D modelling and object-oriented programming techniques", IFAC_INCOM'92, pp.281-286.

125. Walker, I., "A Smalltalk/V VLSI CAD application", Computer-Aided Engineering Journal, April 1991, Vol.8, No.2, pp.47-53.

126. Menga, G., Morisio, M., Mancin, M., "A framework for object oriented design and prototyping of manufacturing systems", Proceedings of the 1991 IEEE International Conference on Robotics and Automation, April, 1991, Sacramento, California, pp.128-135.

127. Graham, J.H., Alexander, S.M., "Object-oriented software for diagnosis of manufacturing systems", Proceedings of the 1991 IEEE International Conference on Robotics and Automation, April, 1991, Sacramento, California, pp.1966-1971.

128. Sturzenbecker, M.C., "Building an object-oriented environment for distributed manufacturing software", Proceedings of the 1991 IEEE International Conference on Robotics and Automation, April, 1991, Sacramento, California, pp.1972-1979.

129. Joannis, R., Krieger, M., "Object-Oriented approach to the specification of manufacturing systems", Computer-Integrated manufacturing, 1992,Vol.5, No.2, pp.133-145.

130. Wu, S.M., Ermer, D.S., "Maximum profit as the criterion in the determination of the optimum cutting conditions", Journal of Engineering for Industry, 1966, Vol.83, pp.435-442.

131. Field, M., Zlatin, M., Williams, R., Kronenberg,M., "Computerized determination and analysis of cost and production rates for machining operations: part 2- milling, drilling, reaming, and tapping", Journal of Engineering for Industry, 1969, Vol.91, pp.585-595.

132. Bhattacharyya, A., Faria-Gonzalez, R., Ham, I., "Regression analysis for predicting surface finish and its application in the determination of optimum machining conditions", Journal of Engineering for Industry, August, 1970, pp.711-714.

133. Barrow, G., "Tool life equations and machining economics", Proceedings of the Twelfth International Machine Tool Design and Research Conference, September 15-17, 1971, pp.481-493.

134. Yellowley, I., Gunn, E.A., "The optimal subdivision of cut in multi-pass machining operations",International Journal of Production Research, 1989, Vol.27, No.9, pp.1593-1588.

135. Ermer, D.S., "Optimization of the constrained machining economics problem by geometric programming", Journal of Engineering for Industry, 1971, Vol.93, pp.1067-1072.

136. Iwata, K., Morotsu, Y., Iwatsubo, T., Fujii, S., "A probabilistic approach to the determination of the optimum cutting conditions", Journal of Engineering for Industry, 1972, November, pp.1099-1107.

137. Hati, S.K., Rao, S.S., "Determination of optimum machining conditions- deterministic and probabilistic approaches", Journal of Engineering for Industry, 1976, Vol.99, pp.354-359.

138. Philipson, R.H., Ravindran, A., "Application of goal programming to machinability data optimization", Transactions of the ASME, 1978, Vol.100, pp.286-291.

139. Iwata, K., Murotsu, Y., Oba, F., "Optimization of cutting conditions for multi-pass operations considering probabilistic nature in machining processes", Journal of Engineering for Industry, 1977, February, pp.210-217.

140. Chang, T.C., Wysk, R.D., Davis, R.P., Choi,B., "Milling parameter optimization through a discrete variable transformation", International Journal of Production Research, 1982, Vol.20, No.4, pp.507-516.

141. Wang, W.P., "Solid modelling for optimizing metal removal of three-dimensional NC End Milling", Journal of Manufacturing Systems, 1988, Vol.7, No.1, pp.57-65.

142. Malakooti, B., Devirpasad, J., "An interactive multiple criteria approach for parameter selection in metal cutting", Operation Research, 1989, Vol.37, No.5, pp.805-818.

143. Balas, E., Ho, A., "Set covering algorithms using cutting planes, heuristics, and subgradient optimization: a computational study", Mathematical Programming, 1980, Vol.12, pp.37-60.

144. Chen, C.M., Lin, S.F., Madsen, N.H., Beckett, R.E., "CAPP for small shops",Proceedings of Manufacturing International'90, 1990, Vol.1, pp.249-255.

145. Tsatsoulis, C., Kashyap, R.L., "A system for knowledge-based process planning", Artificial Intelligence in Manufacturing, 1988, Vol.3, No.2, pp.61-75.

146. Brown, P.F., Mclean, C.R., "Interactive process planning in the AMRF", PED Vol.12, 1988, pp.245-261.

147. Vancza, J., Markus, A., "Genetic algorithms in process planning", Computers in Industry, 1991, Vol.17, pp.181-194.

148. Joshi, S., Chang, T.C., Liu, C.R., "Process planning formalization in an AI framework", The International Journal for Artificial Intelligence, 1986, Vol.1, No.1, pp.45-53.

149. Pande, S.S., Palsule, N.H., "GCAPPS - a computer-assisted generative process planning system for turned components", Computer-Aided Engineering Journal, August 1988, Vol.5, N0.4, pp.164-168.

150. Li, R.K., Bedworth, D.D., "A framework for the integration of computer-aided design and computer sided process planning", Computers Industrial Engineering, 1988, Vol.14, No.4, pp.395-413.

151. Kals, H.J.J., van der Wolf, A.C.H., "A computer aid in the optimization of turning conditions in multi-cut operations", Annals of the CIRP, 1978, Vol.27, pp.465-469.

152. Challa, K., Berra, P.B., "Automated planning and optimization of machining processes: A systems approach", Computer & Industrial Engineering, 1976, Vol.1, pp.35-46.

153. Phillips, R.H., Arunthavananathan, V., Zhou, X., "MICROPLAN: a microcomputer based expert system for generative process planning", 1989, PED-Vol.2 pp.263-273.

154. Korde, U.P., Bora, B.C., Stelson, K.A., Riley, D.R., "Computer-aided process planning for turned parts using fundamental and heuristic principles", Journal of Engineering for Industry, 1992, February, Vol.114, pp.31-40.

155. Small, D.H., Heyman, B., Schroeder, M.,Troxell, W.O., "INTELMIL-an intelligent milling interface for job-shop applications", 1989, PED-Vol.2, pp.361-367.

156. Ssemakula, M.E., Rangachar, R.M., "The prospects of process sequence optimization in CAPP systems", Computers industrial Engineering, 1989, Vol.16, No.1, pp.161-170.

157. Willis, D., Donaldson, I.A., Ramage, A.D., Murray, J.L., "A knowledge-based system for process planning based on a solid modeller", Computer-Aided Engineering Journal, February 1989, Vol.6, N0.1, pp.21-26.

158. Shan, X.H., Nee, A.Y.C., Poo, A.N., "An integrated CAPP system for parts machined on single spindle swiss-type automatics", Computers in Industry, 1990, Vol.14, pp.281-291.

159. Joseph ,A.T., Davies, B.J., "EXCAP - an expert process planning system for turned computers" IEEE Conference Publication, 1990, No.322, pp.130-135.

160. Kusiak, A., Intelligent Manufacturing Systems, Prentice-Hall, 1990.

161. Vittal, V.N., Jain, V.K., Shankar, K., "A computer-aided process planning system for rotational parts(CAPP-RP) for an FMS environment", International Journal of Computer Applications In Technology, 1990, pp.61-69.

162. Pande, S.S., Prabhu, B.S., "An expert system for automatic extraction of machining features and tooling selection for automats", Computer-Aided Engineering Journal, August 1990, Vol.7, N0.4, pp.99-103.

163. Laakko, T., Mantylia, M., Mantylia, R., "Feature models for design and manufacturing", Proceedings of the Twenty-third Annual International Conference on system science, 1990, Vol.2, pp.445-454.

164. Torvinen, S.J., "Integration of a CIM tool management system to an intelligent feature-based process planning system", Computers in Industry, 1991, Vol.17, pp.207-216.

165. Maropoulos, P.G., "Cutting tool selection: an intelligent methodology and its interfaces with technical and planning functions', Proceedings of Institution of Mechanical Engineering, 1992, Vol.206, pp.60.

166. Juri, A.H., Saia, A., Pennington, A.D., "Reasoning about machining operations using feature-based models", International Journal of Production Research, 1990, Vol.48, No.1, pp.153-171.

167. Subbarao, P.C., "Feature based process planning system for machined components using expert systems technology", Proceedings the Fourth International Conference On Industrial & Engineering applications of Artificial Intelligence & Expert Systems, 1991, Vol.1, pp.171-177.

168. Lee, I.B.H., Lim, B.S., Nee, A.Y.C., "IKOOPPS: an intelligent knowledge based object oriented process planning system for the manufacture of progressive dies", Expert systems, February 1991, Vol.8, No.1, pp.19-33.

169. Owusu-Ofori, S., Chen, C.S., "A knowledge-based approach to form-feature recognition from engineering drawings", Proceedings of Manufacturing International '90, 1990, Vol.1, pp.57-66.

170. Wang, M.T., "An object-oriented feature based CAD/CAPP/CAM integration framework", DE-vol.32-1, Advances in Design Automation, 1991, Vol.1, pp.109-116.

171. Srihari, K., Greene, T.J., "Alternate routings in CAPP implementation in a FMS", Computers & Industrial Engineering, 1988, Vol.15, pp.41-50.

172. Shannon, Robert E., "Simulation Modelling in Manufacturing", Simulation and Artificial Intelligence In Manufacturing Conference Proceedings, Long Beach, California, October 14-16, 1987, pp.1-13.

173. Vujosevic, R., "Object-oriented visual interactive simulation", Proceedings of the 1990 Winter Simulation Conference, New Orleans, LA USA, 1990, December 9-12, pp.490-498.

174. Atabakhsh, H., Chan, A.W., "Application of Object oriented technology to discrete event simulation", Proceedings of SimTec'92, Houston, TX. November 4-6, 1992, pp.286-291.

175. Thomasm, T., Madsen, J., "Object oriented programming languages for developing simulation-related software", Proceedings of the 1990 Winter Simulation Conference, New Orleans, LA USA, 1990, December 9-12, pp.482-485.

176. Atabakhsh, H., Chan, A.W., "Use of distributed intelligent objects in discrete event simulation", Proceedings of the 1993 Object Oriented Simulation Conference, San Diego, CA. January 17-20, 1993, pp.121-126.

177. Ulgen, O., Thomasma, T., "SmartSim: an object-oriented simulation program generator for manufacturing systems", International Journal of Production Research, 1990, Vol.28, No.9, pp.1713-1730.

178. Verchueren, ir. Ad. C., 'An object-oriented Design and Simulation system for VLSI", Microprocessing and Microprogramming, 1990, Vol.30, pp.241-246.

179. Baldassari, M., Bruno,G., "PROTOB: an object oriented methodology for developing discrete event dynamic systems", Computer Language, 1991, Vol.6, No.1, pp.39-63.

180. Taylor, Christopher D.A., "Defining and implementation a Unified Instruction Set for a flexible manufacturing cell", Fourth Year Thesis, University of Ottawa, April 1989.

181. Szukalo, E.W., "A Unified Instruction Set for flexible manufacturing cells" Fourth Year Thesis, University of Ottawa, Dec. 1990

182. Fahim, A., Tolkamp, R., "UniSet- A flexible Manufacturing Cell programming, simulation and management Environment", INCOM '92 Present, pp.273-277.

183. Fahim, A., Choi, K.H., "Real time cell control for flexible manufacturing", 2nd IFAC Workshop on Architectures for real time control, Seoul, Korea, Aug. 1992.

184. Chang, T.C., Wysk, R.A., Wang, H.P., Computer-aided manufacturing, Prentice-Hall, Enllewood Cliffs, New Jersey, 1991.

185. Wasserman, P.D., "Neural Computing: Theory and Practice." Van Nostrand Reinhold, pp.43-59, 1989

186. Kosko, B., Neural Networks and Fuzzy Systems, Prentice-Hall, Inc. 1991.

187. Ahmad, S., Tesauro, G., "Scaling and Generalization in Neural Network: A case study", Advances in Neural Information Processing System i, Morgan Kaufman, Palo Alto, CA. 1989. pp.160-168.

188. Bourne, J.R., Object-Oriented Engineering building engineering systems using smalltalk-80, Asken Associates, Inc., 1992.

189. Machinability Data Center, Machining Data Handbook second Edition, Metcut Research Associates, Inc., 1972.

190. Rumbaugh, J., Blaha, M., Premerlan, W., Eddy, F., Loresen, W., Object-Oriented Modeling and Design, Prentice-Hall, 1991.

191. Fahim, A., "UniSet and UniSet Environment for Flexible Manufacturing Cell Seting-up, Programming, Simulation, and Control", Personal Note, Department of Mechanical Engineering, University of Ottawa, 1988.

192. Tokemp, R., "UniSet A Flexible Manufacturing Cell programming, Simulation, Control, and Management Environment", Msc thesis, Department of Mechanical Enginnering, University of Ottawa, 1984.

193. Paul, R.P.C., Robot Manipulators: Mathematics, Programming and Control, 1981, MIT Press.

194. Klumpp, A.R., "Singularity-free Extraction of a Quaternion from a Direct Cosine Matrix", Journal of Spacecraft and Rockets, 1976, Vol.13, No.12, pp.754-755.

APPENDIX A
The Object-Modeling Technique (OMT)

The Object-Modeling Technique (OMT) is a comprehensive analysis and design methodology that consists of the three phases of creating the Object Model, the Dynamic Model, and the Functional Model. OMT provides a large number of outstanding advantages over conventional analysis and design approaches. It orders the thinking process, and separates the design activities. It should be emphasized that the details of implementation are of no importance at this modeling stage. The concepts used in OMT may look and sound foreign to those who are not accustomed to this methodology. This appendix serves as a short precis on the three phases of OMT and their terminology.

A.1 Object Model Notation and Basic Concept

In the first phase, the Object Model of the system is created. This model captures the static structure of a system by showing the objects in the system, relationships between the objects, and the attributes and operations that characterize each class of objects. In this section, basic notations involved in the Object Model are described.

A.1.1 Class

A class in essence is a description of a set of objects with similar characteristics, attributes and behaviors. Objects that share the same behavior should belong to the same class. In general terms they are like templates for specifying an object behavior.

Figure A.1 summarizes object modeling for classes. A class is represented by a box which may have as many as three regions. The regions contain, from top to bottom: class name, list of attributes, and a list of operations.

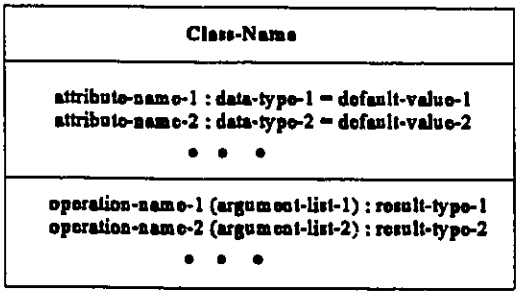


Figure A.1: Summary of object modeling notation for classes

Each attribute name may be followed by optional details such as its type and default value. The operation name may be followed by optional details such as argument list and result type. Attributes and operations may, or may not be shown. It depends on the level of detail desired.

A.1.2 Generalization (Inheritance)

At the abstract level of creating classes, inheritance is called generalization. Generalization is the relationship between a class and one or more refined versions of it. The class being refined is called the *superclass* and each refined version is called a *subclass*. For example, **Machine** is the

superclass of **Robot** and **NC Machine**. Attributes and operations common to a group of subclasses are attached to the superclass and shared by each subclass.

The notation for generalization is a triangle connecting a superclass to its subclasses as shown in Figure A2. The superclass is connected by a line to the apex of the angle. The subclasses are connected by lines to a horizontal bar. For convenience, the triangle can be inverted, and subclasses can be connected to both the top and bottom of the bar.

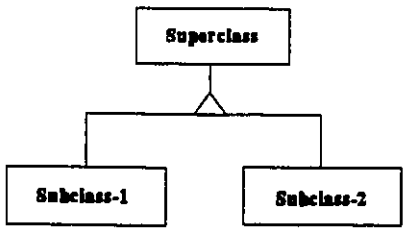


Figure A.2: Notation for Generalization

A.1.3 Association

An association describes the logical link between two or more objects. Given an object B associated with another object A through a relationship R. A may invoke any operation that is applicable to B, and is accessible to A. A is visible to B, i.e., A should know B. For A to know B, it should have an instance variable to point to B. Formally, an association describes a group of links with common structure and common semantics. For example, a robot *delivers* a part.

All the association links leading into an object have to be from objects of the same class. Associations are inherently bidirectional. The name of a binary association usually reads in a

particular direction, but the binary direction can be traversed in either direction. As an example, "robot deliver part" can be reversed to read as "part is delivered by robot".

Figure A.3 shows a one-to-one association with role names. The OMT notation for an association is a line between classes. An association name may be omitted if a pair of classes has a single association whose meaning is obvious. It is good to arrange the class to read from left to right, if possible.

Figure A.3: Notation for one-to-one Association

Figure A.4 shows a ternary association. Three classes, class-1, class-2, and class-3, match together by an association name.

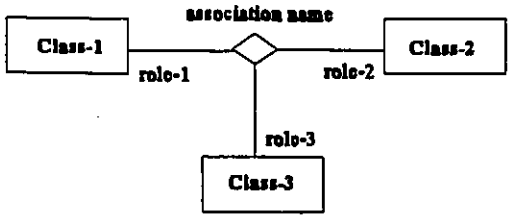


Figure A.4: Ternary Association

A role is one end of an association. A binary association has two roles, each of which may have a role name that uniquely identifies one end of the association. Roles provide a way of viewing a binary association as traversal from one object to a set of associated objects. Each role on a binary

association identifies an object or set of objects associated with an object at the other end. From the point of view of an object, traversing the association is an operation that yields related objects. The role name is a derived attribute whose value is a set of related objects. Use of role names provide a way of traversing associations from an object at one end, without explicitly monitoring the association.

A.1.4 Multiplicity

Multiplicity specifies how many instances of one class may relate to a single instance of an associated class. Multiplicity contains the number of related objects. Multiplicity is often described as being "one" or "many", but more generally, it is a subset of the nonnegative integers. Generally, the multiplicity value is a single interval, but it may be a set of disconnected intervals. For example, the number of doors on a car is 2 or 4.

As shown in Figure A.5, object diagrams indicate multiplicity with special symbols at the ends of association lines. In the most general case, multiplicity can be specified with a number or a set of intervals, such as "1" (exactly one), "1+" (one or more).

There are special line terminators to indicate certain common multiplicity values. A solid bullet is the OMT symbol for "many", meaning zero or more. A hollow bullet indicates "optional", meaning zero or one. A line without multiplicity symbols indicates a one-to-one association. Usually, the multiplicity symbols are written next to the end of the line, for example, "1+" to indicate one or more.

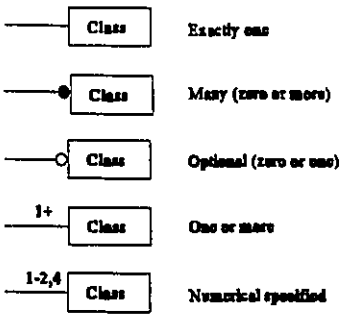


Figure A.5: Notation for Multiplicity of Association

A.1.5 Ordering

Usually the objects on the "many" side of an association have no explicit order, and can be regarded as a set. Sometimes, however, the objects are explicitly ordered. For example, Figure A.6 shows a workstation screen containing a number of overlapping windows. The windows are explicitly ordered, so only the topmost window is visible at any point on the screen. The ordering is an inherent part of the association. An ordered set of objects on the "many" end of an association is indicated by writing "{ordered}" next to the multiplicity dot for the role.

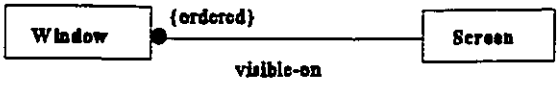


Figure A.6: Ordered Sets in an Association

A.1.6 Aggregation

Aggregation is a strong form of association in which an aggregate object is made of several components. The aggregation is semantically an extended object that is treated as a unit in many operations, although physically it is made of several lesser object. As an example, a vise that is part of an NC machine has an aggregate relation to the machine. However, a vise that is added to a machine would have an association with that machine. In the aggregate case, the vise is part-of in the association case, the machine uses the vise. The notation for the aggregation is shown in Figure A.7. A single aggregate object may have several parts; each part-whole relationship is treated as a separate aggregation in order to emphasize the similarity to association.

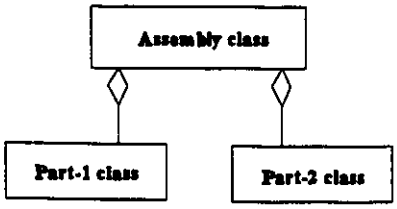


Figure A.7: The Notation for the Aggregation

A.2 Dynamic Model

Temporal relationships are difficult to understand. A system can be understood by first examining its static structure, that is, the structure of its objects and their relationships to each other at a single moment in time. Then changes to the objects and their relationships are examined over time. Those aspects of a system that are concerned with time and changes are the Dynamic Model, in contrast with the static, or Object Model.

The major Dynamic Modeling concepts are *events*, which represent external stimuli, and *states*, which represent values of objects. The *state diagram* is a graphical representation of finite machines that have been considered as a standard concept. The organizations of events and states with notations will be addressed.

A.2.1 Events

An *event* is something that happens at a point in time, such as *Mill Start*. An event has no duration. An event is some occurrence that may cause the state of a system to change.

A.2.2 States

A *state* is an abstraction of the attribute values and links of an object. It specifies the response of the object to input events. The response to an event received by an object may vary qualitatively depending on the exact values of its attributes. However, in the same state the response is qualitatively the same for all values. Moreover, the response is qualitatively different for similar

values in different states. A state has duration, it occupies an interval of time, while events represent points in time. The typical state diagram is shown in Figure A.8.

Figure A. 8: Notation for the State

A.2.3 Notation of Relationships between States and Events

A state diagram relates events and states. A state diagram is a graph whose nodes are states and whose directed arcs are transitions labeled by event names. Figure A.9 shows the notations for the Dynamic Model. With reference to the figure an initial state is shown by solid circle. A final state is shown by a bull's eye.

Conditions can be used as *guard* on transitions. A guarded transition fires when its events occur, but only if the guard condition is true. A guard condition on a transition is shown as a Boolean expression in brackets following the event's name.

An activity in a state can be expanded as a low level state diagram, each state representing one step of the activity.

One state can contain two (or more) activities which perform concurrently. The internal steps of the activities are not synchronized, but both activities must be completed before the state can be changed.

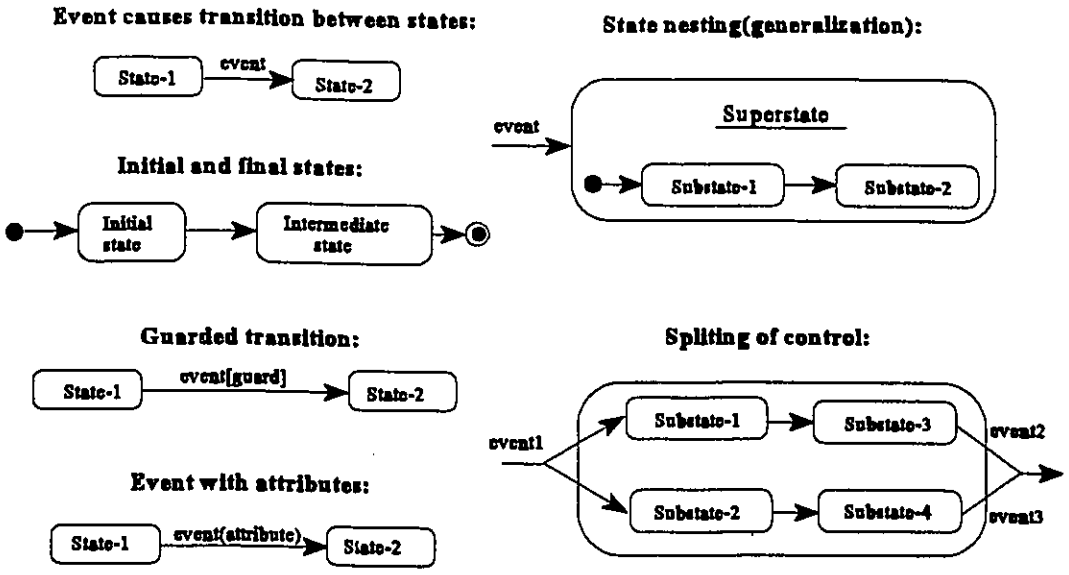


Figure A.9: Notation for the Dynamic Model

A.3 Functional Model

The functional model shows how output values in a computation are derived from input values without regard for the order in which the values are computed. The functional model consists of multiple data flow diagrams which show the flow of values from external inputs, through operations and internal data stores, to external outputs.

A.3.1 Processes

A *process* transforms data values. At the implementation stage, a process is generally referred to as a *method*. The target object is either along the input path, the output path, or both. In

some cases, the target object is implicit. A process is drawn as an ellipse containing a description of the transformation, usually its name. Each process has a fixed number of input and output data arrows, each of which carries a value of a given type. The inputs and outputs can be labeled to show their role in the computation, but often the type of value on the data flow is sufficient. Figure A.10 shows the notation for the process.

Figure A.10: Notation for the Process

A.3.2 Actors

Figure A.11: Notation for the Actor Object

An actor is an active object that drives the data flow graph by producing or consuming values. Actors are attached to the inputs and outputs of a data flow graph. An actor is drawn as a rectangle to show that it is an object. Arrows between the actor and the diagram are inputs of the diagram. The notation for the actor is shown in Figure A.11.

A.3.3 Data stores

A data store is a passive object within a data flow diagram that stores data for later access. Unlike an actor, a data store does not generate any operations on its own but merely responds to requests to store and access data. Data stores are classified into aggregate data stores (database), and a single object data store. Aggregate data stores, such as lists and tables, provide access to the data by insertion order or index keys. A single data store is a member of an aggregate data stores.

A data store is drawn as a pair of parallel lines containing the name of the store as shown in Figure A.12. Input arrows indicate information or operations that modify the stored data; this includes adding elements, modifying values, or deleting elements. Output arrows indicate information received from the store. This includes retrieving the entire value or some component of it.

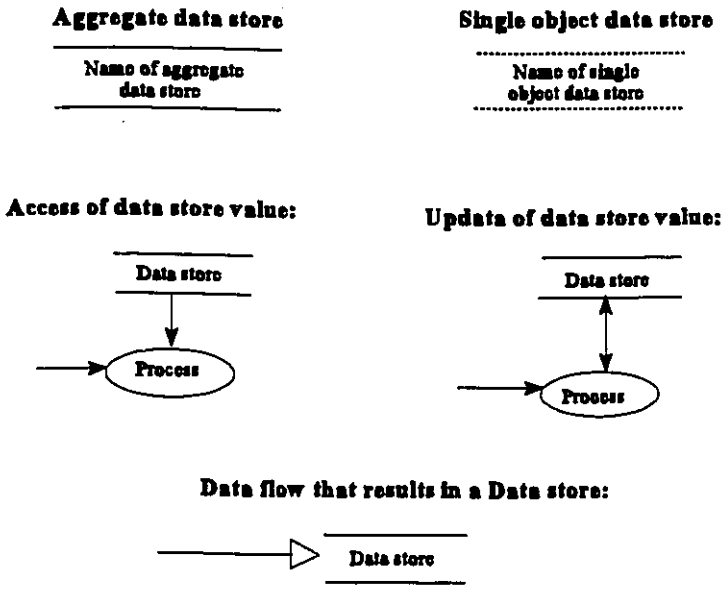


Figure A.12: Notation for the Data Store

APPENDIX B
Transformation of Quaternion from Euler Angle

A quaternion description of the orientation of an object specifies the location of a unit vector, k, around which rotation is considered to occur. It comprises four components,

$$\begin{aligned} Q1 &= \cos\theta/2 \\ Q2 &= k_x \sin\theta/2 \\ Q3 &= k_y \sin\theta/2 \\ Q4 &= k_z \sin\theta/2 \end{aligned} \tag{1}$$

where, k_x, k_y and k_z are the components of the vector in the reference coordinate frame and θ is the angle of rotation around the vector.

A general rotational transformation, Rot(k,θ) for rotation θ about axis k is given by [193]:

$$Rot(k,\theta) = \begin{bmatrix} k_x k_x(1-\cos\theta)+\cos\theta & k_y k_x(1-\cos\theta)-k_z\sin\theta & k_z k_x(1-\cos\theta)+k_y\sin\theta & 0 \\ k_x k_y(1-\cos\theta)+k_z\sin\theta & k_y k_y(1-\cos\theta)+\cos\theta & k_z k_y(1-\cos\theta)-k_x\sin\theta & 0 \\ k_x k_z(1-\cos\theta)-k_y\sin\theta & k_y k_z(1-\cos\theta)+k_x\sin\theta & k_z k_z(1-\cos\theta)-\cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2}$$

The orientation of an object may be described also using Euler angles, ϕ,θ, and ψ. The Euler

transformation, Euler(ϕ,θ,ψ), can be evaluated by multiplying the three rotation matrices together

$$Euler(\phi,\theta,\psi) = Rot(z,\phi)\ Rot(y,\theta)\ Rot(z,\psi) \tag{3}$$

Expending this, yields the Euler transformation,

$$Euler(\phi,\theta,\psi) = \begin{bmatrix} n_x & o_x & a_x & 0 \\ n_y & o_y & a_y & 0 \\ n_z & n_z & o_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} \cos\phi\cos\theta\cos\psi-\sin\phi\sin\psi & -\cos\phi\cos\theta\sin\psi-\sin\phi\cos\psi & \cos\phi\sin\theta & 0 \\ \sin\phi\cos\theta\cos\psi+\cos\phi\sin\psi & -\sin\phi\cos\theta\sin\psi+\cos\phi\cos\psi & \sin\phi\sin\theta & 0 \\ -\sin\theta\cos\psi & \sin\theta\sin\psi & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4}$$

For the same orientation and reference coordinate frame, Equation (2) and (4) are equivalent descriptions, i.e.,

$$Euler(\phi,\theta,\psi) = Rot(k,\theta) \tag{5}$$

The conversion between the two descriptions becomes possible. Given values of ϕ, θ, and ψ, Equation (5) is used to derive k and θ and then the equivalent quaternion description can be obtained. The algorithm for the orientation transformation is shown in figure A.1. This follows the

algorithm suggested by Klump [194] and further refined by Paul [193] to avoid numerical difficulties when dealing with values of θ close to 0° and 180°. Equation (5) also can be used to derive the Euler angles when orientation is described in Quaternion. The reverse conversion is

needed when the robot as a measuring tool to provide cell modeling information in the CFM.

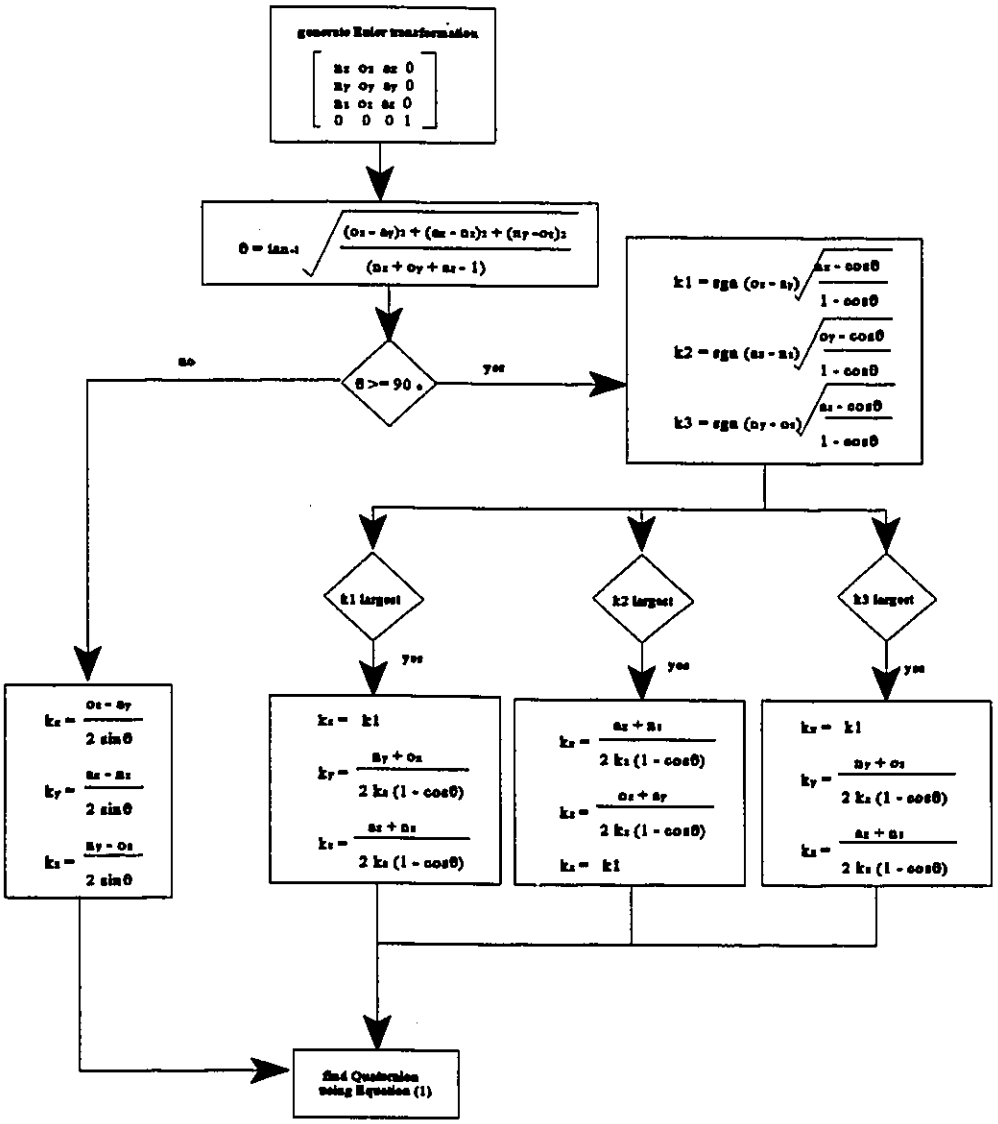


Figure B.1: Algorithm of the Orientation Transformation

APPENDIX C
Validation of Automatic Code Generation and Translation

This appendix examines the validation of the automatically created codes by comparing them to codes which are developed by a user in the target language itself. As examples, two tasks **pick/load** and **make slot** are considered, and two target machine languages VAL II and Word Address are used.

C.1 Pick/Load Code Fragment

Figure C.1 show the code fragment developed in Task level UniSet and UniSet of the pick and load task for the cell configuration given in Figure 7.4. With reference to the figure, all numerical information is hidden in the development stage with Task level UniSet. The Generation Facility generates UniSet commands and the sequence of locations of the end effector of the robot with respect to the robot coordinates.

The comparison between the automatically generated codes and the user developed codes is shown in Table C.1. Robot programming language VAL II is chosen as a target machine language. The user developed VAL II code for the pick and load task is developed for the same cell configuration as before. The important positions, *Mill*, *pick*, and *load*, must be defined, and the most convenient way to define these variables is to use either powered lead through or manual lead

through to place the robot at the desired point and record that point into memory. Statement, **HERE** *Mill*, is used in the lead through procedure to indicate the variable name for the point.

```
{Task Level UniSet}
pick part from:conveyor to:Mill.      ;Pick a part from a conveyor to a mill
load part to:Mill.                    ;Load a part to a mill

{UniSet}
Goto:150@40@120, 30@40@30.  ;Move to the top of a part on a
                            conveyor
Goto:145@110@80, 30@40@30.  ;Move to a part
Close.                      ;Grasp a part
Go:50.                      ;Back off
Goto:100@60@120, 30@40@30.  :Return to the previous location

Goto:-120@160@150, 0@90@10. ;Move to a mill

Goto:-120@200@110, 0@90@10. ;Approach to the load position
Goto:-126@244@100, 0@90@10. ;Move to the load position
Open.                       ;Release a part
Go:70.                      ;Back off
Goto:-120@160@150.          ;Return to the previous location
```

Figure C.1: Task level UniSet and UniSet Code for pick/load Task

All the destination locations in the generated codes are represented in numerical format. The first three numbers depict the position of the end effector and the last three indicate the orientation. These specific formats are translated from UniSet codes by the Translation Facility.

With reference to the table, the two statements in line numbers 1 and 7 employ the different commands, **MOVES** and **APPRO**. Since the approach location is determined by the system, in the automatically-generated code the command **MOVES** is used. Again in statement number 5, the automatically generated code employ the command **MOVES** whose parameters are derived by the code generation algorithm using the rule "via the **Rest Position**"

Table C.1 Comparison between the automatically generated, and the user developed code in VAL II

No	Generated Codes	Developed Codes	Comments
1	MOVES(150,140,120,30,40,30)	APPRO *pick*,30	Approach the part
2	MOVES(145,110,80,30,40,30)	MOVES *pick*	Move to the part
3	CLOSE	CLOSE	Grasp the part
4	DEPART 50	DEPART 50	Back off
5	MOVES(100,60,120,30,40,30)		Return to the previous location
6	MOVES(-120,160,150,0,90,10)	MOVES *Mill*	Move to the mill
7	MOVES(-120,200,110,0,90,10)	APPRO *load*,50	Approach to the load position
8	MOVES(-126,244,100,0,90,10)	MOVES *load*	Move to the load location
9	OPEN	OPEN	Release the part
10	DEPART 70	DEPART 70	Back off
11	MOVES(-120,160,150)	MOVES Mill	Return to the previous location

C.2 Slot Machining Code Fragment

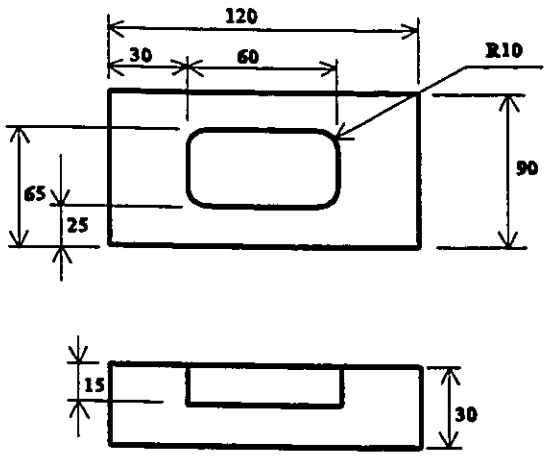


Figure C.2: Geometric data for the slot

The geometric data for the slot is shown in Figure C.2. Its width, length, and depth are 40mm, 60mm, and 15mm, respectively. The corner radius is 10mm. Based on this information, the tool whose diameter is 20mm is selected.

The code developed in Task level UniSet and UniSet for machining the slot is shown in Figure C.3. All detailed arguments in the UniSet code are generated by the Generation Facility. The speed and the feed rate are obtained from the neural network expert system (See Chapter 6).

```
{Task level UniSet}
make slot at:60@45@30; width:40; length:60; depth:15.

{UniSet}
ToolChange;1.                    ; Tool Change for the Slot
Goto,rapid:60@45@32.             ; Move to the top of the center of the Slot
Spindle,cw:900.                  ; Set Spindle speed
Coolant,on.                      ; Set Coolant On
Go:-17.                          ; Driiling
Speed,Unit/min:400.              ; Set Feed rate
Go:0@5@0.                        ; Generating the Slot
Go:10@0@0.
Go:0@-10@0.
Go:-20@0@0.
Go:0@10@0.
Go:10@0@0.
Go:0@5@0.
Go:20@0@0.
Go:0@-20@0.
Go:-40@0@0.
Go:0@20@0.
Go:10@0@0.
Go:17.                           ; Returning the set position
Coolant,off.                     ; Set Coolant Off
Spindle,off                      ; Set Spindle Off
```

Figure C.3: Task Level UniSet and UniSet Code for Machining the Slot

The comparison between the automatically generated code and that developed by a user is shown in Table C.2. Word Address is chosen as the target machine language. With reference to the table, the two code fragments employ the same instructions with the exception of statement number 7. In the automatically generated code, all tool locations are calculated with the system considering the tool offset by itself. However, in the user developed code, the tool offset command itself is used.

Table C.2 Comparison between the automatically generated, and the user generated code developed in Word Address

No	Generated Codes	Developed Codes	Comments
1	M06T1G45H1	M06T1G45H1	Tool Change for the slot
2	G90G00X60Y45Z32	G90G00X60Y45Z32	Move the top of the slot

3	M03S900	S800	Set Spindle speed
4	M07	M07	Coolant On
5	G91G01Z-17	G91G01Z-17	Drilling
6	G94F400	F360	Set Feed rate
7	G91G01Y5 X10 Y-10 X-20 Y10 X10 Y5 X20 Y-20 X-40 Y20 X10	G91G17G47Y15 X20 G48Y-30 X-40 Y30 G46X20 G47Y15 X30 G48Y-40 X-60 Y40 G46X30	Making slot
8	G90G01Z17	G90G01Z17	Tool back off
9	M09	M09	Coolant Off
10	M05	M05	Spindle Off