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

A FLEXIBLE MANUFACTURING CELL PROGRAMMING, SIMULATION, CONTROL AND MANAGEMENT ENVIRONMENT

Thesis submitted

to the School of Graduate Studies

in partial fulfillment of the requirements for the

degree of Master of Applied Science in

Mechanical Engineering

by

Rudy Tolkamp

Ottawa-Carleton Institute for

Mechanical and Aeronautical Engineering

University of Ottawa

Ottawa, Ontario

Canada, K1N 6N5

ISBN 0-612-04887-X

Canada

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

ABSTRACT

The concept of Flexible Manufacturing Cells (FMCs) proves to be one of the major productivity enhancement tools for batch processing. The cost and expertise required to setup a cell, and subsequently to operate it, negates the many advantages of implementing FMCs. This problem hampers the introduction of the concept in many industries. At present, combining machines from different manufacturers to form an FMC is cumbersome. Incompatibility due to unique characteristics and varying programming and control languages among machines present the largest difficulties.

This thesis demonstrates that it is possible to create a manufacturing instruction set and environment labelled UniSet (the acronym for Unified instruction Set) that enables programming and controlling a flexible manufacturing cell with a set of common instructions. In contrast to the efforts of national and international standards organizations, where the aim is to create a standard programming language of protocols, the work undertaken here aims at developing a non-exclusive instruction set that runs on a host computer together with a translation system to the different machine languages.

UniSet is a software environment designed to provide the FMC user with a consistent platform in which to configure, program, simulate and control an FMC irrespective of the constituent machines. The environment is coded in an Object Oriented

Programming (OOP) language; Smalltalk. This thesis describes UniSet as an instruction set, the concept behind the UniSet programming environment and the initial developments of the environment. Also demonstrated are the benefits of object-oriented programming and design to the development of manufacturing software.

ACKNOWLEDGEMENTS

I wish to express my gratitude to my supervisor, Dr. Fahim, for guidance and assistance in completing this degree and in my professional development.

I wish to express my appreciation to the Natural Science and Engineering Research Council for financial assistance that allowed me to pursue these studies.

The ideas that resulted in the completion of this thesis are not entirely my own, the initial idea arose from Dr. Fahim. Many of the developments were a result of collaboration with Dr. Fahim and Hyun Choi.

To my fellow research associates and friends, Gilles Doucet, Samsir Tanary and Kyung Hyun Choi, "Thank you". Discussions, whether technical or less serious, were educational, enlightening and always enjoyable. To Hyun, with whom I shared an office and cultures, I wish success in continuing this research.

This research would not have been possible without the support of family, those who adopted us here in Ottawa and those who supported and encouraged us over the distances. Most of all the support of Stacey, my wife, is greatly appreciated. Special thanks for never doubting and always providing encouragement and support.

These acknowledgements would not be complete without thanks to God, the creator and sustainer of life. May You find me faithful with what You have given.

TABLE OF CONTENTS

CHAPTER 2

CHAPTER 3

Chapter 5

CHAPTER 6

CHAPTER 7

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

THESIS PERSPECTIVE

This chapter outlines the evolution of the FMC, its advantages and limitations. Based on this information the problem definition is detailed and followed by a general description of the material contained in the thesis.

1.1 Flexible Manufacturing

The introduction of computers in manufacturing aroused the hopes for unattended manufacturing. Twenty-four hour unattended manufacturing is still a dream but unattended shifts are becoming a reality [1,2,3,4,5]. Manufacturing automation employing computers first became a reality in 1952 with the development of Numerical Control (NC) machines. Computer Numerical Control (CNC) machines, based on minicomputers followed. CNC differs from NC control because the controller incorporates a digital computer within the machine controller. The computer is responsible for distributing individual NC blocks to the numerical controller rather than the controller receiving blocks from a tape reader or communication link.

The first industrial robot was produced in 1961. Application of robots to manufacturing did not materialize until the late 1970s [6]. The first industrial robots replaced humans in hostile applications on assembly lines such as spray painting and welding. Originally machine tools and robots found applications as stand-alone machines. However, a machine tool running unattended can provide economic benefits [3]. A robot can be configured to service a machine tool while running unnattended. Further development of automated manufacturing theories and techniques should provide substatial economic benefits as computer controlled machines such as machine tools and robots work together in a more integrated environment.

The most recent development in automation has been Flexible Manufacturing Systems (FMSs). They are dedicated systems to be used for mass production and involve the complete manufacturing enterprise from the head office to the shop floor. All steps are coordinated by computers, from filling orders to distributing tasks for the various components. In the ideal FMS paper exchange would be eliminated and all information would be automatically exchanged between computers and machine controllers. The shop floor could be composed of Flexible Manufacturing Cells (FMCs) interconnected by material transportation devices such as Automated Guided Vehicles (AGVs) or conveyors (Figure 1.1). Current implementations of FMSs have not reached this dream, but do involve a large degree of automation. In 1980 it was estimated that 125 Flexible Manufacturing Systems were in operation to some degree world wide [7].

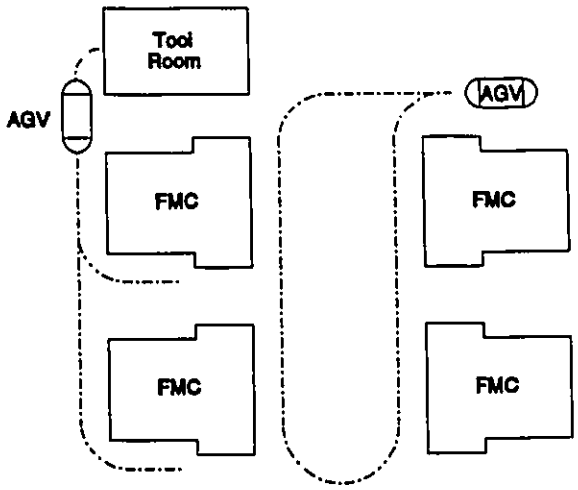

Figure 1.1 Sample FMS

In contrast to the large scale of FMSs is the manufactruing cell for batch production. Batch production is the production of parts in groups ranging in size from a low of approximately one hundred to numbers in the thousands. The idea behind batch manufacturing is to fulfill orders with a group of machines and then be able to use the same equipment, after reconfiguration, to produce different though related parts.

1.2 Flexible Manufacturing Cells

The concept of the flexible manufacturing cell was first introduced by the Production Engineering Laboratory at the University of Trondheim in Norway [6]. The flexible manufacturing cell is generally recognized as best suited for small to medium batch manufacturing.

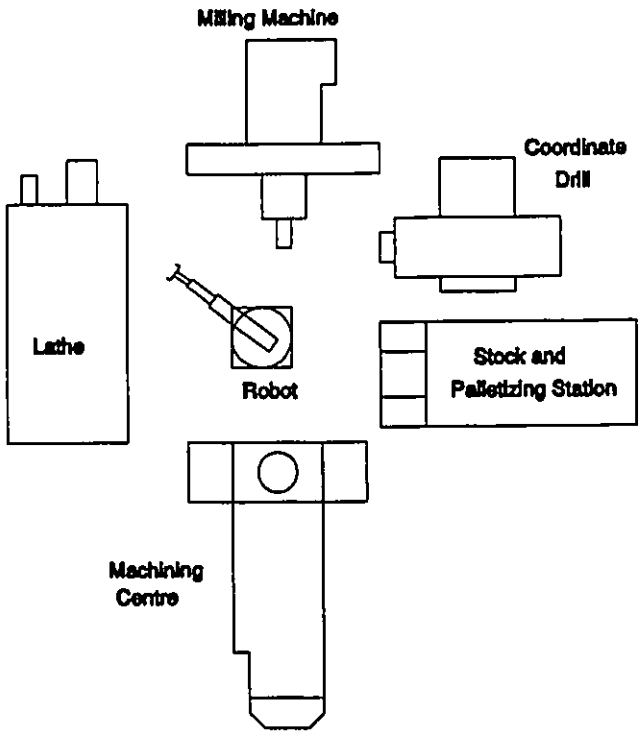

Figure 1.2 Sample Manufacturing Cell

The concepts of FMCs are suited for both machining operations and for assembly. In general both machining and assembly cells may incorporate robots. In assembly cells the robot performs the assembly operations. In machining cells the robot replaces the human operator's responsibilities of loading raw materials, unloading machine parts, changing tools or welding. A block diagram of a machining cell is shown in Figure 1.2. FMCs can be used as a building block for creating the FMS factory.

Advantages to manufacturing due to FMCs have been shown to be multiple. Grouping machines into a cell allows machines to run unattended. Machinists can then use their expertise for operations which humans are considered more adept, including those that require judgement, common sense and intuition [1]. Unattended machining realizes a

reduction in costs associated with labour plus an increase in the number of productive hours per day. Additional economic advantages can be gained if a cell can manufacture more than one part. A change in parts may require a change in programs, tooling and fixtures. If these changes can be done efficiently, without considerable downtime, the cell has additional flexibility.

1.3 Problem Definition and the UniSet Environment Solution

Implementation and programming of an FMC is a tedious task. Cell controller communication and connections can be tailored for an individual cell. Any change in the cell constituents would require a substantial engineering effort. Alternately cells may be constructed using machines from only one manufacturer, the drawback of this alternative is due to the fact that no single manufacturer can supply all the machines required for a cell. Furthermore, constructing an FMC using machines from only one manufacturer may entail using outdated equipment and impede the advantages of using the best technology offered by each manufacturer.

Ideally, programming of an FMC should take place on one computer in one language with one software environment. Total flexibility would be supported by the programming environment and features could be selected as needed. The programming environment would remain consistent independent of the machines in a cell.

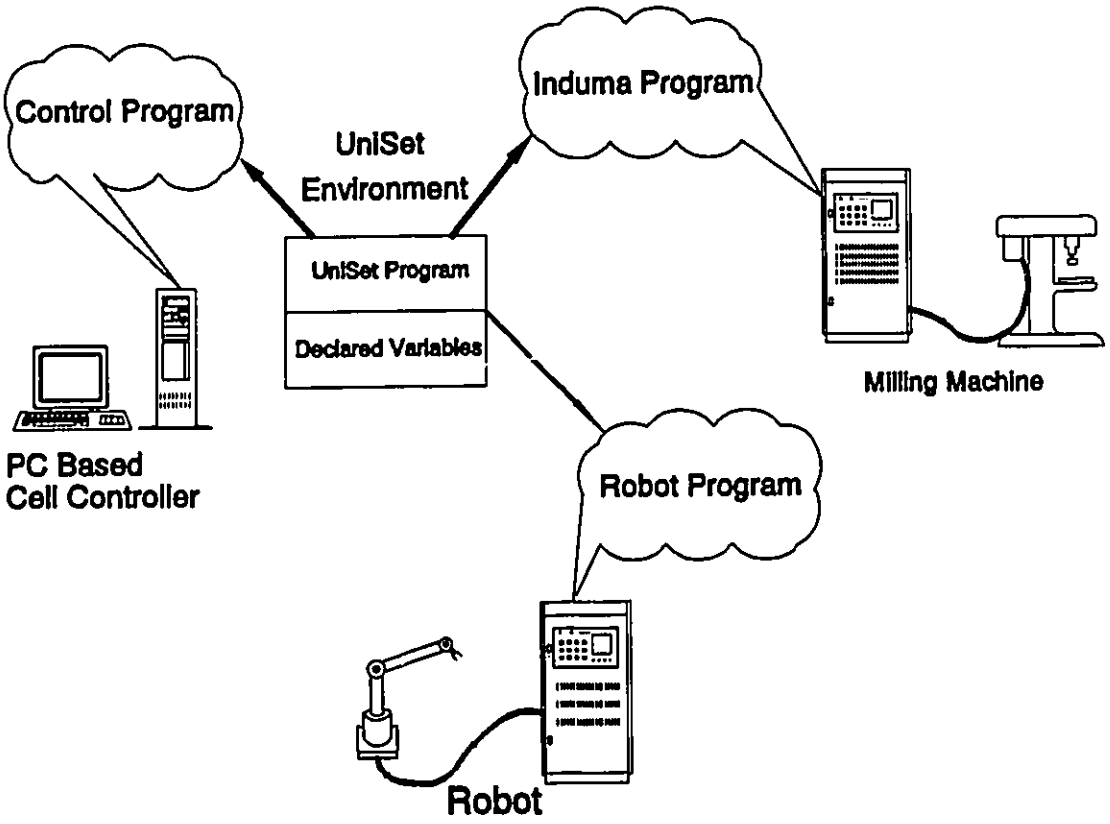

Figure 1.3 Cell Pictorial

A simplified diagram of the operation of the programming environment that fits the above definition and the FMC is shown in Figure 1.3 [8,9]. Programming of the cell takes place within the UniSet software on the PC based Cell Controller. One linear program in the UniSet language is created for the whole cell. After creation of the cell program each part is tranlated into one of the machine languages. The translated programs are downloaded to each machine and the PC then functions as a Cell Controller coordinating all the activities between the machines in the cell.

1.4 Thesis Overview

This thesis describes the current state of Flexible Manufacturing Cell technology and the problems that remain unsolved. Then an instruction set and programming environment are presented as a solution for the presented problems. Chapter 3 is devoted to the selection of design methodology and its impact on choice of implementation language. Chapter 4 defines the primitives for UniSet, the instruction set, that are based on the results of a detailed survey and comparison of VAL II, ASEA, APT and Word Address programming languages. The translation of the primitives from UniSet to machine native code is described in chapter 5. Chapter 6 outlines the design of the classes for the UniSet implementation and translation. The graphical user interfaces of the UniSet environment are detailed in chapter 7.

Appendix A has been included to present an introduction to the basic concepts of object-oriented programming and to serve as a reference while reading the thesis. Appendix B describes classes that were created in order to provide expanded functionality for the user interface. A formal defininition of UniSet syntax is shown in appendix C.

CHAPTER 2

CURRENT STATE

2.1 Two Working FMCs

Two working cells documented in literature that have incorporated multi-vendor equipment are presented here. The first cell has been developed at the National Research Council (NRC) of Canada's Manufacturing Technology Centre [1]. It produces precision parts with sculptured surfaces such as turbine blades. The second cell was created at the Technical Research Centre of Finland for assembly operations [4].

Both cells use multi-vendor equipment and contain one robot. Both authors have commented on the lack of uniformity in communication among machines. The cell in Finland solved communication problems by creating flexible and configurable communications software. The NRC group standardized communication by employing the RS-232 standard for all machines.

In both cases, cell control is implemented on a central controller with custom software. Programs are produced using each machine's native language. Programming the cell in Finland involves four different languages. A conveyor is programmed in McBasic (BASIC for Machine control), a Unimation robot is programmed in VAL, a

McBasic (BASIC for Machine control), a Unimation robot is programmed in VAL, a SEIKO-2000 assembly robot is programmed in DARL (a BASIC-like robot programming language), and the Electrolux commercial storage is controlled by an Autolog programmable logic controller.

The NRC's cell includes a machining centre, a lathe, a robot and a tool rack. The machining centre and lathe execute programs that have been post processed from CAD generated tool paths and downloaded. The ASEA robot is programmed using the teach in method with the ASEA programming language.

Both research centres have been successful in creating a cell that can work unsupervised for extended periods of time. The problems demonstrated in implementation of both cells are typical of the programming and hardware difficulties of creating cells based on multi-vendor equipment.

2.2 Existing Automation Languages and Environments

Over the last few years there have been a number of research efforts in search of solutions to many of the problems associated with developing automated manufacturing facilities. The following is a description of the most notable and promising research efforts that have created general manufacturing languages and programming environments.

2.2.1 AML and AUTOPASS

Research work at IBM has produced some significant results [10]. Since 1972 IBM's research division has been at work in the field of robotics and machine vision. Initial work resulted in an experimental programming language AUTOPASS in 1977. AUTOPASS was one of the first English-like robot programming languages. AUTOPASS' main fault is that it contains the incorrect assumption that there are no inaccuracies in the geometric models of robots, workstations or parts [11].

Concurrently with the development of AUTOPASS, IBM also produced AML. The acronym AML initially stood for 'A Manipulator Language' but has since been updated to 'A Manufacturing Language.' The change in name reflects expanded goals of the research group. These goals extend beyond manipulator programming to include other common manufacturing programming tasks such as NC machines and manufacturing database management.

AML as a robot programming language has achieved some very positive results, however, to date no results have shown it to be any more than a robot programming language. The possibility of expanding it to NC programming, has been speculated [12,13].

AML has been used on IBM robots in IBM's industrial line since 1978 [13] and has been successfully used to program a non-IBM, ASEA, robot. The ASEA robot controller was not altered during implementation of AML. Rather, an IBM PC was patched into the

controller. The language, path generation and inverse kinematics were reimplemented on the PC [14].

Since its first introduction AML has been recoded using the 'C' programming language. 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. AML runs under DOS, UNIX and VM/CMS operating systems [12].

AML encourages user extensions. AML provides a powerful base language which the user can extend with transparent subroutines written in either AML or 'C'. The AML environment also includes motion checking and simulation for robot movements.

AML has been partially successful as a robot programming language and has been shown to include the ability to cater for sensory information. Its development into a complete manufacturing language remains to be seen.

2.2.2 CML

In 1981 Carnegie Mellon University and Westinghouse Electric Corporation began cooperation in order to automate a swaging cell for the production of turbine blade preforms [15]. The research efforts resulted in an automated experimental production cell. The language developed for the programming and control of the cell has been given the acronym CML for Cell Management Language [16]. Initially CML was used to operate

a machining cell [17]. The development of CML and the swaging cell is well documented in papers regularly published by the research group [15,16,17,18,19,20,21].

CML is based on non-decomposable primitives that can be combined into complex programs. Programming in CML is rule-based, thus programs are not sequential but are goal oriented allowing real-time decision making. Rule-based program execution can be slow as the decisions are made and rules are fired.

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.

The FMC for swaging was ready for experimentation in late summer 1982 and ready for production in early 1983. By late 1984 it was running under the control of CML. Wright [17] mentions that complete cell autonomy has not been achieved because of the extreme dangers possible if the cell malfunctions. The cell includes a very high temperature furnace and high capacity hammers. Success has been demonstrated through reduced setup time, increased machine utilization and reduction in the number of rejected parts. Westinghouse has plans to commercialize CML and its environment [19].

2.2.3 APT Based Languages

APT is the original automation language. APT was developed by the Massachuset Institute of Technology. APT III was released in December 1961 [22] for use with stand-alone NC machines. APT is a recognizd ANSI standard. Maintenance of the language is performed by the IIT Research Institute. Languages that are extensions of APT have been developed by research groups for other automated applications.

MCL and RAPT are the result of two such research efforts. MCL was developed by the McDonnell Douglas Corporation under contract with the United States Air Force ICAM program [23]. RAPT was initiated as an experiment in off-line robot programming [24,25]. Both RAPT and MCL use geometric description for programming, making them object level programming languages. Post processors must be built for every machine to generate low level instructions. Both languages include extensions for real-time decision making. MCL includes statements for controlling manufacturing machines other than robots and NC machines.

2.2.4 Comparison

The work of the many research groups developing automation languages has many common features. With the exception of CML, all the groups reimplemented much of the basic functionality of the individual machine controllers. All are developing new languages. No group claims to have solved the difficulties of automation, although each effort has yielded positive results.

LANG-UAGE	DEVELOPER	INTENDED USE	MAJOR FLAW	SUCCESS
AML	IBM	ROBOTS & VISION	MAJOR REWORK FOR NEW MACH.	PORTABLE
AUTOPASS	IBM	VISION	NO ALLOWANCE FOR ERRORS	VERY LIMITED
CML	CARNEGIE MELON UNIV	CELL CONTROL	RULE BASED	REDUCED SETUP TIME, INCR. PRODUCTION. MULTI-VEND OR EQUIP.
APT	MIT	NC MACHINES	LIMITED TO NC MACHINES	MOST POPULAR NC LANGUAGE
MCL	McDONNEL DOUGLAS CORP	ALL MAN. MACHINES	PROGRAMMING BY GEOMETRIC DESCRIPTION	LIMITED
RAPT		ROBOT PROGRAMMING	PROGRAMMING BY GEOMETRIC DESCRIPTION	LIMITED

The current work has provided directions for the future. Most groups realize the need for an extensive programming environment. Stress is placed on the modularity of the tools for automation. Any environment and language should support user modifications and extensions and the software should be portable between computer hardware and automated machines.

As early as 1983 the members of the Advanced Numerical Control group realized that a common interface for every machine controller would be an asset [26]. More recently the Standards Council of Canada has been attempting to address the problem of

automation and programming languages. They have dedicated a technical committee 'ISO/TC 184/SC3/WG2' to the task. The results of their preliminary discussions demonstrate the need for a programming language environment that provides extensive support [27].

2.3 Machine Programming Languages

Since the creation of the first NC machine, work has been continuing on new and improved automation languages. This section focuses on languages for programming individual machines. Recently the most extensive work has centred on robot languages.

2.3.1 Robot Programming Languages

Numerous articles on robot languages have been published exemplifying the strengths, weaknesses, and new features incorporated in each language [28,29,30,31,32]. Bonner & Shin [28] published a thorough article describing robot languages. Among their conclusions, Bonner & Shin suggest that a robot language should include structured programming techniques implemented in an English-like syntax. Variable name lengths should be unrestricted and transforms should be broken into their two basic components of translation and rotation to simplify use and user education. They also mention the need to include commands that could apply to any robot or peripheral device. The application

of one instruction to a number of different receivers is known as polymorphism and will be demonstrated more fully throughout the thesis.

A more recent article by Gruver et. al. [29] also concludes that variable names should be less restrictive. The article iterates that the language should provide room for expansion, new applications and equipment should be easily integrated in the language. Gruver et al. also mention the need for a natural interface that supports novices and experts. Simple tasks should be programmed simply, more difficult tasks should be supported with more advanced constructs.

The robot languages studied for the purposes of this thesis are VAL II and the ASEA robot programming language. VAL II and its predecessor VAL are products of Unimation. VAL II is a textual programming language that includes some high level commands for communicating with other machines. The first robot employing VAL as the programming language was a PUMA. It was delivered to General Motors in mid 1978. VAL II development started in 1981 as a result of efforts to provide communications between robots and other devices and to provide greater flexibility in path control. VAL II was released in 1982. It provides communication abilities and mathematical capabilities common to high-level computer programming languages. Val II also allows the integration of sensory information into the robot control [33,34].

The ASEA programming language was developed for ASEA robots. It is also a textual programming language although most programming takes place at the teach-in pendant. Programs are sequential, each line preceded by numerically ordered line numbers

much like the BASIC language for computer programming. Its functionality is intermediary between that of VAL and VAL II. The language allows for limited sensory input but does not have high-level computational abilities. A robot using the ASEA language is best suited for teach-in programming [35].

For a further description of VAL II and ASEA commands refer to section 4.1.1.

2.3.2 Machine Tool Programming Languages

Machine tool programming languages are far less numerous than robot programming languages. Machine tool programming languages have been in existence for close to forty years [22] and have faded from the mainstream of automation research. APT is the dominant machine tool language employed by industry. Many languages such as ADAPT, EXAPT and RAPT are extensions of APT. Most of the information on APT can be found in book form [22,36].

APT is recognized as an international standard. APT instructions are a combination of major words followed by minor words and applicable arguments. Major words define the function to be performed such as *GO* and *SPEED* for movement and tool speed respectively. Major words do not completely define a machine tool action, they need

further specification. Minor words provide the specifics for the accompanying major word. An example of an APT instruction is:

GO, LEFT/3.

The major word in the preceding APT statement is 'GO', the minor word is 'LEFT' and the '/' slash precedes the argument which indicates how far to go left. Left is in reference to the current most recent direction of machine travel.

The APT Long Range program is charged with the maintenance and development of APT [22]. APT provides many constructs for geometric description but is lacking in control structures and communication capabilities that are present in more recently developed languages.

Word Address is a lower level machine tool programming language. Word Address programming originated with NC control of machine tools by tape reader. The codes are very cryptic and have minimal ties to English. A sample Word Address instruction is:

G91;G01X3Y0Z0

The preceding instruction is a 3 unit move in the x direction, the 'G91' code indicates that any following instructions are to be interpreted as relative movements. The 'G01' code is a movement command, 'X1Y0Z0' is the three dimensional displacement for the movement.

Word Address includes codes for the very basic machine tool functions. It includes minimal control structures and communication abilities whose implementation depend on the machine controller.

2.4 Implementation Requirements & Difficulties

The goal of any FMC implementation is to create a group of machines that work as a unit to perform a single task. The implementation of a cell is a complex and technically challenging endeavour. The minimum hardware includes machine controllers, sensors and two way communication links. The minimum software functionality includes programming capabilities for each machine and sensor, communication drivers and state information. All components should be 'plug compatible.'

Plug compatibility refers to the independence of components. A component is plug compatible if it is encapsulated such that its implementation is independent of all other components. To use a plug compatible component one need only understand the component's interface, not any of its internal details.

Communication involves the transfer of programs, machine states and sensory information. Machine state and sensory communication allows machines to be coordinated without the need to sequence them based on time delays, as well as allows the cell to respond to errors. The development of working communication paths within the cell is non-trivial. Controller communication is non-uniform. Different manufacturers and machine models include varying functionality within the individual machine controllers. Although most machine controllers provide at least RS-232 communication the type of information and level of control available over the communication port varies.

The cell implementor must decide on a control hierarchy. Traditionally one controller is in a supervisory role and coordinates the tasks between machines and ensures proper sequence of execution. Some cells include a separate controller while others rely on one of the machine controllers, typically the robot's, to provide cell control.

Cell controller choice affects and is affected by the choice of cell control software. The software must provide efficient means for programming each machine within the cell and provide for communication between the controller and constituent machines. The cell control software is as important a component of the cell as the hardware. Larin [37] states that "*a new generation of controllers at the cell and machine levels is the key to plant-floor automation.*"

Sensors are vital to the functioning of the manufacturing cell. It is sometimes assumed that in perfectly developed manufacturing cells ample sensors and sensory information and error recovery mechanisms are not required because nothing unexpected will occur. This may be true in a theoretical environment but the manufacturing community realizes that in reality this is not a viable assumption. Rather, there is much research centring on sensors and error recovery in cells. Cell developers must plan for the inclusion of sensors, communication and control. Also of major concern to the cell developer is the distribution of sensory information. If the same sensory information is needed by more than one machine a decision must be made as to which machine has primary access to the sensory information.

Inconsistencies among the individual machines particularly in regards to languages are major obstacles to the implementation of a manufacturing cell. Unique languages arise from machines that are produced by different manufacturers, that are developed at different times and that perform different functions. This forces the cell developer to solve the problem of different languages by either learning all the languages, developing a cell controller to handle all the varying languages or re-implementing a new language on all the machines.

Commercial cell control suppliers have neglected to provide features needed to include machines from other suppliers because it is in their own best interest to sell complete FMCs or systems [21].

Larin [37] presents a good review of the current status of cell controllers. He concludes that the largest gains made toward automated manufacturing in the next decade will come from cell control software and programming languages. Larin shows that the main difficulties in cell operation arise from the inflexibility of the existing systems. Nearly every application requires customized hardware and software that does not accommodate change. Bourne [21] laments about the same problems in that a new part style may require re-configuration of the equipment by the supplier. Larin and Bourne both comment on the inconsistencies among machines in implementing language and communication protocol.

Bourne regards Artificial Intelligence techniques as the next major breakthrough in cell programming. Larin sees the next major advances to come from improvements in the

flexibility and programmability of the controller. Future languages must be flexible and extendible. Flexible automation systems of the future must be retrofittable to existing machinery.

Off-line programming for a cell would be a major enhancement. Off-line programming for NC machines is relatively mature and in wide spread use among industry. Automatic NC programming through CAD/CAM systems such as AUTOCAD, Pro-Engineer and Anvil 5000 is a common feature. Off-line programming of FMCs is still mainly in the research stage because of robot programming difficulties. Many research groups are currently tackling the difficulties of off-line programming [38,39,40,41].

The main problems associated with implementing a manufacturing cell arise from incompatibility. Machine incompatibility particularly in regards to language and communication protocol have proven to be major obstacles. The incompatibility of cell controllers with machines is another difficulty.

CHAPTER 3

IMPLEMENTATION LANGUAGE SELECTION

UniSet and the UniSet environment have been implemented in the object-oriented programming language Smalltalk. This chapter starts with an analysis of why an object-oriented programming language is very beneficial to UniSet's development. Having shown that object-oriented programming (OOP) is the most suitable means for constructing UniSet, justification for selecting Smalltalk as the implementation language is presented.

3.1 Selection of Object-Oriented Programming

Object-oriented languages reduce development time and maintenance costs when compared to sequential programming languages. This new breed of programming languages simplifies creation of new systems and new versions of old systems [42]. For a brief introduction to the terminology and concepts of OOP languges the reader is referred to appendix A.

3.1.1 Standard Protocols

Inherent to true object-oriented programming languages are the features of polymorphism, inheritance, abstraction and data encapsulation. These features encourage the development of common protocols among classes. Protocol refers to the set of messages to which an object will respond. The mechanism of polymorphism allows several classes to use a simgle implemented protocol. If several classes use the same protocol then those classes and their instances are said to be 'plug-compatible' or 'pluggable' [42,43]. 'Pluggable' objects can be used in the same applications without any modifications. Complex objects can be constructed by combining compatible objects. This leads to a style of programming called 'building tool kits' [42].

The standard protocol of 'pluggable' classes provides a common vocabulary for communication between programmers. Smalltalk/V 286 includes standard protocol for its collection and window classes [44]. This protocol is widely used, all experienced Smalltalk programmers know and use it.

The mechanism of inheritance provides a natural way for standard protocols to develop. Since all subclasses inherit the methods of their common superclasses a standard protocol can be automatically inherited. Development of standard protocols is also aided by the use of abstract classes. An abstract class should not be instantiated, it is created "with the expectation that its sublcasses will add to and complete its structure and behavior, usually by completing the implementation of its typically incomplete methods [45]." A good example of a group of classes that are related under an abstract class are the

Smalltalk collection classes. "*Smalltalk collection classes define several different data structures which serve as containers for arbitrary objects ... The collection classes are useful because they provide similar protocol for:*

1. *Iterating over the elements of a collection.*
2. *Searching a collection for a particular element.*
3. *Adding and removing elements.*
4. *Accessing and changing elements.* [44]"

Subclasses of the abstract **Collection** class include **String** which serves as a container for a sequence of characters, **Set** which contains an unordered collection of nonduplicated objects and **SortedCollection** which contains objects sorted in a specified order. The methods *select:*, *collect:* and *detect:* are implemented in the abstract **Collection** class definition. They are all written in terms of a method *do:* which must be implemented in the subclasses. The implementation of the method *do:* varies depending on the class because objects in a collection may be indexed or stored in key/value pairs. The careful design of the collection heirarchy has ensured a common protocol, that includes the methods *select:*, *collect:* and *detect:*, for all collection subclasses.

UniSet employs a similar class structure with the **WorkStation** class and its subclasses. Subclasses of **WorkStation** include **Robot**, **Lathe**, and **Mill**. Class **WorkStation** is an abstract class that implements methods common to its subclasses. For instance *send:* and *receive:* are methods implemented in the **WorkStation** class but must

be further defined in the subclasses because each machine implements communication individually.

Abstraction and encapsulation also promote the development of standard protocols. Abstraction deals with the external view of an object. A good abstraction relates only the details of the object that are necessary to use of the object. Encapsulation deals with information hiding and can be thought of as the opposite side of abstraction. An object that is a good abstraction must also be well encapsulated. To present only the necessary details of an object also entials that all the other details are well hidden. The implementation of objects that are both good abstractions and encapsulations will be a major benefit to the development of UniSet and progession of FMC development. For example every machine must have abilities to communicate with UniSet but in a well designed object the details of communication would be hidden aspects of the object. Communication should be provided by singular abstract commands such as *send* and *receive*. A poorly constructed object would require the programmer to always know the internal details of communicating such as baud rate, parity and signal lines. Well planned abstractions would allow machines that are implemented differently to share the same protocol for communication.

Standard protocol is important to the development of UniSet. It provides an ideal way to program machines within the manufacturing environment. A standard programming protocol for all machines solves many problems because the programmer is relieved of knowing each machine's atypical way of implementing the common functions. Learning

how to program new machines is accelerated if new machines abide by the standard protocol. Addition of new machines to the UniSet environment is also eased. If a class is implemented with the standard protocol, the class is plug compatible with other classes that implement the same protocol.

3.1.2 FrameWorks

Frameworks as described by Johnson and Foote [42] can prove to be a valuable tool for developing manufacturing programming environments. A framework is an object-oriented abstract design that provides a means of reusing software fragments. An application programmer needs to create classes that adhere to the interfaces defined by the framework.

Johnson [42] shows that the Smalltalk **Collection** hierarchy is a good example of a framework. These classes provide general means for manipulating objects in collections such as **Sets, Bags, Arrays,** and **SortedCollections.** Each class is thought of as an application independent solution to a large range of common problems.

It should be possible to describe a framework suitable for manufacturing programming. This framework would provide a consistent solution to many of the common problems facing programmers of manufacturing equipment such as consistent communication between machines, a consistent programming interface and even a consistent means for specifying data such as a three dimensional location. The Canadian

Standards Council [46,47,27] can is working towards the development of a graphical user interface framework for manufacturing.

3.1.3 Maintenance and Programming-by-Difference

The mechanism of inheritance provides many unique advantages. One of these advantages is the ability to program-by-specialization or difference. A new class can be developed by choosing a closely related existing class and adding a subclass. The new class inherits a basic structure and its development involves programming the differences between the new class and its superclass. Programming-by-difference within the context of manufacturing allows new machines to have their classes defined with little effort. The addition of a machine tool with an automatic tool changer may mean only adding a subclass to class **Machine_Tool** and supporting the concept of automatic tool changing.

Inheritance also eases maintenance and software extension. A subclass is an automatic copy of its superclass, thus modifications and enhancements can be made to a class while leaving the original intact. The nature of manufacturing makes this a very attractive feature. Small batch sizes and the ever increasing push to increase productivity means software should not be a stagnant component of a manufacturing enterprise but should develop with new applications.

3.1.4 Object Editors

OOP has accelerated the development of Object Editors [48]. Object editors provide a graphical means of editing instances. Graphical user interfaces (GUIs) provide a flexible way to directly manipulate objects. Highly complex objects can be edited through such interfaces without revealing the intricacies of the object's implementation. The development of GUIs is enhanced through the use of OOP [49]. It is possible to develop general user interfaces that allow editing of instances of various classes. Such object editors can provide a convenient method of configuring manufacturing software. WorkStation, sensor and communication instances can easily be conceived as being configured through object editors.

3.1.5 Examples

Some manufacturing software research has used object-oriented techniques. Modular programming has been common for many years, well designed modules can often appear object-oriented. Johnston [1] describes the software for the NRC's cell as being very modular. Their communicator processes provide high-level functional interfaces. All the communicators look the same to the I/O processes and function in the same manner. Fabian and Lennartson [50] have demonstrated the object-oriented approach for control of manufacturing systems and have developed common messages for communication for all manufacturing devices.

3.2 Smalltalk - The Language of Choice

Smalltalk was chosen as the implementation language for UniSet. Smalltalk is the purest OOP language currently available [51,52]. It has many features that make it well suited to the development of the Unified Instruction Set and the UniSet environment.

3.2.1 Smalltalk Features

The problem definition for this thesis stresses the need for a supportive programming environment for cell programming. The Smalltalk environment provides such support for OOP. Lalonde and Pugh [53] stress that OOD requires an interactive design tool such as the Smalltalk environment.

Smalltalk programming is far from traditional. The Smalltalk environment is a starting point for a project. As programming progresses the Smalltalk environment is transformed into the final product. Smalltalk objects and user modifications develop a history allowing the programmer to build on events of previous programming sessions. Objects defined within Smalltalk persist between Smalltalk sessions. A Smalltalk application is the result of software evolution. The Smalltalk environment evolves into the final program.

The mechanism of polymorphism can be powerful especially within the context of developing UniSet. Smalltalk provides full featured polymorphism because it is an interpretive language that uses late binding for procedure calls. Late binding means that

the types of all variables and expressions are not known until runtime. C++ does not provide full featured polymorphism. A C++ object must have the correct superclass to receive a message, not just the right protocol [42].

Smalltalk has limitations. It is not particularly well suited for mathematically intensive operations and high speed communications. The procedural programming language 'C' is well suited to such applications. Smalltalk can be linked to primitives and programs written in 'C' [54]. Thus the power of 'C' can be combined with Smalltalk's reliability and the advantages of OOP.

The hardware platform that UniSet is implemented on is very important. The use of IBM PCs for control applications is popular [37]. Smalltalk and thus UniSet is implemented on a personal computer requiring a minimum of an 80286 processor. Smalltalk/V 286 operates under DOS and OS/2. It is file compatible with Smalltalk/V for IBM xt compatibles and Smalltalk/V MAC. Thus, Smalltalk and any Smalltalk applications are highly portable.

3.2.2 Smalltalk Productivity Enhancers

In addition to the advantages with being object-oriented Smalltalk has many advantages that are specific to developing the UniSet environment and instruction set. The Smalltalk environment provides many pre-defined classes and existing tools to ease the object-oriented programmer's work. The base Smalltalk/V system comes with over one hundred classes and two thousand pre-defined methods that are completely accessible to

the user [44]. Smalltalk not only provides graphical user interfaces for program development but provides classes that the programmer can use as building blocks for developing application specific GUIs.

The **Collection** class and its many subclasses provide a library for solving common problems. The **Dictionary** class is very important for developing UniSet. A dictionary is similar to an array except that its indices can be any objects. Thus a dictionary allows collections of objects to be paired. Dictionaries have proven to be very beneficial for developing UniSet translations and the UniSet Translator because instructions from machine native languages can be the indices and the UniSet translation the associated value.

Smalltalk also provides complete class support for file access. Class **FileStream** provides a complete set of messages for writing and reading ASCII files. This class simplified the task of providing ways for Smalltalk and UniSet objects to be saved to files such as UniSet languages.

Smalltalk is attractive for prototyping and developing because the language is interpreted and provides extensive tools for debugging code. An interpreted language allows immediate testing of the code without the need for user directed compiling or linking. The tools for debugging include a debugging window that displays a trail of the most recently executed messages with the accompanying method and any relevant objects to the code. The window also provides the ability to perform step-by-step execution. The objects and their variables may be inspected at any time and for further validation of the code.

Smalltalk memory management is not of concern to the programmer. Smalltalk provides its own memory management and purges the system of released objects [44] so that memory does not become an object graveyard. Smalltalk can manage up to 16 megabytes of memory.

Chapter 4

MIP: UNIFIED INSTRUCTION SET

The creation of a Manufacturing Instruction Protocol is the major goal of this thesis. Justification for work to proceed on unifying automation languages is provided by preliminary studies of automation languages performed by Szukalo [55], Taylor [56] and Quail [57]. These studies concluded that automation languages contain many commands that are similar in function. Continuing with their investigation this chapter presents a detailed comparison of four languages: APT, Word Address, VAL II and ASEA. UniSet is constructed following this comparison.

4.1 Survey of Machine Programming Languages

The development of UniSet is based on a core of common instructions that every automated machine requires. Each core instruction is considered a primitive. UniSet provides translations for these primitives.

4.1.1 Machine Tool Programming Languages

The two most common machine tool programming languages are Word Address and APT. Word Address is not a high-level language. It was developed for use with NC machines and is still used by most modern CNC machine tools. Word Address programs are very cryptic and difficult to read and have very little correlation to English, much like Assembly Language for personal computers. Each statement, or block, consists of a character that specifies the address of the word and a number representing the address' new content. A word specifying a tool position of X = 1.0 is 'X1.0' [36].

APT is the acronym for Automatically Programmed Tool. CNC controllers that accept APT programs are rare. For most CNC applications APT must be processed to generate a CLFile (Cutter Location File). The resulting CLFile must be post-processed to produce NC (Word Address) code.

Compared to Word Address, APT is a high level language (Table 4.1). The APT language uses english like syntax while Word Address uses a cryptic combination of letters and numbers. The APT command GOTO is equivalent to the Word Address command G01. The main power of APT is that it allows the programmer to develop geometric descriptions of the part to be machined, and then a tool can be driven about a geometric description without the programmer performing any in depth mathematical calculations. The standardized APT language contains about 600 vocabulary words [36] most of which are used for geometric descriptions. There are many ways to define most geometric

APT	WORD ADDRESS
GOTO/x,y,z	G90;G01XxYyZz
DELAY/t	G04Pt
SPINDL/s,CLW	MO3Ss

Table 4.1 Comparison of APT and Word Address Primitives

entities. For example there are 26 statement formats for defining a point and 27 formats for defining a line.

The APT vocabulary consists of major and minor words. Each APT statement consists of a major word and possibly a minor word that serves to modify or clarify the meaning of the major word. For example, Table 4.5 shows the APT statements for controlling spindle feedrate. The major word FEEDRAT followed by a number specifies the feedrate in inches per minute. If FEEDRAT is followed by the minor word IPR and then a number the statement is interpreted as indicating inches per revolution.

APT programs are processed and reduced to Word Address programs. Many APT commands are high-level commands that are translated to a combination of Word Address words, much like a macro. UniSet instructions will not be provided for high-level APT commands, only primitives. High-level instructions should be developed as UniSet instructions. The set of APT primitives required to operate a machine tool are limited to

those operations provided in Word Address code and their equivalent APT instructions. Every Word Address code has an APT equivalent.

4.1.2 Industrial Robot Programming Languages

Robot programming is the subject of much current research. Recently developed robot languages contain very powerful instructions. Languages that were originally developed for robot programming, such as AML, have been transformed into manufacturing languages with the hopes that they can control the whole plant floor. This thesis will only focus on basic robot instructions that are required to operate a robot as part of a manufacturing cell. UniSet instructions will not be provided for robot functions that are intended for controlling other machines machines since control of individual machines within the UniSet environment should be performed by the UniSet controller and not individual machine controllers.

VAL II and ASEA include commands for robot motion and simple program control. The languages share instructions for low level operations. Higher level operations vary greatly in availability and implementation between robots and their languages. Most of these commands are not available in the ASEA language. Through UniSet high level operations such as interfacing to other equipment and sensors will be implemented in a common format.

NO.	INSTRUCTION	COMMENT
10	V=700 MM/S MAX=900 MM/S	Full and maximum speed.
20	TCP 0	Tool Center Point
30	ROBOT COORD	Use Robot Coordinates
40	FRAME 0	Coordinate axis origin
50	POS V=50% (X=569.6 Y=21.4 Z=630.1)	Absolute position instruction.
60	POS V=50% STORE LOCATION1	Store current position in location register 1.
70	POS V=50% LOCATION1 OFFSET (X=0 Y=0 Z=200) MM	Move up 200 mm.
80	POS V=50% FINE LOCATION1	Return to LOCATION1.

Figure 4.1 Sample ASEA Program

Figure 4.1 shows a sample ASEA program including absolute and relative positioning instructions. The information in lines 10 through 40 must be present as the first four lines of every ASEA main program.

4.1.3 Programming Language Comparison

Development of a manufacturing instruction protocol is based on a comparison of machine tool and robot programming primitives. The primitives include motion (Table 4.2, Table 4.3, Table 4.4), speed control (Table 4.5), gripper and chuck actions (Table 4.6), program control (Table 4.7), signal I/O (Table 4.8), and spindle and coolant instructions (Table 4.9).

Absolute Positioning	
Word Address	G90;G00<loc>
APT	RAPID;GOTO/<loc>
VAL II	MOVE <pos> or ; <float> specifies MOVET <pos>,<float> ; hand opening.
ASEA	ROBOT COORD POS V= <i>% <loc> MM
Relative Positioning	
Word Address	G91;G00<loc>
APT	RAPID;GODLTA/<loc>
VAL II	MOVES HERE:<trans> or MOVEST HERE:<trans>,<float>
ASEA	ROBOT COORD POS V = <i>% <loc> OFFSET <loc> MM

Table 4.2 Joint Interpolated Motion

Motion control includes three separate primitives because of path requirements. Joint interpolated motion is used for rapid movements when the exact path is not of concern. The primitives for joint interpolation are listed in Table 4.2.

Linear interpolated motion is used to drive a machine through a straight path when the exact path of the tool is critical. Linear interpolated motion is used by robots for welding applications or maneuvering between objects. Machine tools use linear interpolated motion for removing metal in a straight line.

Absolute Positioning	
Word Address	G90;G01 <loc>
APT	GOTO <loc>
VAL II	MOVES <pos> or MOVEST <loc>, <float>
ASEA	RECTANGULAR COORD POS V = <i> % <pos> MM
Relative Positioning	
Word Address	G91;G01 <loc>
APT	GODLTA/ <loc>
VAL II	MOVES HERE: <trans> or MOVEST HERE: <trans>, <float>
ASEA	RECTANGULAR COORD POS V = <i> % <pos> OFFSET <loc> MM

Table 4.3 Linear Interpolated Motion

The VAL II commands for joint and linear interpolated motion have two formats. The second format is differentiated from the first by a 'T'. These commands require a second argument of type float that indicates the desired gripper opening after completion of the manipulator movement. Table 4.3 lists linear interpolated motion primitives.

Circular interpolation is also for path critical movements. Circular movement is actually approximated by a series of straight lines either by the post processor or machine controller. The circular motion commands could be replaced by a high-level UniSet command that translates to a series of straight lines but this would be duplicating a

preexisting function that already exists in three of the languages studied. The circular motion commands are listed in Table 4.4. A noticable vacancy in Table 4.4 is the lack of circular motion command in the Val II language. While programming in Val II a circular trajectory can be approximated by locating a series of points in a circle and moving the end effector through the series of points with joint or linear interpolated movements.

The circular interpolation instruction for APT is two instructions. The instruction 'ARCSLP/ON' is an instruction to the APT postprocessor. It indicates to the postprocessor that the APT machine controller can perform circular control and the postprocessor need not approximate the circle with a series of straight lines. The instruction 'TLONPS,GOFWD/<circle> tells the machine to proceed to move along the path defined in the circle argument. This argument may define a circle or a variable that has been previously defined as a circle.

Table 4.5 shows speed control instructions for both robots and machine tools. The robot instructions control the speed at which the end effector travels. Machine tool speed control instructions control the feedrate of the tool. The instruction that defines the speed in terms of 'Feed/min' is for use with milling machines. The 'Feed/rev' instruction controls lathe tool speed by feeding the tool into or along the workpiece the specified distance every revolution of the workpiece.

Speed programming in robot languages differs from machine tools. Machine tool speed can be changed anywhere within a program by resetting the speed. Generally robot

Word Address	
XY plane, cw	G17G02<circle>
ccw	G17G03<circle>
XZ plane, cw	G18G02<circle>
ccw	G18G03<circle>
YZ plane, cw	G19G02<circle>
ccw	G19G03<circle>
APT	ARCSLP/ON TLONPS,GOFWD/<circle>
VAL II	N.A.
ASEA	RECTANGULAR COORD POS V=<i>% <pos> POS V= <i>% CIRCLE <pos> POS V=<i> <pos>

Table 4.4 Circular Interpolation

speeds are set as a percentage of a basic speed. It is also possible to control speed similar to NC controlled machines by changing the basic speed.

Robot and machine tools require commands for closing and opening the gripper, chuck, tool holder or vise. Functionally the actions are the same. Robot grippers and machine tool chucks, tool holders and vises are all used for grasping an object. It is appropriate that the same command be used to implement all these actions. Word Address instructions for such functions are usually machine specific and implemented using the

Word Address -- Tool Speed	
Feed/min	G94F<number>
Feed/rev	G95F<number>
APT -- Tool Speed	
Feed/min	FEEDRAT/<number>
Feed/rev	FEEDRAT/IPR,<number>
Val II -- End Effector Speed	
inches/sec	SPEED <number> IPS ALWAYS
mm/sec	SPEED <number> MMPS ALWAYS
percentage	SPEED <number> ALWAYS
ASEA -- End Effector Speed	
Basic Speed	BASIC VEL. <number> MM/S
Speed and max	V = <number> MM/S MAX = <number> MM/S

Table 4.5 Speed Primitives

extra 'M' (miscellaneous) codes provided in the controller. There is no command in the APT language for controlling the chuck and tool holder. The gripper control for the ASEA robot is a boolean function 'GRIPPER' that alternates the gripper state. VAL II provides instructions for opening and closing the gripper plus proportional control of the gripper. The gripper commands 'close' and 'open' functionally equivalent. Both are proportional gripper commands causing the gripper to be opened to the distance specified. VAL II gripper openings are adjusted during the next motion statement unless the

Word Address[1]	
Clamp 4TH axis	M21
Unclamp 4TH axis	M22
APT	
	N.A.
VAL II	
Close gripper	CLOSE
Open gripper	OPEN
Immed. Gripper Control	OPENI <number> or CLOSEI <number>
ASEA	
Close gripper	GRIPPER
Open gripper	GRIPPER

[1] These extra 'M' codes vary between machines.

Table 4.6 Gripper Commands

commands **'closei'** or **'openi'** are used. Gripper command comparisons are shown in Table 4.6.

Program control instructions are a necessity for programming. Machine tool and robot languages provide varying degrees of program control. Word Address and APT provide very simple control structures. A memory rewind resets the instruction counter back to to the previous memory rewind instruction or to the beginning of the program. This term dates back to the time when programs were stored and run from tapes. ASEA's

Program End: Memory rewind	
Word Address	M30
APT	END
Program Stop or Wait	
Word Address	M00
APT	STOP
VAL II	
High	WAIT SIG(<int>)
Low	WAIT SIG(-<int>)
ASEA	WAIT (UNTIL INPUT <int> = <int>)
Timed Delay	
Word Address	G04X<number>
APT	DELAY/<number>
VAL II	DELAY <number>
ASEA	WAIT (TIME <NUMBER>S)

Table 4.7 Control Structures

control structures are a little more developed. VAL II's 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, at this stage, will only be concerned with the low level primitives. Table 4.7 shows only the most primitive control structures.

Machines must be able to communicate their states to the cell controller. The Word Address language does not provide specific means for communicating information, thus 'M' codes or auxiliary sensors must be employed for this purpose. APT is lacking in this area as well. Both robot languages provide adequate output signals. A comparison is shown in Table 4.8. In VAL II a line is set high by providing the positive number of

Word Address	Extra 'M' codes
APT	N/A
VAL II	
High	SIG(<line number>)
Low	SIG(-<line number>)
ASEA	
High (+24 V)	OUTPUT <line number> SET
Low	OUTPUT <line number> RESET
Invert	OUTPUT <line number> INVERT
0.5 sec pulse	OUTPUT <line number> PULSE

Table 4.8 Signal Output

the signal line as the argument for the SIG<*arg*> command. A line is set low by negating the signal line number.

Control of spindle and coolant are operations specific to the operation of machine tools. Spindle and coolant control are vital machine tool functions, UniSet must include

Spindle Control	
Word Address	
Spindle off	M05
Clockwise	M03S<int>
Counter-Clockwise	M04S<int>
APT	
Spindle off	SPINDL/OFF
Clockwise	SPINDL/<int>,CLW
Counter-Clockwise	SPINDL/<int>,CCLW
Coolant Control	
Word Address	Extra 'M' functions
APT	
Flood	COOLNT/FLOOD
Mist	COOLNT/MIST
Tapping	COOLNT/TAPKUL
On	COOLNT/ON
Off	COOLNT/OFF

Table 4.9 Machine Tool Spindle and Coolant Control

such primitives. Word Address does not provide a consistent way to implement the coolant control. The extra 'M' codes provided with the controller must be used. A comparison of the commands relevant to spindle and coolant operation is shown in Table 4.9.

4.2 Requirements

UniSet should have rules regarding syntax and command structure. Without consistent syntax and design the programmer's job is more difficult. Therefore UniSet must be designed with certain guidelines regarding the format of commands. By following the guidelines, definition of new UniSet instructions is eased.

Even though considerable effort has been applied to ensure that UniSet provides many tools to the FMC programmer it will always be developing and evolving. However, it is not forseen that UniSet would cater inherently to all machine features. There are some machine features that would require major redevelopment to be implemented in UniSet. To encourage flexibility and not limit integration of machine vendor developments UniSet is non-exclusive. The non-exclusive nature refers to UniSet's ability to accept commands in other formats. UniSet allows commands to be entered in the machine's native language. Thus, a particularly beneficial feature of a machine or a programmer's valuable experience in any particular programming language can still be employed within the UniSet environment. Also, separately developed program files can be imported into a UniSet program. This will be most beneficial because machine code generated by other systems can be utilized by UniSet. Code generated by a CAD/CAM system tool path generator, for example, is the most likely to be used in this manner. To provide such a tool within UniSet would require major reimplementation of existing technology.

4.3 Syntax

4.3.1 Alphabet

UniSet uses the same alphabet as Smalltalk. The alphabet is shown in Appendix C.

4.3.2 UniSet Instruction Format

There are three formats for UniSet instructions. They are shown in Table 4.10. 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.

Class	Format	Example
UnaryCommand	*command*	Stop
UnaryKeyWord	*command:* <arg>	Go: home
		Goto: 5,5,1
UnaryModified	*command, modifier*	Mist, off
	command, keyword: <arg>	Gripper, open: 5

Table 4.10 UniSet Syntax

The second instruction format is known as **UnaryKeyWord**. A **UnaryKeyWord** is a single word accompanied by an argument. **UnaryKeyWord** commands must end with a full colon (:). The colon serves as a syntax delimiter. A sample statement using a

UnaryKeyWord is *GOTO: <arg>*. The argument class is verified during command translation. Each **UnaryKeyWord** knows the valid classes of arguments that can accompany the argument.

The third instruction format is **UnaryModified**. **UnaryModified** instructions consist 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 same root instruction can be used to program related functions. Modifiers may be either a **UnaryCommand** or a **UnaryKeyword**. A statement that includes a **UnaryModified** instruction must include a comma immediately following the root of the instruction. The comma serves as a syntax delimiter. An example of a set of similar instructions are spindle speed instructions. A spindle can be rotated clockwise or counterclockwise at a set speed, or stopped. The instructions that would represent these functions are:

SPINDLE, CW: <arg>

SPINDLE, CCW: <arg>

SPINDLE, OFF

4.3.3 UniSet: Manufacturing Instruction Protocol

A partial list of UniSet primitives are presented in Table 4.11 and Table 4.12. All UniSet commands adhere to a consistent format with English-like syntax. This should allow the instructions to be easily understood and easily committed to memory. Most

Joint Interpolated Motion	
Absolute	Goto, rapid: <position|location>
Relative	Go, rapid: <position|location>
Linear Interpolated Motion	
Absolute	Goto: <position|location>
Relative	Go: <position|location>
Circular Interpolation	
Speed Control	
Feed/min	Speed, F/min: <number>
Feed/rev	Speed, F/rev: <number>
Inches/sec	Speed, I/sec: <number>
mm/sec	Speed, mm/sec: <number>
No Units	Speed: <number>
Gripper Instructions	
Open	Open
Close	Close
Proportional	Open: <number>

Table 4.11 UniSet Primitives

UniSet commands are polymorphic, that is they represent a similar function for different machines.

Delay	
Wait	Wait
Wait on Signal	WaitFor, signal: <integer>
Wait on Time	WaitFor, time: <number>
Output Signal	
Set Signal	Signal, set: <integer>
Reset Signal	Signal, reset: <integer>
Spindle Control	
Turn off	Spindle, off
Clockwise	Spindle, cw: <number>
Counterclockwise	Spindle, ccw: <number>
Coolant Control	
Turn off	Coolant, off
Turn on	Coolant, on
Flood	Coolant, flood
Mist	Coolant, mist
Tapping	Coolant, tapping

Table 4.12 UniSet Primitives (cont'd)

4.3.4 Concurrency and Synchronization

The concept of Flexible Manufacturing Cells is based on the idea that individual machines could work together to perform a singular goal. In reaching the goal it is necessary that the individual machines must interact and cooperate. A robot loading a

machine tool would be a common interaction. The action of a robot entering into the working volume of a machine tool has many potential hazards. To control the hazards the cell programmer must have at his disposal tools to start and stop operation of each machine.

Programming the interaction between machines requires the cell programmer to be able to easily shift between programming different machines. Signals are required to allow one machine to wait for another machine to finish execution before proceeding. The UniSet commands provided for machine synchronization are *wait:* <aMachine> and *signal:* <*aMachine*>. The *wait:* command means the receiver must wait to receive a signal from the machine specified before executing the next command sequence. The *signal:* command instructs the machine to set a signal line. A signal command should be issued after a critical region of code is executed. The signal should be sent to machines waiting for the critical region to be completed.

Following is an example of code laden with machine interaction. The code assumes the robot has already picked up a piece of raw stock and the mill's vice is empty. The code for a robot loading a part into a mill would include:

```
...
Mill
        signal: Robot.                  ; signal Mill is stopped
Robot
        moveTo: millAproach.            ; millApproach is a position.
        wait: Mill.                     ; Wait for Mill to stop
        moveTo: millChuck.              ; millChuck is a position.
        signal: Mill.                   ; signal Robot stopped
Mill
        wait: Robot.                    ; wait for Robot to stop
        close                           ; close vice
        signal: Robot.                  ; signal part gripped
Robot
        wait: Mill.                     ; wait for Mill to stop
        open.                           ; release part
        moveTo: millApproach.           ; back away from Mill
        signal: Mill.                   ; signal all clear
        moveTo: home.                   ; go to neutral position
Mill
        wait: Robot.
        ... (machining sequence)        ; continue with machining
```

The program above requires much synchronization to perform the task. The programmer must ensure that all waits and signals are properly matched. Improperly place waits and signals can lead to collisions or deadlock. Deadlock occurs if a machine waited for a signal it might never receive.

4.4 Machine Level

For the concept of synchronization to work successfully each individual machine involved must allow its own execution to be halted and restarted. Machines must also be able to send signals as part of program execution. A UniSet cell is set up as a star network, all

communication must be sent through the UniSet environment. The UniSet environment must be able to send and receive signals to and from each individual machine.

4.4.1 Machine Tool

Word Address and APT provide little flexibility in regards to program execution control. There is no ability to conditionally execute programs based on a signal. The Word Address language provides no conditional statement at all, a looping statement is the only form of execution control. APT provides a conditional statement using the command 'IF (s) JUMPTO/ label'. The condition 's' can only involve a relational operation such as 'GE' or 'EQ' and numerical values.

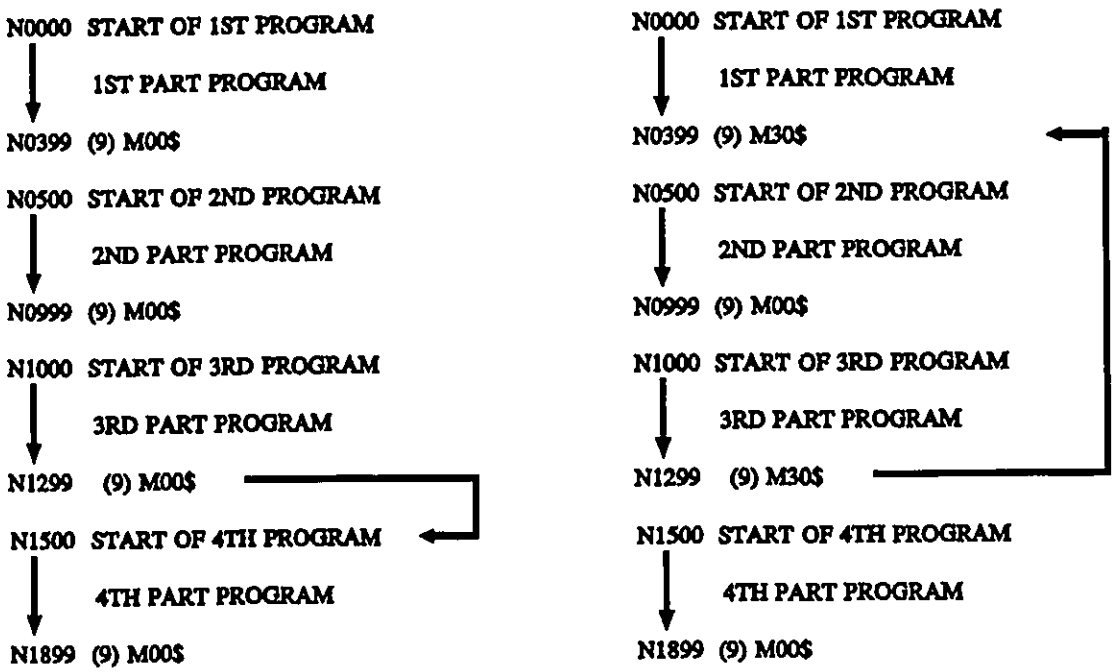

Figure 4.2 M30 and M00 Word Address Instructions

The Word Address language, and also APT, provide two ways for ending a program. M30 halts program execution and also causes a memory rewind. A memory rewind means the program counter will return to the statement immediately following the previous 'M30' statement. The effect being that the memory is reset to the beginning of the program just executed. A cycle start will then execute the same code. In contrast an 'M00' statement at the end of a program will only halt execution, the program counter will still point to the next operation. A cycle start will allow the next sequence of actions to take place.

Figure 4.2 shows the difference between the memory rewind of the M30 instruction and the M00 instruction. After execution of the third program the program counter on the left will halt but point to the start of program four. The M00 instruction causes the execution to halt. In the list on the right, execution will halt at the end of program three, because of the M30

instruction the program counter will be reset to just after the previous M30 instruction at the start of program 2. Note that the memory rewind even skipped the M00 instruction at the end of program 2.

The M00 code will allow the machine tool to halt execution until the machine tool receives a cycle start command. This is equivalent to waiting for a signal.

There is a direct APT equivalent to the M30 and M00 instructions in the Word Address language. The instruction 'END' corresponds to M30 and 'STOP' corresponds to M00. Thus machines that use Word Address or APT can be treated identically.

The machine tools must not only be able to receive a message but they must be able to send a message to the UniSet cell controller. The signal sent from the machine tool to the UniSet cell controller will indicate that the machine tool is at a certain stage in execution.

Most machine tools do not implement a consistent command set for generating communication signals. If the machine tool has the capability for generating the signals the command is a miscellaneous or 'M' code. Generally the less common commands are in the M90 to M99 range. We will assume this range and take a somewhat arbitrary value of M95.

The inflexibility of machine tools and there controllers is a major stumbling block when trying to make them part of a cell. The absence of the wait and signal commands makes machine synchronization more than a simple task.

4.4.2 Robot

Generally robot languages are more flexible and contain higher level commands for control of program execution than machine tool languages. The ASEA and VAL II programming languages provide a wait instruction that allows the argument to be a signal. This instruction provides the exact function necessary for synchronization. The format of the command for the two languages is slightly different but the information and result is the same.

The wait command for ASEA is of the following format.

WAIT (UNTIL INPUT <int> = <int>)

The integer following 'INPUT' specifies the signal and the integer following the equal sign specifies the condition. A '1' in this position means the circuit is closed and a '0' means an open circuit.

VAL II provides a similar command, its format is the following.

WAIT SIG(int)

The absolute value of the integer specified between the brackets selects the signal line to wait on. If the integer is positive the wait continues until the signal line is on. If the integer is negative the wait continues until the signal line is off.

Val II and ASEA also implement instructions for sending signals. The format for generating a signal with ASEA is the following:

OUTPUT (int state)

The integer argument selects the signal line. The state can be one of four choices: SET, RESET, INVERT or PULSE. SET generates a 24 V signal, RESET generates 0 V. INVERT toggles the output value. If the output was SET, INVERT causes the output to RESET. If the output was RESET INVERT will SET the output. PULSE is similar to INVERT except that it does not change the state until the next OUTPUT instruction. PULSE only holds the new signal value for 0.5 seconds.

VAL II only provides two options for generating a signal. The format of the command in VAL II is:

SIGNAL int

Like the WAIT instruction for VAL II the absolute value of the integer in the SIGNAL instruction selects the signal line. If the integer is positive the signal is turned on, if negative the signal is turned off.

The more recent development of robots and the continuing research in robot languages explains why robot languages contain the commands necessary for synchronization. Much robot programming research centres on machine interaction with the outside world.

Chapter 5

TRANSLATION

A cell based on the UniSet software should be linked together in a star network with the central node being a personal computer as shown in Figure 5.1. Cell program segments are distributed by the PC to each machine. The PC initiates and supervises operation of the cell. All cell communication must flow through the PC based cell controller. The PC coordinates rendezvous' between machines ensuring that the proper sequencing is followed.

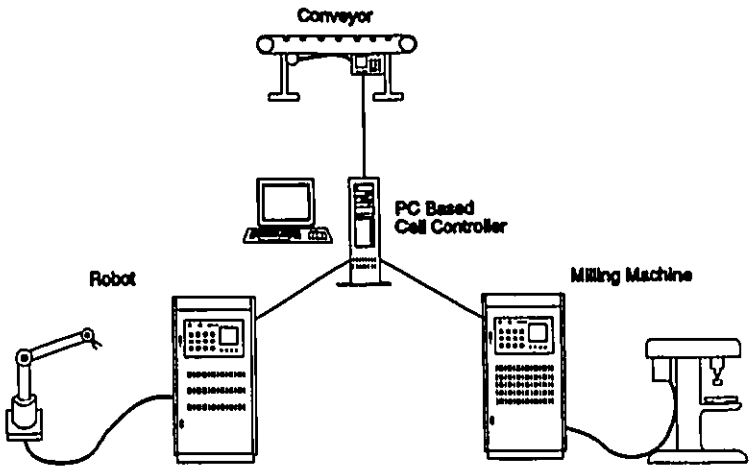

Figure 5.1 Cell Communication

UniSet cell programs are sequential. A program is translated in parts into the native languages of the constituent machines. For example only commands that are relevant to the robot operation are translated into the native robot language. Translations are performed

according to translation tables defined within the UniSet environment. A graphical representation of the translation results is shown in Figure 5.2.

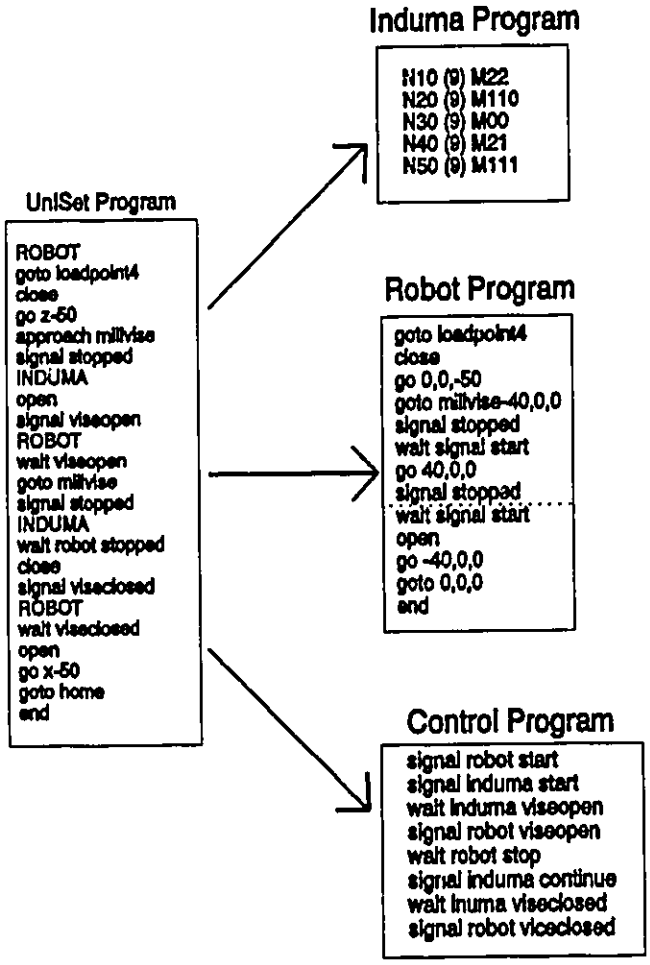

Figure 5.2 Program Translation

The left box in Figure 5.2 represents a UniSet program. It is created by the cell programmer. All events are programmed sequentially including code to control all the machines within the cell and their interactions. The benefit of this style of programming is that it is easy to convert from a conceptual understanding of machine interactions to a program. Since all

machines are controlled with one user program machine interactions are programmed naturally as the program's destination machine switches from one machine to the next. Programming in this manner is not possible when each machine must be programmed separately in distinct languages. It is even more difficult when programs must be written at each machine's separate controller.

Before a UniSet program can be used to control cell operation it must be converted into a format that the machines can utilize. The programs segments that apply to each individual machine are translated separately. Figure 5.2 shows the program segments following translation. The program segments are then transferred to each individual machine.

Along with translation of program segments the translation algorithm creates a program to be executed by the PC during cell operation. This is called the 'Control Program'. The control program supervises all machine interaction. If one machine is waiting on the completion of a task by another machine the control program ensures that the idle machine will not resume program execution until the control program has received notification, via a communication signal, that the appropriate tasks have been completed.

5.1 Syntax Analysis

An example of UniSet code is shown in Figure 5.3. Shown are the essential parts of a UniSet phrase. All phrases must refer to a destination machine. The phrase in Figure 5.3 identifies the **Robot** as the destination machine. Each phrase does not require the destination machine to be identified explicitly. Any phrase following **Robot** is interpreted as a command

for the robot until a new destination machine is identified. Phrases are composed of UniSet commands and data, both need to be translated.

Each machine name in the UniSet code must be the name of a **WorkStation** instance defined in the UniSet environment. Phrases are translated into the language of the most recent machine name.

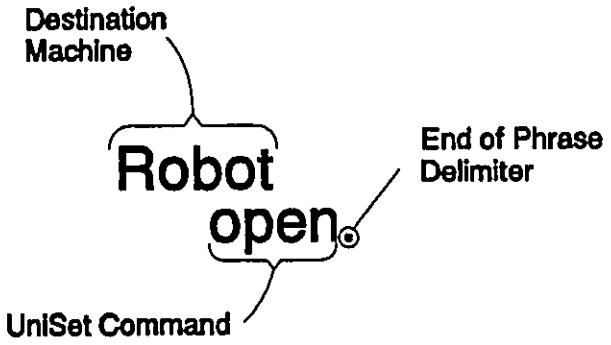

Figure 5.3 Program Segment

An instance of class **UniTranslator** is responsible for all syntax analysis performed on UniSet programs. A partial flow diagram for **UniTranslator**'s syntax analysis is shown in Figure 5.4. The figure shows the flow of the **UniTranslator**'s analysis during program translation.

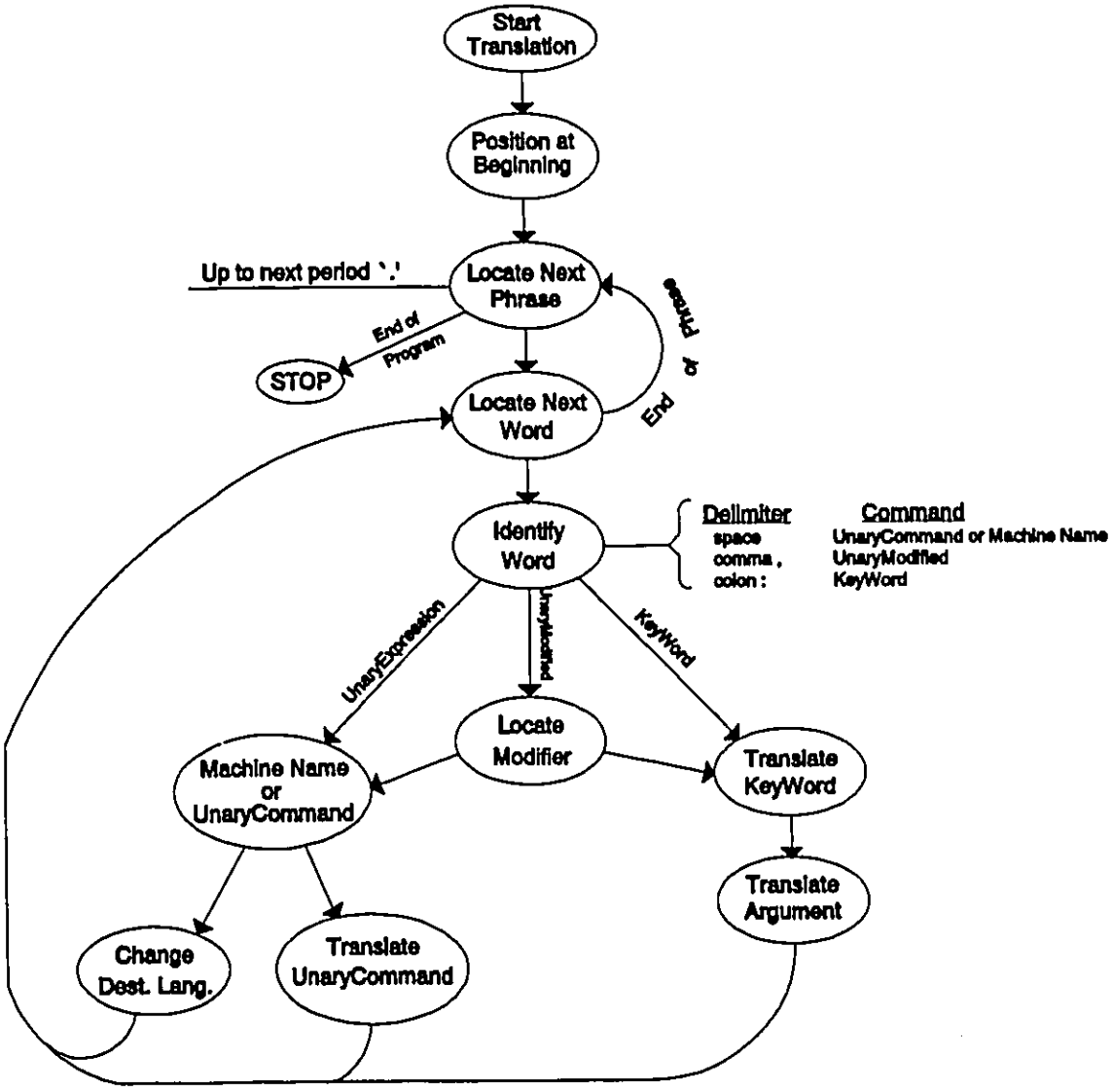

Figure 5.4 UniParser Translation Flow

During translation of a UniSet program the **UniTranslator** isolates and translates each phrase separately. A period '.' is the delimiter for each phrase. After a phrase is isolated it is parsed from left to right. The first word in a phrase is isolated. The next character is then analyzed, it is the command delimiter. The command delimiter is an indication as to the identity of the word. A space means that the word is either a **UnaryCommand** or a destination machine

Delimiter	Message	Meaning
. *period*	none	End of phrase
space	*unaryExpression*	**UnaryCommand** or Machine name
, *comma*	*unaryModified*	**UnaryModified**
: *colon*	*keyWord:*	**UnaryKeyword**

Table 5.1 Delimiter Table

name. A comma identifies the word as a **UnaryModified** command and a colon identifies a **KeyWord** command. All delimiters are shown in Table 5.1

When the **UniTranslator** identifies a machine name the machine instance is sent a message to retrieve its **UniLanguage**. The **UniTranslator** sets its instance variables and performs all following translations with the new **UniLanguage**'s command and data translation methods.

The next method executed by the **UniTranslator** is based on the delimiter. The delimiters are stored in a **Dictionary** that is assigned to the *GrammarDictionary* class variable of **UniTranslator**. The delimiters are the keys of the dictionary. The values are Smalltalk messages that the **UniTranslator** executes. Table 5.1 presents the messages associated with each delimiter.

5.1.1 UnaryExpression

A UnaryExpression can be either a machine name or a UnaryCommand. Both have the same syntax and both are shown in Figure 5.3. If a word is identified as a unaryExpression the first task of the *unaryExpression* method is to compare the word with the machine definitions. If the unaryExpression matches a machine name it is a machine identifier. Machine identifiers are not translated, they represent an indicator to the **UniTranslator** to change state in preparation for translating the instructions that follow. The **UniTranslator** changes state by finding the new machine's associated **UniLanguage** and assigning it to be the current language for translation.

If the word is not found to be a machine name it is assumed to be a **UnaryCommand**. The **UniLanguage** of the current machine is then searched to find the **UnaryCommand** instance. The **UnaryCommand** knows how it is to be translated. If the word identified as a **UnaryExpression** is not found to be a valid machine or UnaryCommand instance the word is incorrect and a syntax error message is displayed to the programmer.

5.1.2 KeyWord

If the character immediately following a word is a colon ':' then the word is a **KeyWord**. A UniSet phrase that includes a **KeyWord** is shown in Figure 5.5. A KeyWord phrase is not complete unless the KeyWord is immediately followed by an argument. To determine the translation of a **KeyWord** the **Uniparser** searches the active translation language, a **UniLanguage** instance for the **KeyWord**. If the command is not found in the current langauge

for translation an error message is generated informing the programmer of the syntax error. When a **KeyWord** command is identified in the current **UniLanguage** the word is translated as described in Section 5.2. Translation of the data following a **KeyWord** is described in Section 5.2.1

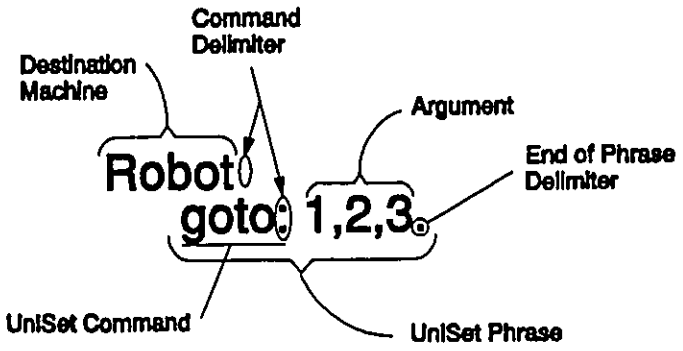

Figure 5.5 UniSet Program Segment

5.1.3 UnaryModified

If the character immediately following the first word in a UniSet phrase is a comma ',' then the first word is the prefix for a **UnaryModified** command. A phrase that includes a **UnaryModified** command is shown in Figure 5.6.

A **UnaryModified** command is composed of two words. When a **UniTranslator** encounters a comma, it must locate the word preceding and following the comma that constitute the **UnaryModified** command. The first word is the root word of the command and the word following the comma modifies, or clarifies, the meaning of the root word. The modifier may be either a **UnaryCommand** or a **KeyWord.**

Figure 5.6 includes a **UnaryModified** command with a **KeyWord** modifier. Shown in the figure are the delimiters that serve as indicators to a **UniTranslator** as to the class of instruction.

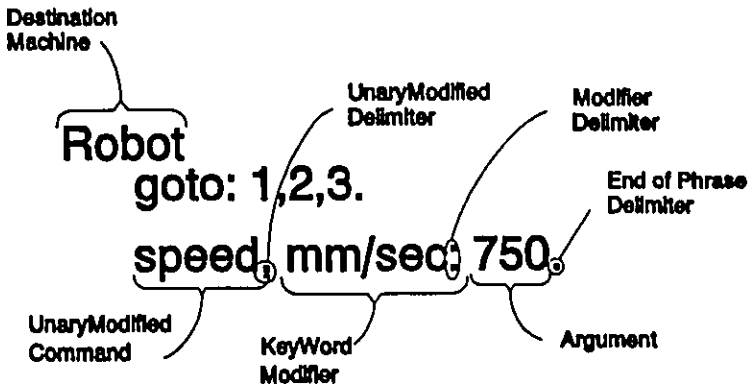

Figure 5.6 UnaryModified Program Segment

Once the class of modifier is determined translation of the command executes according to the modifier's class and does not differ from translating a **UnaryCommand** or **KeyWord** phrase. ? shows the flow of translation for all three types of UniSet commands. The flow for translating a **UnaryModifier** command branches after locating the modifier.

5.2 Instruction Translation

The unique characteristics of OOP allow the task of translation to be directed by individual UniSet instructions rather than by one central parser. This architecture allows each command to translate itself based on its own knowledge. Thus, the command, which has the most knowledge about translation performs this function.

In UniSet there are two different translation algorithms. The first method of translation is simple substitution. It is labelled 'Direct Tanslation' in Figure 5.7. This method is

employed if there is a one-to-one correspondence between a command in UniSet and a command in the machine's native language. If the command to be translated is not a **KeyWord** the translation involves replacing the UniSet word with a machine instruction. If the command is a **KeyWord**, the argument must be translated as well. Data translation is explained in detail in section 5.2.1.

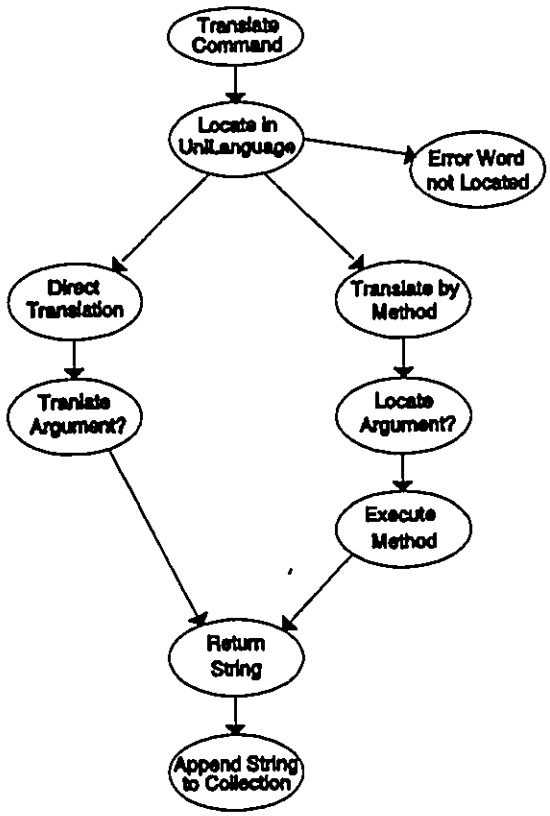

Figure 5.7 Command Translation

In contrast to a direct translation, the second type of translation is through the execution of a Smalltalk method that is specific to the command being translated. An example of this type of translation is to provide a circular interpolation command for a machine that does not provide circular interpolation motion in its native language. For example the ASEA language includes commands for circular interpolation but VAL II does not. To provide such a command in VAL II would require that the UniSet circular interpolation instruction output a series of linear interpolation commands in VAL II. This could be accomplished by writing a method to be called when translating in VAL II.

This second type of translation provides much more flexibility in developing commands, especially high-level commands. The programmer has all the powers of Smalltalk at his disposal. The disadvantage of implementing a translation method is that the implementor must be an experienced Smalltalk programmer. The Smalltalk method to be executed must be an instance method in the appropriate command class - either class **UnaryCommand, KeyWord** or **UnaryModified.** The translation method must return a string that is the complete translation of the UniSet phrase.

When translating a **KeyWord** UniSet command with a Smalltalk method the argument is not translated separately. Rather the argument accompanies the Smalltalk method to be used in the translation.

When building the UniSet language and translation tables the use of a Smalltalk method is specified by entering the message in the translation box of the **UniSet Language Builder** window. The message name must be preceded by '#' to indicate it is a symbol. See section 7.2.1 for further explanation.

5.2.1 Data Translation

Like instructions data must be translated. The automation languages studied in this thesis include the definition of several identical data types, but with different representations. Similar data types that have different representations among the languages are represented by one common structure in UniSet.

Each data class must contain a Smalltalk method for each language which requires translation. The message for these methods must be declared in the **UniLanguage** instance variable *dataTranslator*. The method must return a string which is the complete translation. For instance the method for translating a **Location** into ASEA format is shown in Figure 5.8. The method extracts the data contained in **Location** instance variables *x*, *y* and *z* and prints them to a **Stream** in the ASEA format. The contents of the stream are returned. The ASEA format for a **Location** is X=*x* Y=y Z=z. If a data translate method is not defined for a **UniLanguage** the *defaultTranslate* method will be used in its place.

```
1    asea
2            | stream |
3    stream := WriteStream on:''.
4    stream nextPutAll: 'X='.
5    x printOn: stream.
6    stream nextPutAll: ' Y='.
7    y printOn: stream.
8    stream nextPutAll: ' Z='.
9    z printOn: stream.
10   ^stream contents
```

Figure 5.8 Location Translation ASEA

Class Name	Description
Location	Three dimensional point.
Rotation	Rotations in degrees about the Cartesian axes.
Cubic	A three dimensional rectangular volume.
Cylinder	A cylindrical volume.
Transformation	A translation and rotation.

Table 5.2 Data Classes

Table 5.2 lists the data types declared in the UniSet environment. Data types must be declared within the Smalltalk class hierarchy. Construction of data type classes is performed in the *Class Hierarchy Browser* in the traditional Smalltalk manner. Use of the new data types follows the rules of all other classes within the Smalltalk hierarchy.

When using an argument in a UniSet program any Smalltalk method may be used. Any information following a **KeyWord** command is evaluated using the Smalltalk compiler.

CHAPTER 6

SOFTWARE DESIGN

This chapter provides a general description of the most important UniSet classes. The classes described include **UniLanguage, UnaryCommand** and its' subclasses, **Data** and its' subclasses and **UniTranslator.** The graphical notation is a subset of that presented in Booch's book *Object Oriented Design with Applications* [45]. A summary of the notation is found in Appendix A.2. The classes that implement UniSet's graphical user interfaces are described in Chapter 7.

6.1 Top Level Design

The top-level design of the UniSet environment and the relationships between the major classes are shown in Figure 6.1. The **FMC** class is not part of either the implementation of the **UniEditor** and **UniTranslator** classes but must be visible to both. The **UniEditor** class needs to have access to the **FMC** class to enable the UniEditor GUI to be context sensitive based on the cell being programmed. Class **UniTranslator** must be have access to the **FMC** class during translation. Thus, both **UniEditor** and **UniTranslator** have interface relationship with class **FMC.**

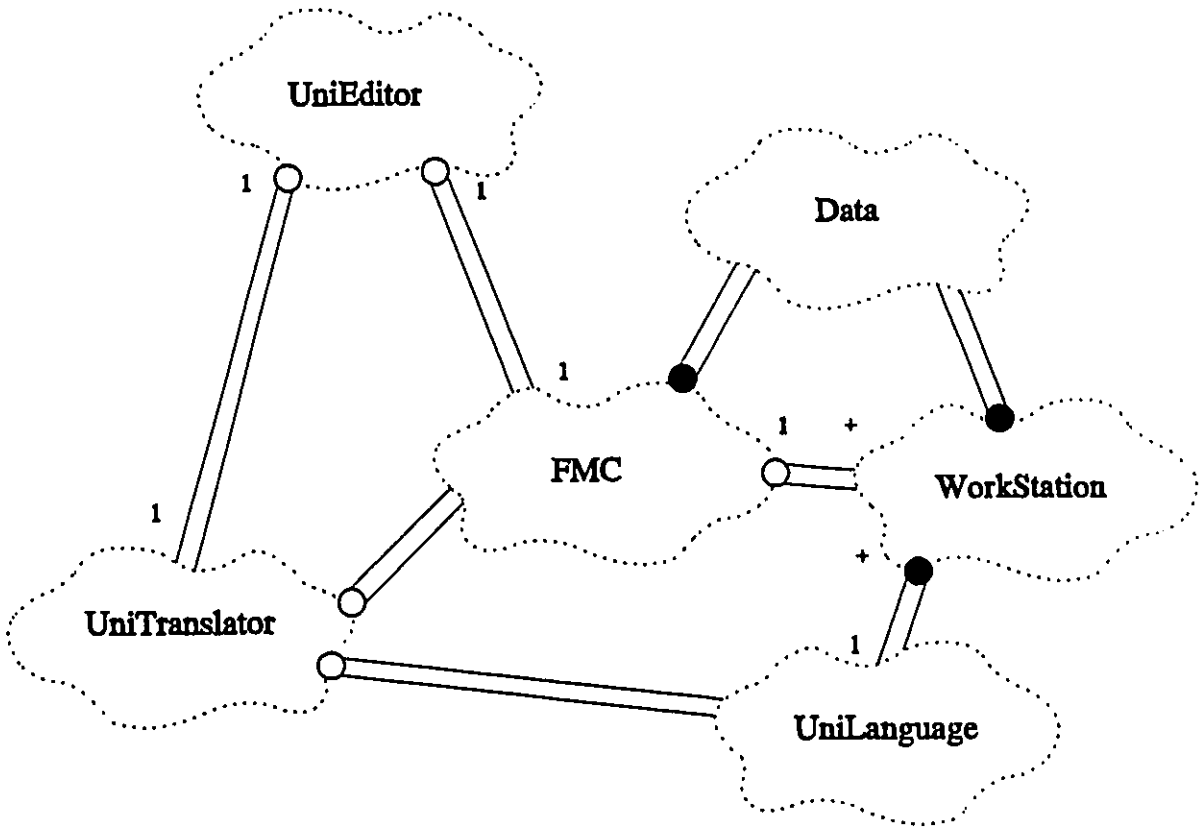

Figure 6.1 Top-Level Class Diagram

Class **UniEditor** and **UniTranslator** require access to the information contained in the classes **UniLanguage, WorkStation** and **Data.** The information contained in instances of these classes is employed during programming and translation. These classes are hidden from view from both classes and they are only accessible through the **FMC** class interface. There is one exception, during translation of a program class **UniTranslator** accesses the interface of class **UniLanguage** while bypassing the **FMC** class. This was done because of the large amount of information that **UniTranslator** requires of each **UniLanguage** during translation.

Class **UniEditor** accesses the **Data**, **Workstation** and **UniLanguage** classes through the **FMC** interface because a UniEditor window operates on these classes only in the context of programming an FMC. The cardinality relationship of one **UniEditor** to one **FMC** shows that for every one **UniEditor** there must be only one **FMC**. This is a practical restriction, one UniEditor cannot be used to program more than one FMC at a time. The reverse should be a manner of practice. It is possible to open more than one UniEditor window on one FMC but the usefulness and possible confusion should limit the use of one UniEditor window per FMC.

Translation of a program is launched from within the UniEditor window. The **UniTranslator** class is visible to **UniEditor**, and its instances are instantiated by it. There should be only one **UniTranslator** instance per **UniEditor**.

6.2 UniLanguage

Instances of class **UniLanguage** store translation information. Each **UniLanguage** instance is a collection of UniSet commands and translations for one automation language. A UniLanguage instance can be used for more than one machine. For instance two milling machines with the same controller would share the same Word Address code and thus be able to share the same UniLanguage instance.

The **UniLanguage** class has two instance variables. The first instance variable, *dataTranslator*, contains one instance of class **Symbol**. The symbol is the translation message for instances of **Data** subclasses in order to perform translations (see section 5.2.1). If the

symbol's value is nil, class **UniTranslator** will use the *defaultTranslate* method of the **Data** class for translation.

The second instance variable, *commands*, is a **Dictionary** instance. Class **Dictionary** is a native Smalltalk class. For information on its implementation and interface refer to the Smalltalk manual [44]. The **Dictionary's** keys are UniSet commands. The values are instances of **UnaryCommand** and its subclasses (See Figure 6.2).

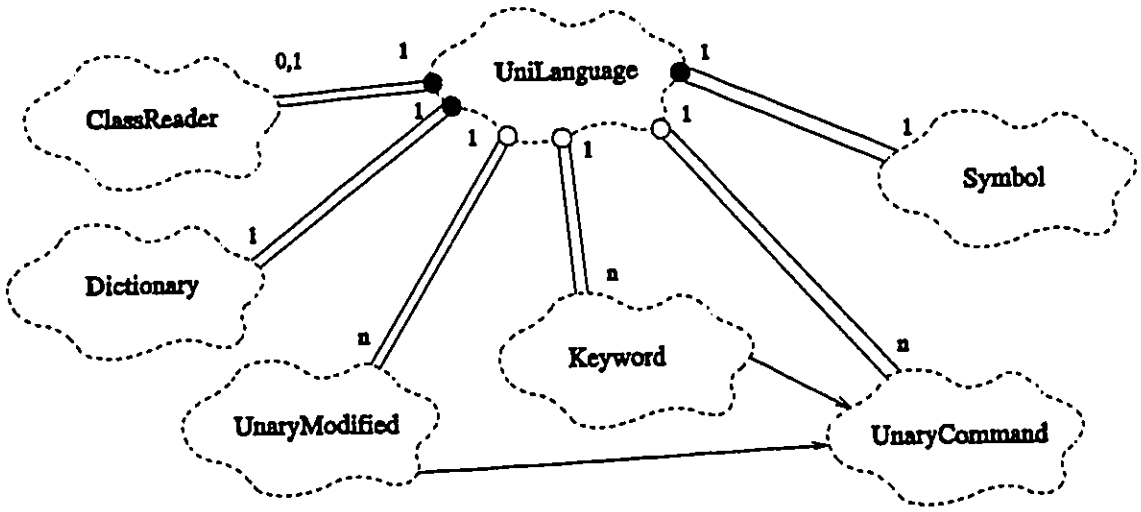

Figure 6.2 UniLanguage Class Diagram

Class **UnaryCommand** and its' subclasses are also part of **UniLanguage's** class structure. There is an interface relationship between these classes because **UnaryCommand** instances are not part of a **UniLanguage** instance. The instances are just stored in a **UniLanguage** instance.

UniLanguage also uses for its' implementation the **ClassReader** class when saving to disk. While in the 'Disk Services' UniSet interface the operator has the opportunity to save

UniLanguage, **Workstation**, and **FMC** instances to disk. The action of saving instances is initiated in the window but the save message is sent to the selected instance. It is the required that each class have a method that will save the instance to disk. During execution of the 'saveInstance' from the **UniLanguage** class that a **ClassReader** is instantiated. The **ClassReader** instance takes care of all the details of saving the instance to disk. Immediately after the ClassReader instance completes the save the it is no longer needed and disposed of through Smalltalk's garbage collection system. The transitory nature of **ClassReader** instances is shown in Figure 6.2 with a cardinality of zero or one beside the class. Class **ClassReader** is a native Smalltalk class, further details regarding the class can be found in the Smalltalk manual.

UniLanguage instances are passive, they respond to messages by supplying **UnaryCommands** contained within the **Dictionary**. The **UniLanguage** class does not initiate messages on other classes, except for **ClassReader** during 'saveInstance' operations.

6.3 UnaryCommand & Subclass

UniLanguage's instances serve as retainers for instances of **UnaryCommand** and its subclasses. The **UnaryCommand** class design is shown in Figure 6.3. It uses for its'

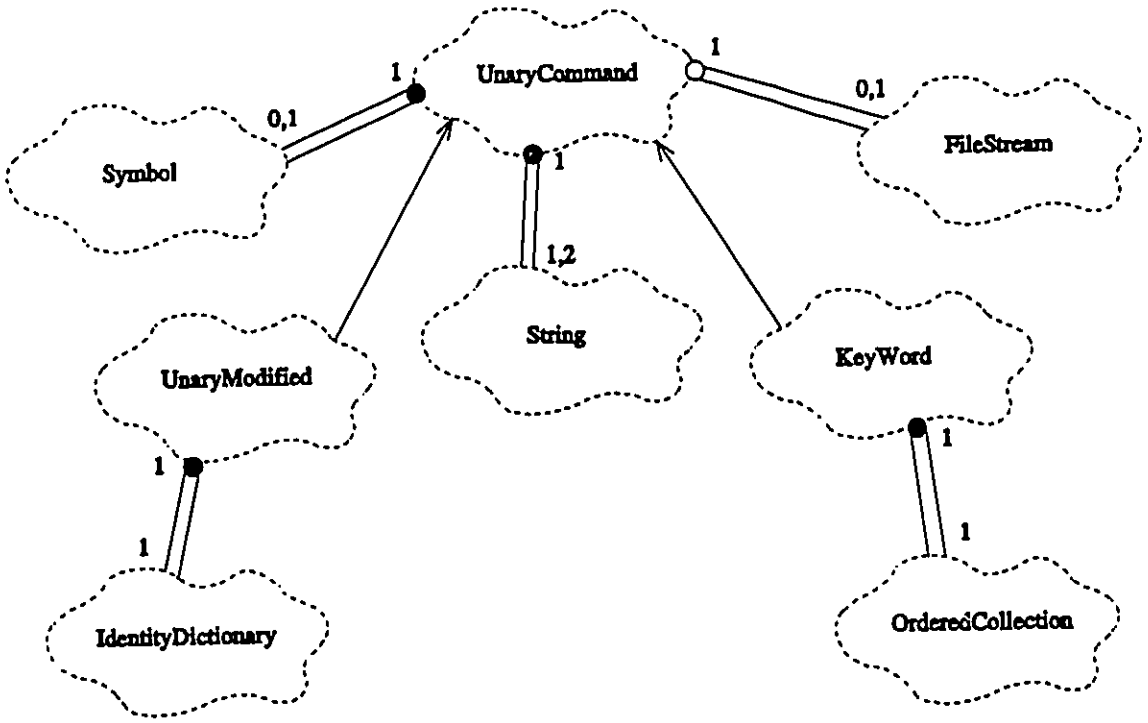

Figure 6.3 UnaryCommand Class Diagram

implementation classes **Symbol** and **String**. The instance variable *translation* is either a **String** or **Symbol**. The two options for the *translation* instance variable correspond to the two methods for translating a command - either direct substitution or by a Smalltalk method. If variable *translation* is a **String** the string represents the direct translation of the command. If *translation* is a **Symbol**, the symbol is the Smalltalk message to be sent to the **UnaryCommand** that will return the translation string. The variable cardinality shown for the **String** and **Symbol** classes

in Figure 6.3 represents the two possibilities. The **String** class is used a second time within the **UnaryCommand** implementation for storing a comment about the command. The comment is entered by the programmer when creating a translation within the '*UniSet Builder*' window. The comment is intended as an explanation for the command and could be used by a on-line help system.

Classes **UnaryModified** and **KeyWord** are subclasses of **UnaryCommand**. They have one additional instance variable each.

UnaryModified class includes the instance variable *modifiers*. It is assigned an **IdentityDictionary**. The keys of the **IdentityDictionary** are **Strings** and each is a modifier for the **UnaryModified** command that holds the dictionary. The values associated with the strings are either **UnaryCommand** or **KeyWord** instances. A UniSet programming statement that includes a **UnaryModified** includes the **String** for the root command and one of the **Strings** from the keys in the *modifiers* **IdentityDictionary**.

An example of UniSet code that includes a **UnaryModified** instruction is shown in Figure 6.4. The UniSet instruction is shown in the upper right of the diagram, its destination machine is the 'Mill' and the desired action is the activation of the coolant. A sample of the Mill's **UniLanguage** is shown including several commands. The **UnaryModified** command *coolant* is one of the keys in its **UniLanguage**. Each **UnaryModified** command contains a **IdentityDictionary** in the *modifiers* instance variable. The *on* modifier is found within the keys of *coolant*'s *modifier* dictionary. The command instance, in this case a **UnaryCommand** that is the value for the key contains the full translation of the root word and modifier combination.

Because of this the translation of a **UnaryModified** command is identical once the root word and modifier are used to yield either a **UnaryCommand** or **KeyWord** instance.

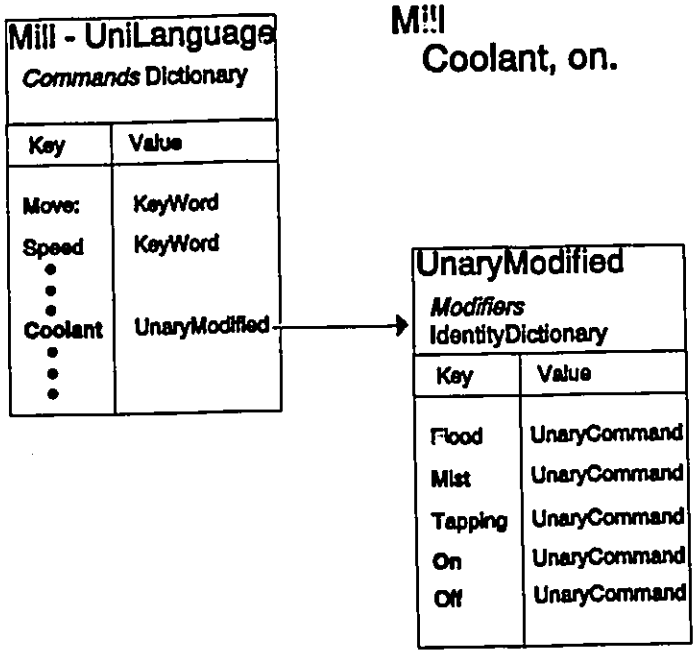

Figure 6.4 UnaryModified Example

Class **KeyWord** allows a UniSet command to be accompanied by an argument. Its' instance variable *inputParameter* is assigned an **OrderedCollection** of classes that includes all possible arguments for the **KeyWord**. During translation of a **KeyWord** instance, the argument is compared to the elements of *inputParameter* until its class matches one of the elements. If the search is not successful the translation fails and generates an error indicating that the class of argument is not suitable for the chosen instruction. A walkback window opens displaying the message 'UnSuitable Argument.'

An object diagram of a **UnaryCommand** is shown in Figure 6.5. The figure shows that a **UnaryCommand** instance only shares a **FileStream** instance with other instances. No other parts of a **UnaryCommand** are visible to its clients. **UnaryCommand** uses the fileStream for saving itself to disk. During saving it sends the message *saveOn:* to all of its components to be

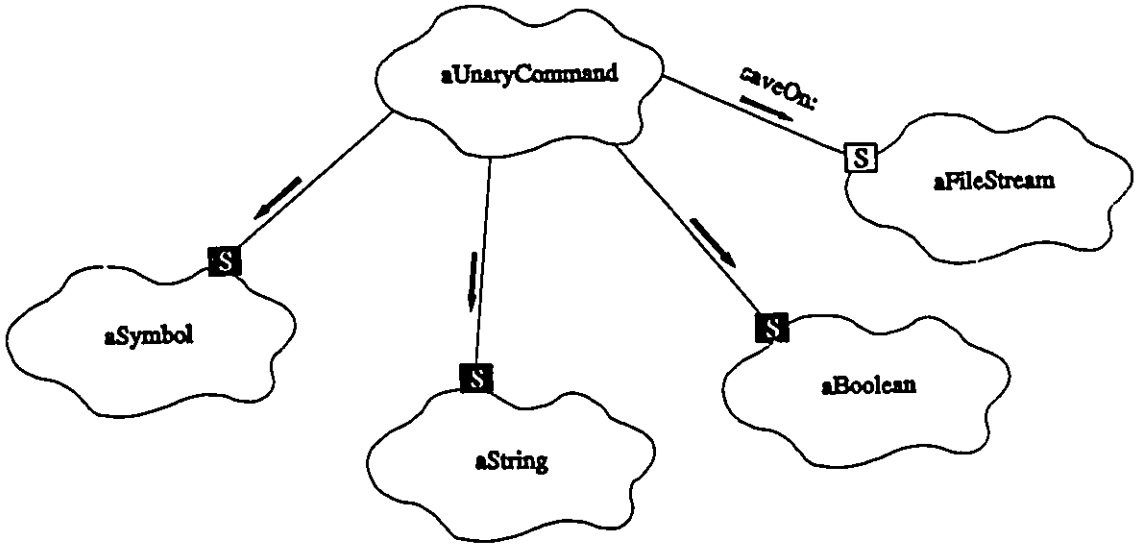

Figure 6.5 UnaryCommand Object Diagram

properly recorded in the open file. Depending on whether the translation of the command is direct either the translation will be held in the **String** instance or if the command is translated with a Smalltalk method the **Symbol** contains the message that will translate the **UnaryCommand.**

UnaryCommands are instructions that are not accompanied by arguments. They find application in the UniSet environment for simple commands such as *stop* or as the modifier of **UnaryModified** commands.

Figure 6.6 is the object diagram for **UnaryModified** instances. It is the same as the object diagram for **UnaryCommand** instances except it includes an **IdentityDictionary** instance for storing the modifier translation information. This information is not shared and is part of

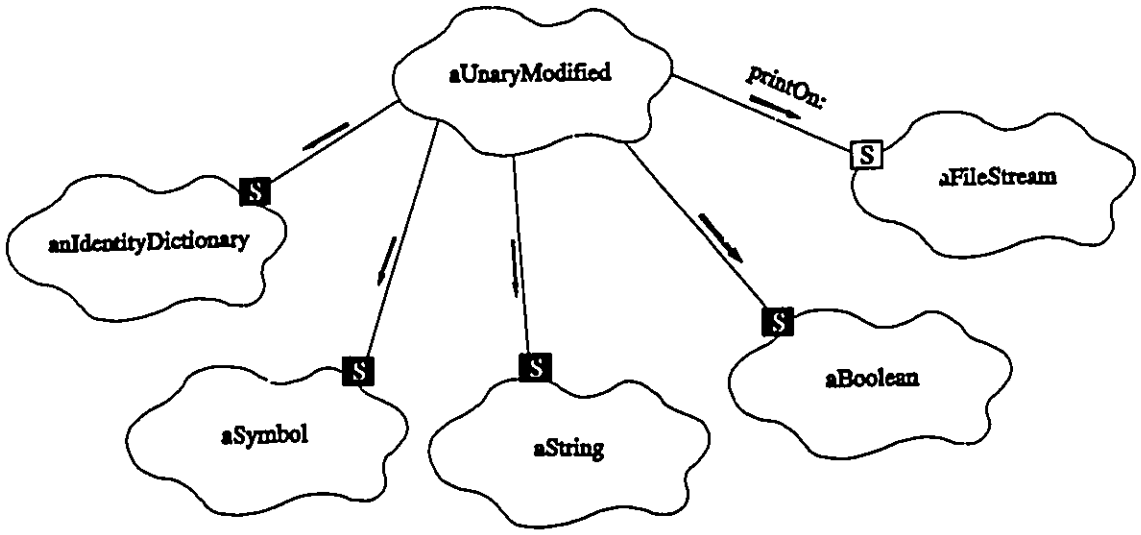

Figure 6.6 UnaryModified Object Diagram

the encapsulated inner details of the class. For a command such as *coolant* shown in Figure 6.4 the modifier instances are held in the **IdentityDictionary**.

An object diagram for **Keyword** instances is shown in Figure 6.7. This diagram is also similar to the **UnaryCommand** object diagram except for the addition of an **OrderedCollection** instance. An OrderedCollection is used to store the class names of the arguments that can legally be used with the **KeyWord** instance. The classes kept in the **OrderedCollection** are used for error checking during translation. For example *goto: 10,20,0* and *goto: false* are both syntactically syntactically **Keyword** instructions but semantically the second instruction does not make logical sense because of the argument. The argument for *10,20,0* is an instance of class **Location** while **false** is an instance of class **False**. By specifiying the class of acceptable

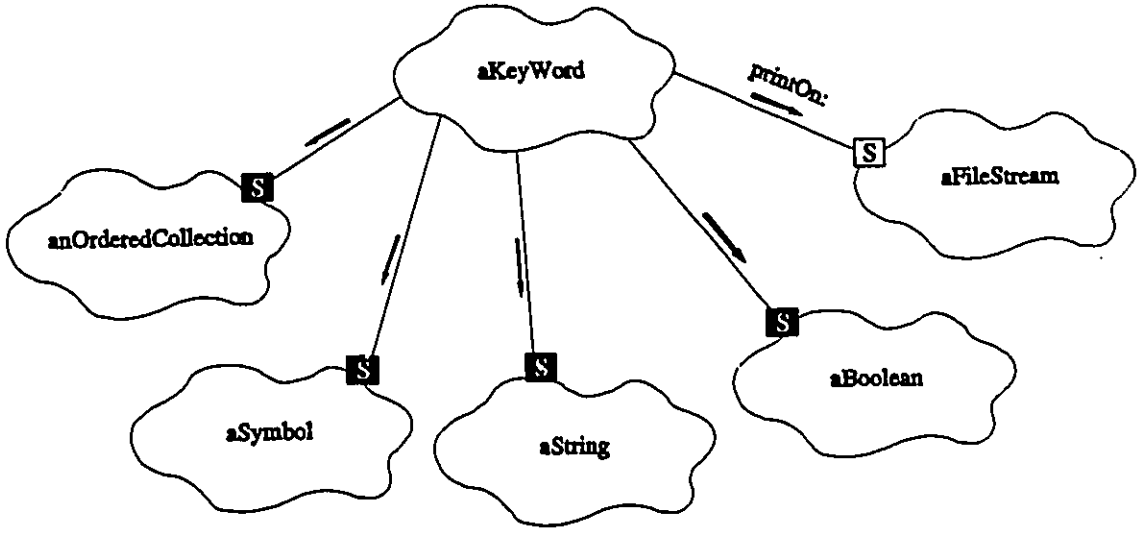

Figure 6.7 KeyWord Object Diagram

instructions during translation arguments are evaluated to ensure that they are of an appropriate class.

6.3.1 UnaryCommand & Subclass Protocol

All instances of **UnaryCommand** and its subclasses respond to a framework of messages. The framework is shown in Table 6.1. Most of the methods alter or retrieve information about individual commands.

Methods *saveOn:* and *buildFrom:* are for writing and retrieving the instance from disk. *SaveOn:* writes a string representation of the receiver command to a **Stream**, usually a **FileStream** for writing to disk. The method *buildFrom:* can decipher the format created by method *saveOn:*. It is sent by a **UniLanguage** class while building a **UniLanguage** from data

buildFrom: aString (Class method)	Builds a **UnaryCommand** or subclass instance from aString.
comment	Returns the comment.
comment: aString	Sets the comment.
saveOn: aStream	Writes the command format to aStream.
translate	Translates the command.
translationString	Returns a string representation of the translation.
translationString: aString	Stores the information in the string.

Table 6.1 UnaryCommand Protocol

on a disk. It is a class method because it must be sent to a **UnaryCommand** class or subclass with a **String** from which it creates an instance of **UnaryCommand** or subclass based on the information in the **String**. Both of these methods must be defined in every **UnaryCommand** class. Each class has unique instance variables that require a special format for saving the relevant information.

A sample of a command that has been saved to a file is shown in Figure 6.8. This sample is only a segment of a complete language definition that has been saved to file. Included is all the information to completely rebuild a **UniLanguage**. The first line identifies the **UniLanguage** instance that is saved in the file, in this case **VALII**. Brackets '{ & }' surround each **UnaryCommand** description. Following the opening bracket is the UniSet command, *Goto:*, following the arrow is the command's class, **KeyWord**. The 'T' between the vertical

Disk File	Comments
UniLanguage:'VALII'	Language Header
{Goto:→KeyWord	UniSet command & Class
¦T¦'MOVE <arg>'	*DirectTrans* instance variable value & Translation
'Absolute Positioning, Joint Interp.'	Comment

Figure 6.8 KeyWord Command Disk Save Format

lines is the value of the **UnaryCommand** instance variable *directTrans*. A true value means the next portion of the file is the translation of the command. A false value means the next remainder of the line is the Smalltalk message to be executed for translation. The third set of single quotes encloses the comment with the command instance. The square brackets enclose the class name or names of the arguments that can accompany the command. UnaryCommand instances would not include the square brackets because they do not accept arguments. A complete description of a language would include many commands.

6.4 Data Classes

Data can be used as arguments for **KeyWords**. Class **Data** is an abstract class, its class diagram is shown in Figure 6.9. The most commonly used subclass of **Data** is class **Location**. Class **Location** is similar to the Smalltalk class **Point** except that **Location** is three dimensional instead of two. Classes **Location** and **Point** share a similar protocol. **Location** has three instance variables, *x*, *y* and *z*, that represent the cartesian coordinate system.

Class **Rotation** also has three instance variables, *thetaX*, *thetaY* and *thetaZ*. They represent rotations about the cartesian axes whose values should be **Numbers** representing angular displacement in degrees.

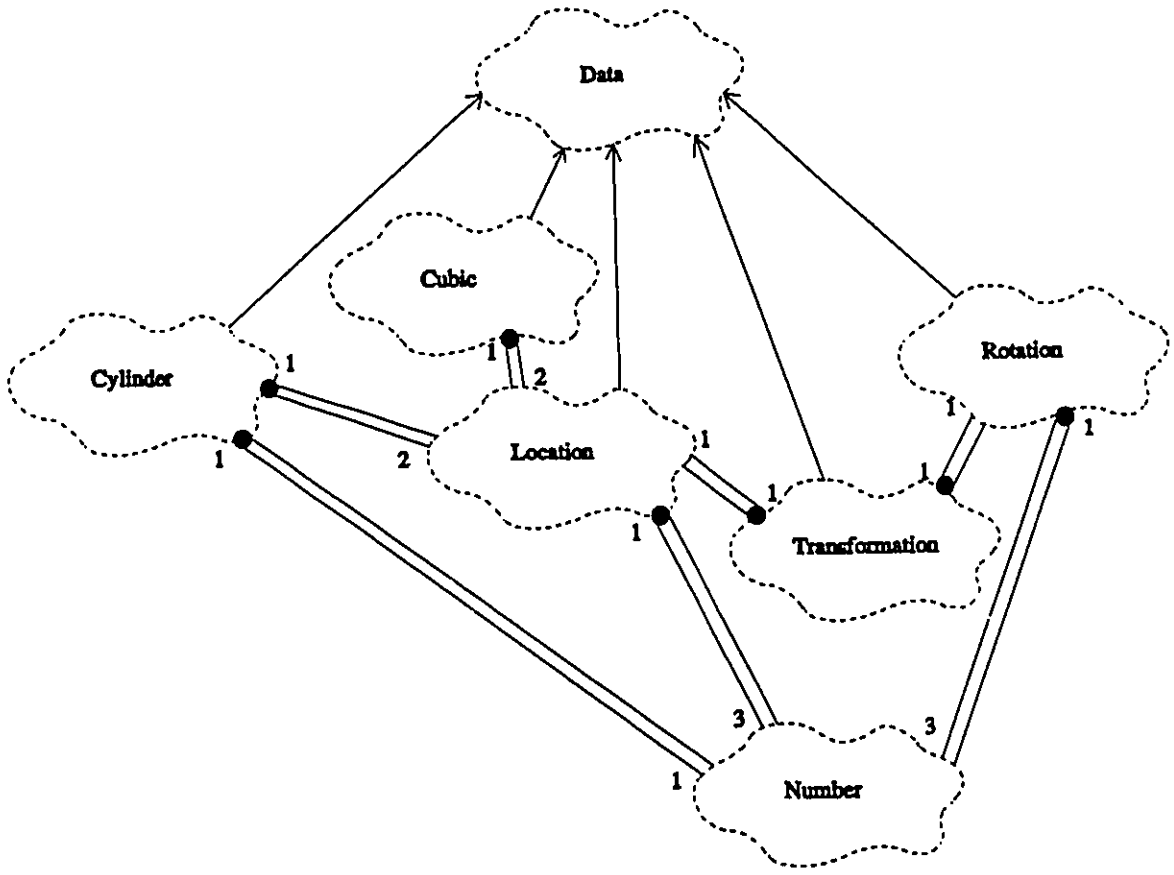

Figure 6.9 Data Class Diagram

Class **Transformation** inherits from the **Data** class and uses **Location** and **Rotation** to represent a machine independent location and orientation. Transformations are typically used in robot programming. Vall II contains commands for defining transformations.

Class **Cubic** is similar to the Smalltalk **Rectangle** class except that its *origin* and *corner* are **Locations** rather than two dimensional **Points**. Class **Cubic** and **Rectangle** share similar protocols.

Class **Cylinder** represents a cylinder with its axis defined by the **Locations** *origin* and *end*. The definition of a **Cylinder** is completed by the *radius* which may be any numeric value. **Cylinders** and **Cubics** define volumes. Both classes can be used to define physical obstructions within a cell, such as the work volume of a milling machine or the bed of a lathe. Volume definitions can be used to prevent collisions within the cell. Error checking of robot paths against defined volumes could provide warnings before program execution that would prevent a programmer accidentally instructing a robot to move through the volume occupied by the bed of a lathe. These classes share similar protocol. The instance method 'containsLoc:' must be accompanied by a **Location** instance as the argument. This method returns 'True' or 'False' depending on whether the location is within the boundaries of the receiver.

The object diagram for a **Transformation** instance is shown in Figure 6.10. A transformation has location and rotation fields. Its diagram is representative of any instance of the **Data** subclasses. **Data** subclasses do not share their

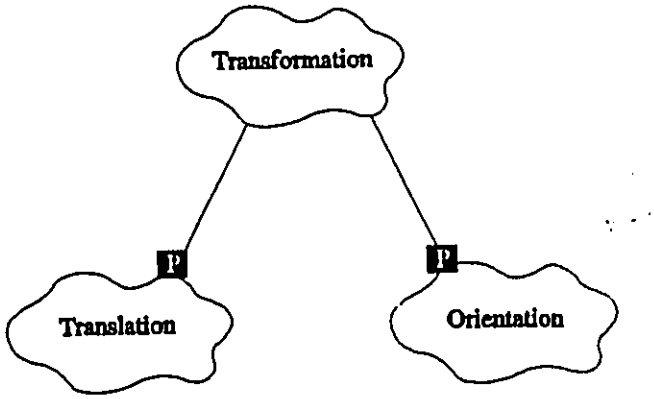

Figure 6.10 Transformation Object Diagram

implementation with any other objects. All **Data** subclass objects have unshared fields that use either the **Location** or **Number** classes.

6.5 UniTranslator

The function of the UniTranslator is to parse the UniSet code created in the editor, into its constituent components. The parser finds each phrase in the UniSet program, isolates it and finds the top level UniSet command of the phrase. Command instances are sent the message *translate:* along with the rest of the command phrase sent as an argument. If the command phrase requires further translation the phrase is redirected back to the UniTranslator for further parsing.

The **UniTranslator** is able to identify the command instance of every word by delimiters. The delimiters and their meanings (described in 5.1) are kept in a **Dictionary**. If more

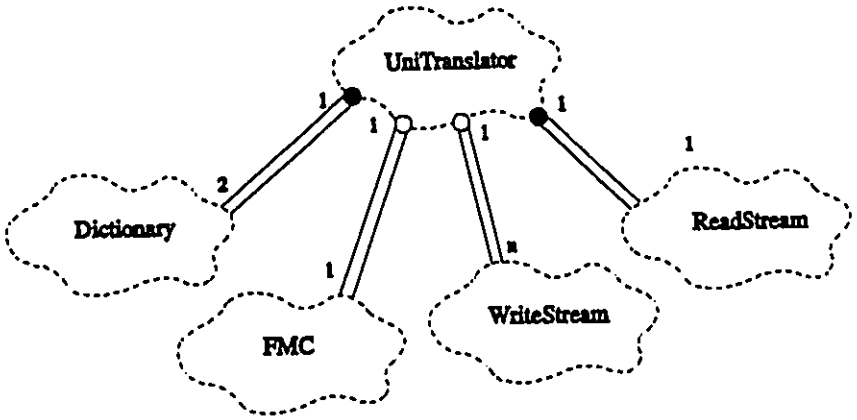

Figure 6.11 UniTranslator Class Diagram

command patterns need to be added, special delimiters can be added to the **UniTranslator** class in the **GrammarDictionary**.

The *programFiles* variable points to the other **Dictionary** used by the **UniTranslator**. During translation every machine has an associated **WriteStream** held in this **Dictionary**. The keys of the **Dictionary** are the names of the machines in the cell. Translations of the UniSet program created in the UniEditor window are written to the **WriteStreams**.

The **UniTranslator** class has access to the interface of **FMC** class. Class **UniTranslator** must be visible to the cell and all its constituent machines. The visibility is necessary because translation of the UniSet program is based on the machine instances. Machine instances include **UniLanguage** instances that are accessed by the **UniTranslator** instances

Class **ReadStream** used by class **UniTranslator** is provides easy access and manipulation of the UniSet program during translation. When the command for translation of the UniSet program is issued a **ReadStream** is instantiated with the UniSet program as its contents. Class **ReadStream** is a native Smalltalk class that includes all necessary methods to stream over the contents and extract instructions from its contents.

6.6 FMC

The **FMC** class is not fully developed in its present implementation. It serves as a means to group **WorkStation** instances together and retain information about their relationships in the

cell. It is possible to have a **Cell** instance without any **WorkStation** instances but such a cell would have no basis in reality.

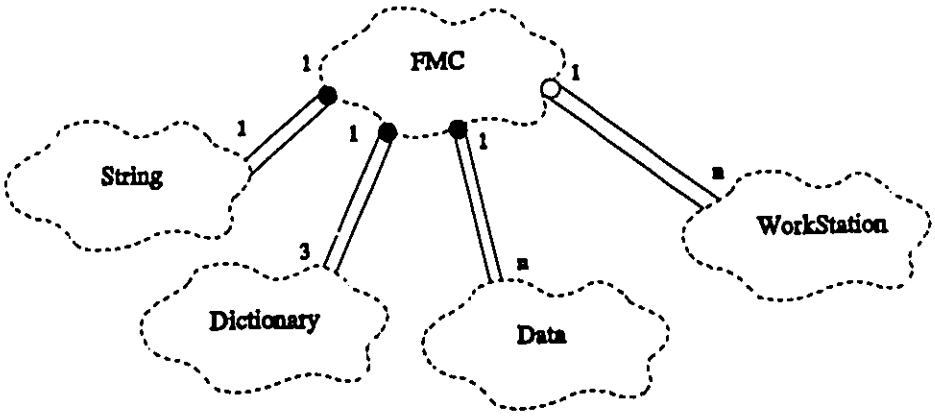

Figure 6.12 FMC Class Diagram

Figure 6.12 is an **FMC** class diagram. The **String** instance contains the **FMC**'s name and should be a meaningful identifier for the cell operator.

Class **FMC** uses three **Dictionary** instances. One of the **Dictionaries** holds **WorkStation** instances, the **WorkStation** names are the keys. An other **Dictionary** holds UniSet programs that have been created for the cell instance. The **Dictionary**'s keys are the program names. It is logical to store the programs with the cell since they can not be used without the defined cell. The third **Dictionary** is used for storing cell variables. Variables declared in this **Dictionary** should be cell specific. An example of a cell variable might be a shape definition of an obstacle within the cell volume, or the number of parts to be machined by the cell.

6.7 WorkStation and its Subclasses

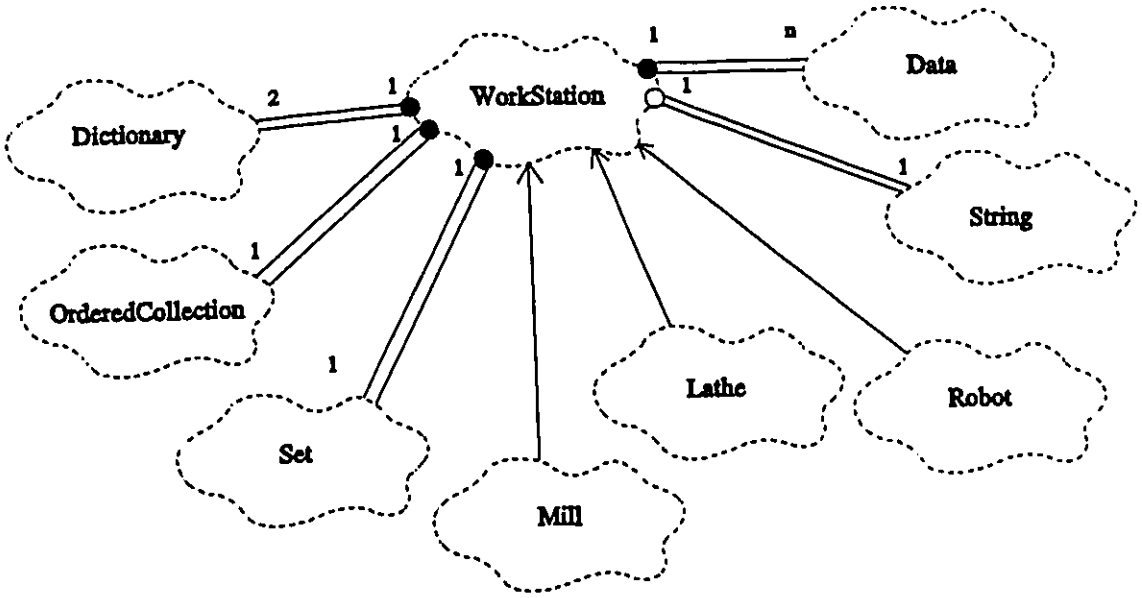

Figure 6.13 WorkStation Class Diagram

Class **WorkStation** is an abstract class. It is the superclass of all machine classes within the UniSet environment. Any future machine classes should also be defined as subclasses of **WorkStation.** Class **WorkStation** provides much of the structure and functionality for its subclasses. In particular it implements the protocol required by the graphical user interfaces.

The **WorkStation** class includes one pool dictionary called *MachineIcons*. *MachineIcons* contains all the forms to be used as icons with the UniSet Editor window. As each machine class is created it should be assigned an icon. The icon is used to represent the machine in the **UniEditor** programming window.

There are two important **WorkStation** class methods. The method *classInit* should be executed only when the **WorkStation** class has just been filed in. It initializes the class' pool dictionaries and class variables. Executing the method at any other time may wipe out valuable information. Executing it assigns a new **Dictionary** to the *MachineIcons* pool dictionary, a new **OrderedCollection** to the *MachineClasses* class variable, and a new Set to the *MachineInstances* class variable.

```
classInit
    "To be executed immediately after
     filing in the WorkStation class."
MachineClasses := OrderedCollection new.
MachineIcons := Dictionary new.
MachineInstances := Set new
```

Figure 6.14 *classInit* Class Method

The second class method of significant importance is *compileAll*. It overrides the *compileAll* method inherited from **Object**. The method was implemented to give the user control over the **WorkStation** subclasses that are added to the *MachineClasses* collection. *MachineClasses* is a collection of concrete **WorkStation** subclasses. The user needs this control because in Smalltalk there is no reliable means to distinguish between abstract and concrete classes. The classes added to this collection should not include abstract classes because abstract classes may not have instances. This prevents the user from having to know which **WorkStation** subclasses are abstract and which are specialized enough to be instantiated.

The *compileAll* method is executed automatically whenever a new class is created. When a **WorkStation** subclass is declared the user is asked whether the new class should be added to the *MachineClasses* collection. The new class should only be added if it is a concrete class.

```
compileAll
    "Will be executed with every new subclass.
     The machine should be added to the MachineClasses
     if it is not an abstract class."
(Menu confirm: 'Add ', self name,' to MachineClasses?')
    ifTrue:[MachineClasses add: (self name)].
^super compileAll
```

Figure 6.15 *compileAll* Class Method

Class **WorkStation** is not included in the collection because it is abstract and should never have instances. After the user makes the choice the method passes control to the *compileAll* super method of class **Object**. When a new machine is instantiated the user is asked to choose the class of the machine from a menu. The list displayed in the menu is based on the *MachineClasses* collection.

The currently defined icons are shown in Figure 6.16. The icon forms can be created using the **FreeDrawing** editor provided with Smalltalk. The forms should be no larger than twenty bits square. After creating a form in a **FreeDrawing** editor the form can be transferred to the *MachineIcons* pool dictionary by executing the following expression.

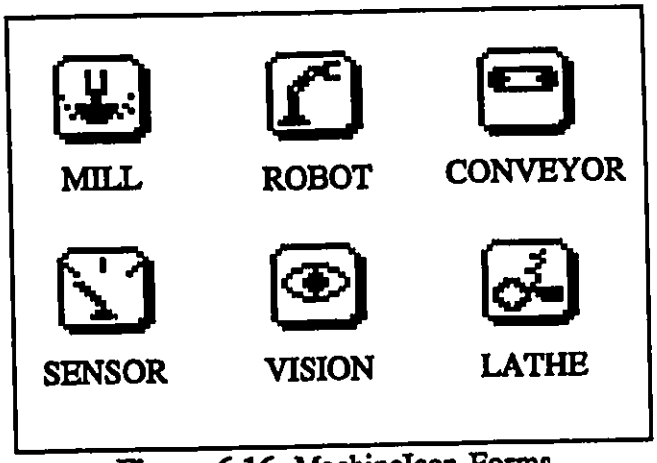

Figure 6.16 MachineIcon Forms

WorkStation machineIcons at: *formName* put: (FreeDrawing pictureDictionary at: *formName*)

There are many additional classes that could be defined as subclasses of class **WorkStation.** For example classes **Mill** and **Lathe** could be made more specific

CHAPTER 7

THE TEXTURAL ENVIRONMENT OF UNISET

The UniSet environment includes three Graphical User Interfaces (GUI's). In traditional Smalltalk fashion the interfaces allow keyboard and mouse input.

7.1 Disk Services Interface

7.1.1 Functionality

Figure 7.1 is a screen capture of the Disk Services Interface. It provides the ability to save UniSet objects to disk. The left pane's contents are controlled by the three buttons at the very bottom of the window. The three buttons 'Machines', 'Cells' and 'Languages' initiate a UniSet memory walk that returns all the instances of the button pressed and lists them in the left pane. The list shown in Figure 7.1 displays language instances as a result of pressing the 'Languages' button.

The bar running across the lower middle of the screen contains a button for every available disk drive. Picking one of the buttons alters the Directory and File Panes. The Directory Pane on the right side of the window displays all the directories for the selected disk.

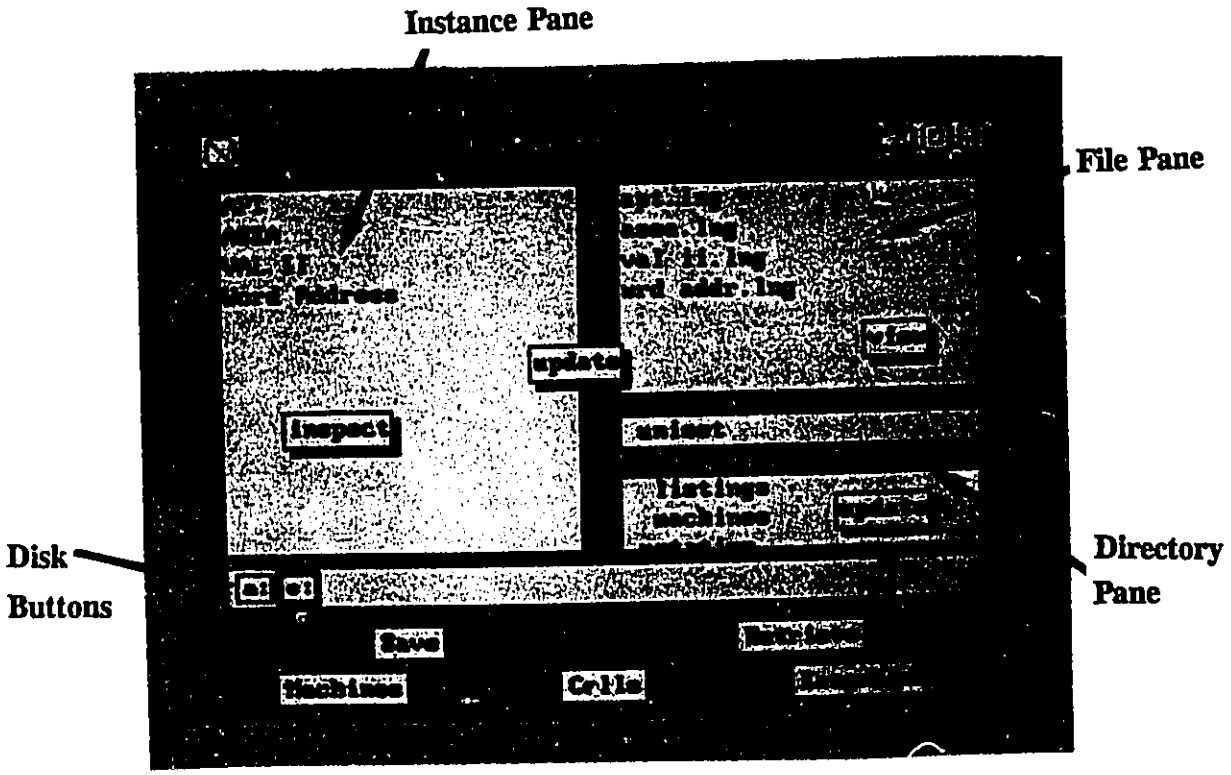

Figure 7.1 Disk Services Interface

If a directory has subdirectories its name is followed by three dots. Double clicking on such a directory name with the left mouse button will expand the list of subdirectories, allowing subdirectory selection. The File Pane displays the files contained in the selected directory.

The button 'Save' at the bottom of the screen saves the instance highlighted in the Instance Pane to the selected directory. File extensions are assigned as follows:

Instance of	Extension
UniLanguage	*LNG*
WorkStation & subclasses	*MCH*
FMC	*CLL*

If there is question to the identity of any instances listed in the Instance Pane, they can be inspected through the *inspect* selection from the Instance Pane menu.

To load a previously saved instance into UniSet, select the appropriate file and click the 'Retrieve' button. If there is some question as to which file is correct the contents of the file can be viewed by through the *view* option of the file pane menu.

7.1.2 Disk Services Implementation

The implementation of the Disk Services window follows traditional Smalltalk windows implementation. The Disk Server Class Diagram is shown in Figure 7.2. The window and its subpanes are initialized during the *open* instance method of class **UniDiskServer**. Implementation of the window uses **TopIconPane** and **IconPane**, two classes not provided by Smalltalk. They both allow a more flexible employment of icons than the vendor supplied windows classes. Both classes are fully documented in B. The three **Collections** used by the **DiskServer** class are used to keep lists for display in the three list panes. The **Directory** class

is a native Smalltalk class. It provides the ability to access a directory, retrieve a list of files in the directory, access a file in the directory or create a new file.

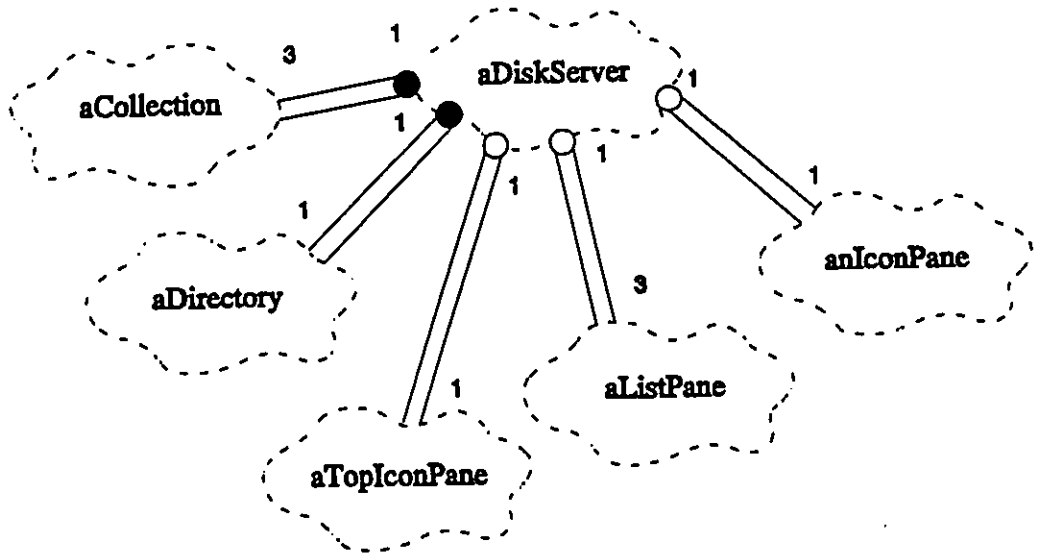

Figure 7.2 Disk Server Class Diagram

Figure 7.3 is the object diagram for a DiskServer window. The instances of **Pane** subclasses are dependents of the DiskServer. There are three **ListPane** instances per DiskServer window. The **TopIconPane** is used in place of the Smalltalk class **TopPane**. This allows buttons to be placed in the window. The **IconPane** provides functionality for the disk buttons.

The directory and file listings shown in the **ListPanes** on the right side of the window behave identically to the Smalltalk **DiskBrowser** window. Large portions of the methods that implement the functionality are borrowed from the **DiskBrowser** class.

Two of the most important methods for the DiskServer window are *retrieveObject* and *saveInstance*. The methods are simple but important to understand since they partially define

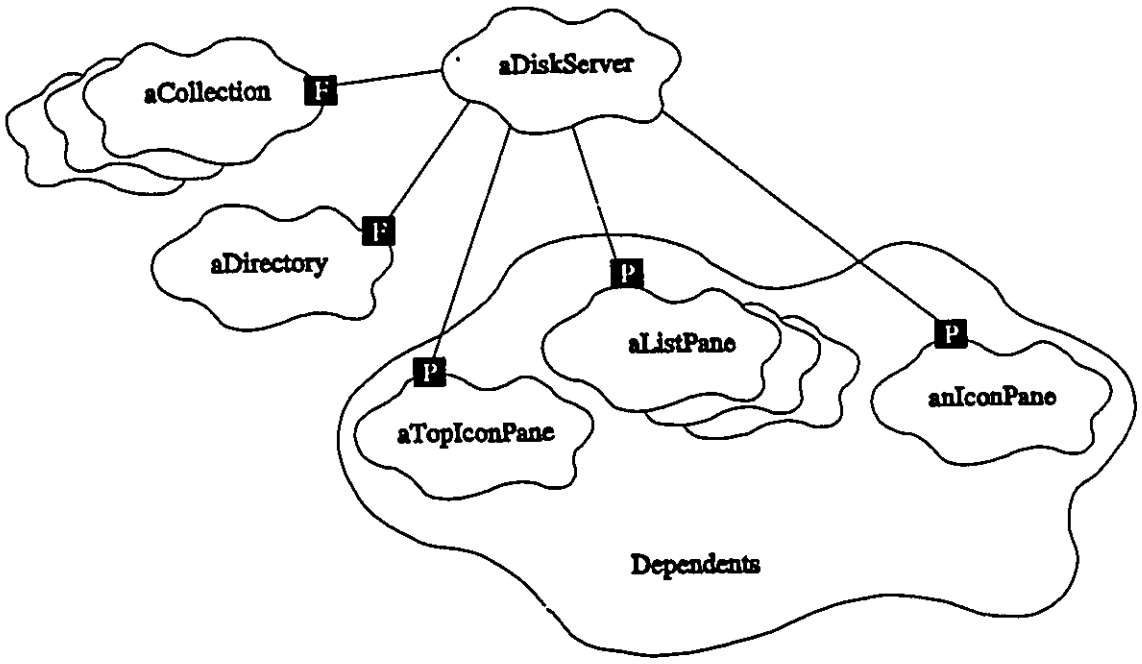

Figure 7.3 DiskServer Object Diagram

the framework for UniSet classes. Only UniSet instances that are specified by the cell operator for inclusion in a cell, such as FMC, WorkStation and UniLanguage instances, need to be saved to a file. Therefore only these classes need to implement these methods.

The code for *retrieveObject* is shown in Figure 7.4. It relies on the definition of a *retrieveFrom:* class method. This method must be defined in for each class of objects that are to be saved to a file. In the method shown lines four through six provide the user notification if a file has not been selected. Lines seven and eight instantiate a **Stream** on the file. Line nine locates the class of the object being retrieved. This information is always found at the very beginning of the file as shown in Figure 6.8. The class identified by *classString* is sent the message '*retrieveFrom:* fileStream' in line eleven. Line twelve updates the Instance Pane.

```
1   retrieveObject
2         "Retrieve an Object saved in the selectedFile."
3      | fileStream classString|
4   selectedFile isNil                    "Checks if file exists"
5      ifTrue:[Menu message: 'Select a file'.
6            ^nil].
7   fileStream := File pathName: selectedFile in:
8         selectedDirectory.              "Creates Stream on file"
9   classString := fileStream upTo: $:.    "Reads class name"
10  (Smalltalk at: classString asSymbol)   "Finds class in Smalltalk"
11  retrieveFrom: fileStream.             "Sends class message"
12  self updateInstances.                 "Updates window"
```

Figure 7.4 DiskServer *retrieveObject* Method

```
1   saveInstance
2         "Save the chosen instance."
3   selectedDirectory isNil                          "Check if directory selected"
4      ifTrue:[^Menu message:'Select a Directory first'].
5   selectedInstance isNil                           "Check if instance selected"
6      ifTrue:[^Menu message: 'Select an Instance first'].
7   selectedInstance saveTo: selectedDirectory.      "Save instance"
8   self changed: #fileList                          "Update window"
```

Figure 7.5 DiskServer *saveObject* Method

The *saveInstance* method shown in Figure 7.5 further defines the framework for UniSet classes. After ensuring that an instance to be saved and a directory in which to create the file has been indicated the *selectedInstance* is sent the instance method *saveTo:*. Each class must include *saveTo:* as an instance method. Thus, *saveTo:* must be part of the framework for all UniSet classes are to be saved to a file. Following the saving of the file, the window is updated.

The methods *retrieveFrom:* and *saveTo:* are not independent. The method *retrieveFrom:* must be able to decipher the format created by the *saveTo:* method. The **DiskServer** methods

retrieveObject and *saveInstance* work on all instances by relying on the methods shown and described in the following table.

Method	Class/Instance Method	Description
saveTo: aStream	Instance method	Record the instance to a Stream.
retrieveFrom: aStream	Class Method	Create an instance from a Stream

7.2 UniSet Browser

7.2.1 Functionality

The interface shown in Figure 7.6 functions as an object editor for the UniSet language. The window provides the interface in which to develop and edit relations that define the translations between UniSet and programmable machine languages. Two machine languages may be compared and edited side-by-side.

The window is divided into three basic parts. On the left is the UniSet Command List Pane for displaying commands. The remaining two parts of the window are identical to each other. These two parts allow two destination languages to be displayed in the same format simultaneously.

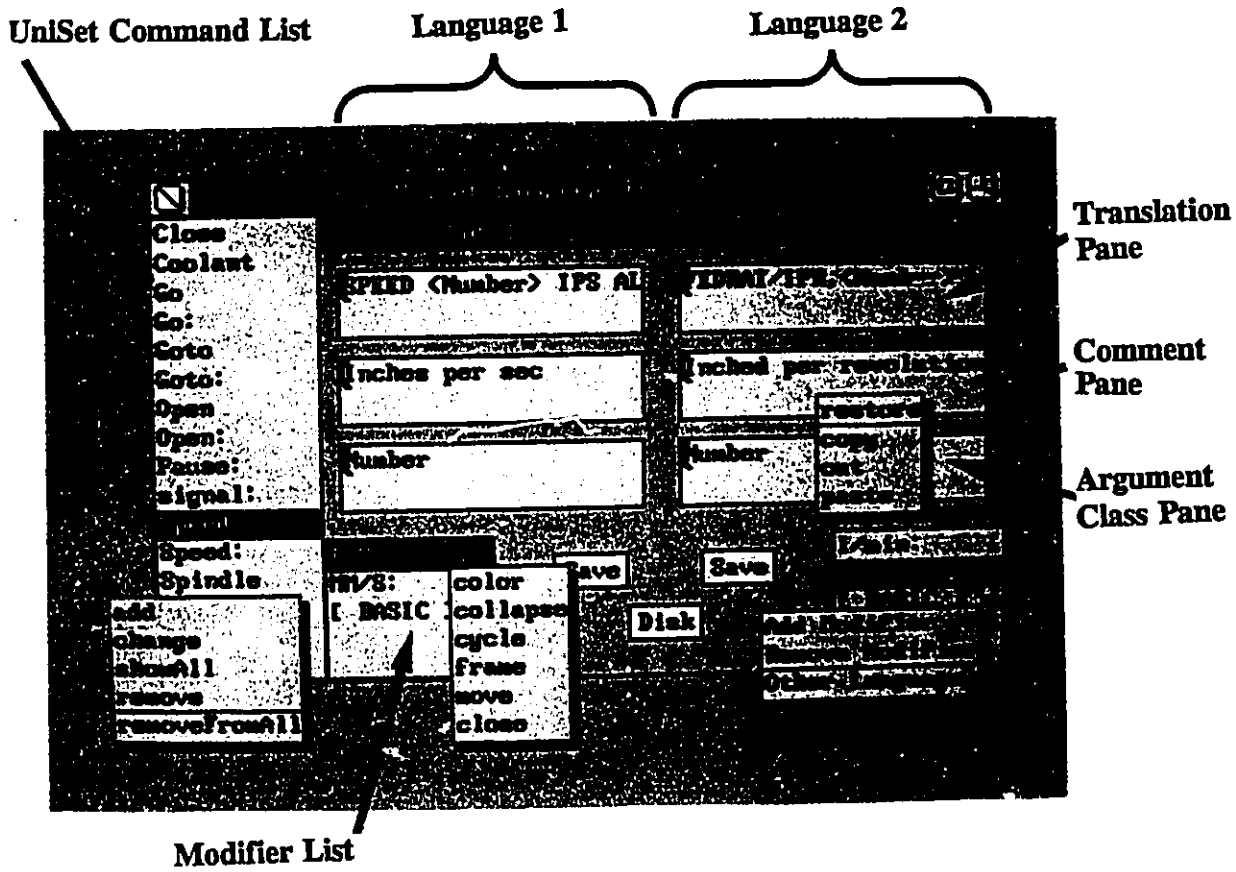

Figure 7.6 UniSet Browser Window

Upon opening the window all panes are blank except for the bottom two list panes. The two bottom ListPanes each show a list of languages for which UniSet translations are defined. The Language List Pane shown in the bottom right pane of Figure 7.6 displays such a list. Upon choosing a language in the Language List Pane the panes above are updated. The language name will be displayed on the black label immediately under the window label. The three panes directly below remain blank and the language list pane will also be blank. The UniSet Command List Pane on the far left will list all the commands defined for either of the chosen languages.

Making a selection in the command list pane will show the translations for the chosen UniSet command to the languages chosen. If the command does not have a translation to one of the displayed languages all the panes for that language will be left blank. In Figure 7.6 the 'Goto' command is highlighted and the APT translations are shown on the left half of the window. The 'Goto' command shown is a **UnaryModified** command. The modifiers for the command are shown in the Modifier List Pane at the bottom of the window. Selecting one of the modifiers will show the translation for the command and modifier combination. The modifier selected in Figure 7.6 is '*straight:*.'

The six evenly sized panes are instances of Smalltalk **TextPanes**. They provide text editing capabilities. The top of the three, the Translation Pane, shows the translation of the UniSet command. If an argument should be included with the command the string <*arg*> should be placed in the pane. If the translation is not direct but requires a Smalltalk method to be executed the pane will show a Smalltalk message preceded by the '#' character indicating the text is a symbol. The middle of the three panes displays comments. The bottom of the three text panes displays the class of arguments that may accompany the UniSet method. The classes must be separated by commas. Any changes to the text in any of the text panes must be followed by a click on the **Save** button to record the change to the Smalltalk image. The **Disk Server** window described in section 7.1 may be accessed at any time by clicking the **Disk** button at the bottom middle of the window.

Clicking the right mouse button anywhere on the window will raise a menu. The bottom **ListPanes** have two menus associated with them. The first menu allows language additions and

deletions and is only accessible when the pane shows a language list. The second menu is accessible only when the pane shows a modifier list. It allows addition and deletion of modifiers. Also on this menu is the choice *Other Language*. This selection provides the ability to redisplay the list of languages.

The menu for the six editing panes provide common Smalltalk editing options. The *restore* options allows the contents of the pane to return to the last saved state.

The Command List Pane menu has five choices. The *add* and *remove* selections modify the command list for the displayed languages. The *change* selection allows editing of the UniSet command without needing to remove and add the changed command and reenter all the translation information. The *removeFromAll* selection removes the selected command from every language defined within the UniSet environment and should be used very carefully. The *showAll* selection expands the command list to include all UniSet commands, not just the commands for the languages currently being edited. Those commands that do not have translations for the languages currently displayed are shown indented two spaces from the left side of the pane and bracketed by square brackets. The Command List Pane menu is context sensitive. *LanguageSet* replaces the *ShowAll* option on the menu, which allows the list to be restored to commands for the languages displayed in the window.

7.2.2 UniSet Browser Implementation

The UniSet Browser window combines two UniSet instance editors incorporated into the same window. Each instance editor is implemented in the **UniSetBrowserSide** class. The object diagram for a **UniSetBrowser** is shown in Figure 7.7.

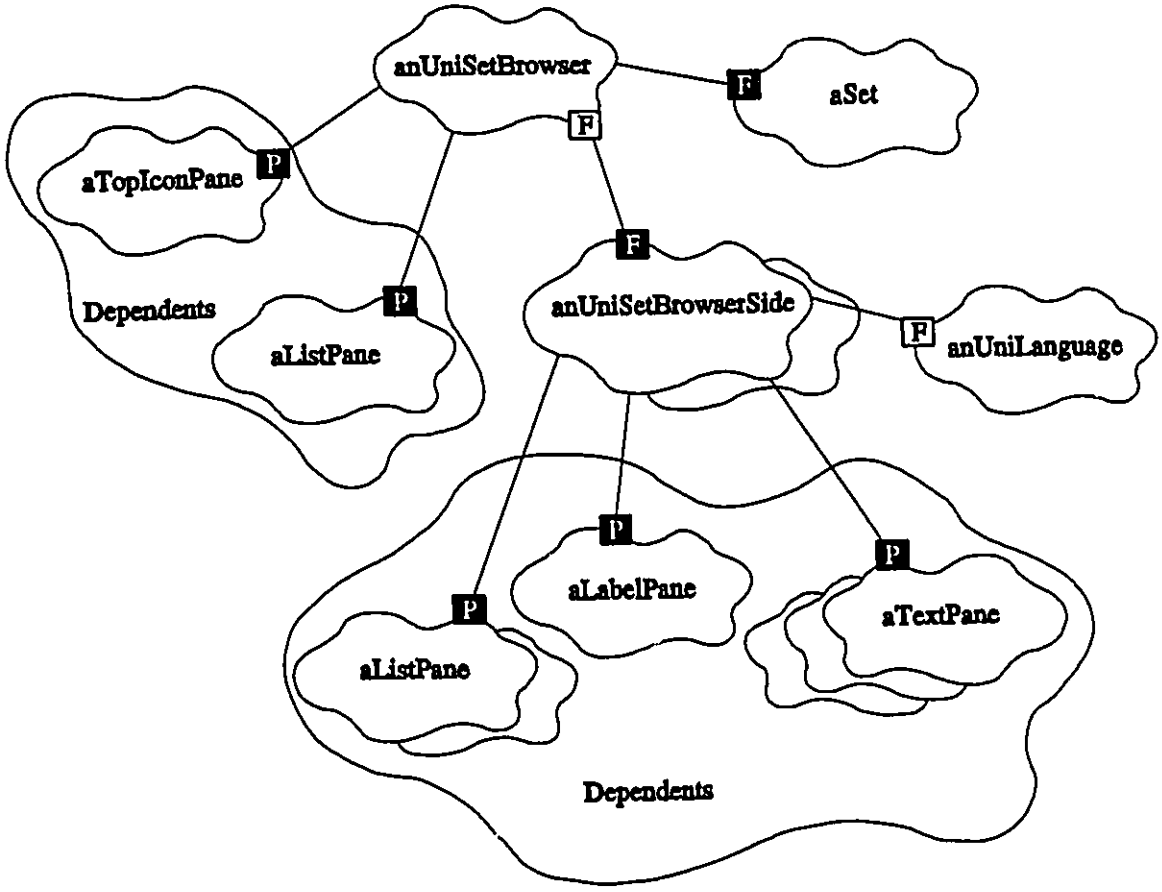

Figure 7.7 UniSet Browser Object Diagram

Like the Disk Browser, the UniSet Browser uses class **TopIconPane** instead of the **TopPane** class supplied with Smalltalk. Each **UniSetBrowserSide** includes two **ListPanes**, three **TextPanes** and one **LabelPane** as dependents. They are dependents because this allows

changes to the model, the language being edited, to be broadcast to the dependents and updated in the other panes. Class **LabelPane** is used to display the language names. It is not an original Smalltalk class and is more fully described in the appendix B.3.

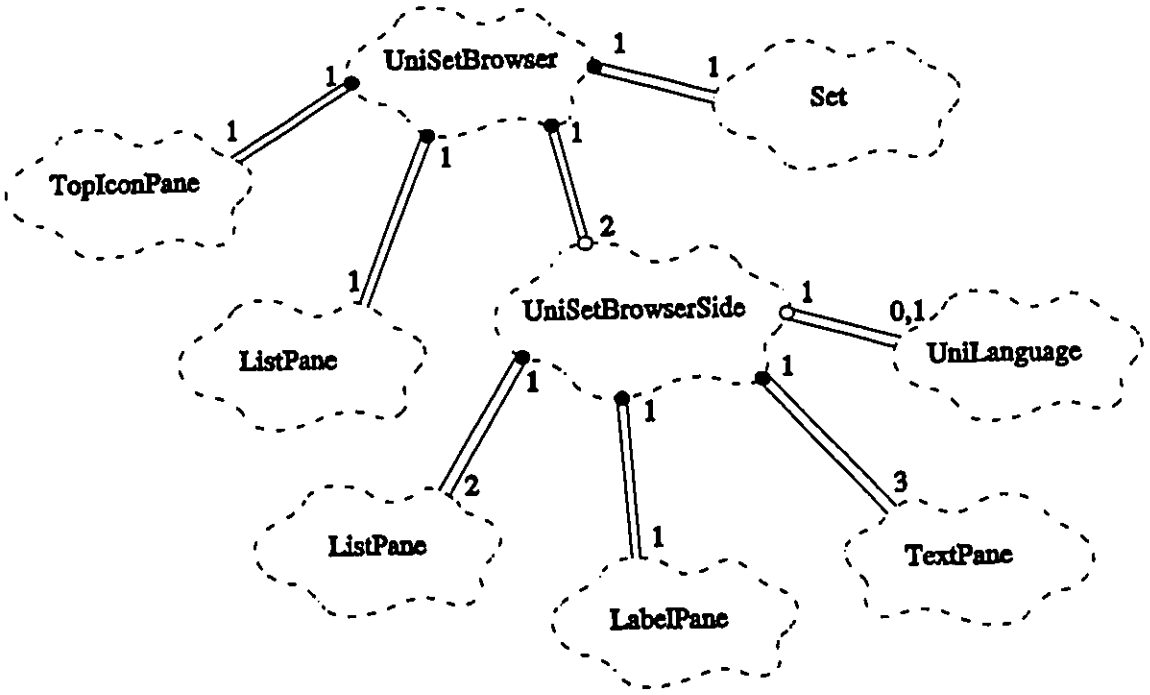

Figure 7.8 UniSetBrowser Class Diagram

The **UniSetBrowser** class diagram is shown in Figure 7.8. Class **UniSetBrowser's** interface uses the windows classes. The **Set** subclasses are part of the class' implementation and are used for containing lists of commands and languages. The commands are compiled from each **UniLanguage**, the languages from UniSet itself. The **UniLanguage** class also has an interface relationship with **UniSetBrowser** class. It is not part of the inner workings of the **UniSetBrowser** class and must be visible to other classes. The **UniSetBrowser** interface is

visible to the **UniSetBrowserSide** class because events in one side of the window can result in changes to the whole window.

7.3 UniSet Editor

7.3.1 Functionality

The UniSet Editor window shown in Figure 7.9 is the interface for creating UniSet cell programs. The Editing Pane is a **TextPane** for the creation of cell programs. The Machine Icon Pane in the top right corner displays an icon for each machine in the cell. Selection of an icon with the mouse causes the name of the machine to be displayed in the **LabelPane** immediately beneath the **IconPane**. The UniSet commands for the selected machine are displayed in the Command List Pane below both these panes. When a command is selected the Modifier List Pane displays the command modifiers.

Choosing an icon or any of the selections from the list panes with the left mouse button will add the selection to the text pane at the current **I-Beam** position. The icons are representations for machines defined in the cell. Making selections with the middle mouse button does not post the selection to the editing pane. This serves as a method to preview selections and will not send any information to the text pane.

The Icon Pane menu provides selections to open other windows. The Editing Pane menu allows programs to be *saved* to or *restored* from Smalltalk memory. The third selection, *translate* initiates the UniSet environment to start translation of the displayed program. There

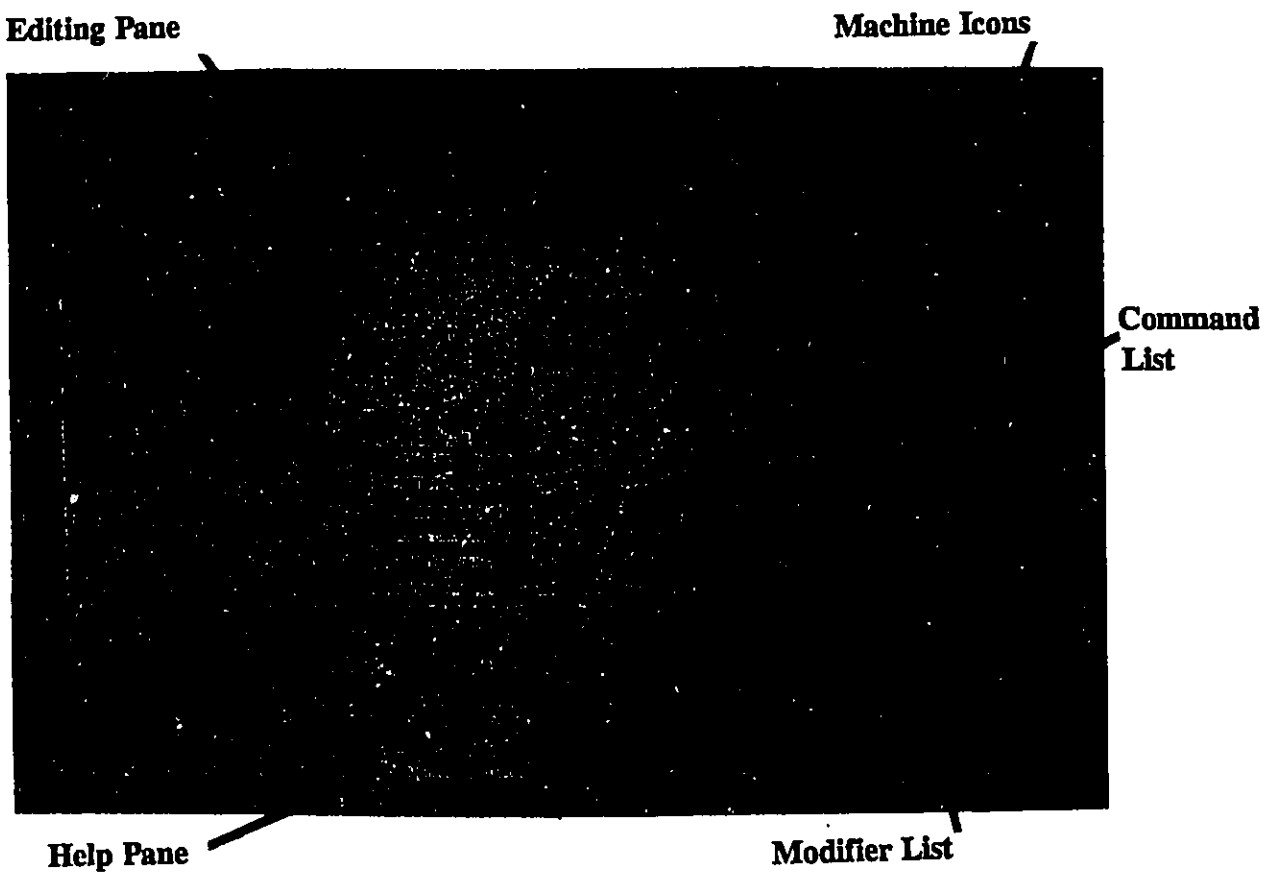

Figure 7.9 UniSet Editor Window

is no Command or Modifier List Pane menu. Clicking the right button in these panes will open a Help Pane at the bottom of the window. The Help Pane can be closed by moving the cursor over the pane. The comments can not be edited in this pane, only in the UniSet Language Builder window described in section 7.2.

7.3.2 UniSet Editor Implementation

The UniSet Editor functionality is encapsulated in the **UniEditor** class. A graphical description of the **UniEditor** class is shown in Figure 7.10. The implementation of the window is through a combination of **Pane** subclasses provided with Smalltalk and those developed with the application. **TopPane** is a basic Smalltalk class, it references the interfaces of the original Smalltalk classes of **TextPane** and **FieldPane**. Class **TopPane** also references the interface of **LabelPane, IconPane** and **GreyListPane** classes.

A **GreyListPanes** displays selections in grey rather than black as in the native Smalltalk class **ListPane**. More importantly the pane allows three mouse buttons to be used with in the pane. The left mouse button is used for selection and in these instances sends the selection to the Editing Pane. Selection from the list with the middle mouse button allows previewing of the command by updating the Modifier List pane and Help Pane without sending the selection to the Editing Pane. The right button still pops up the pane menu. A more complete description of the **GreyListPane** class and its implementation is presented in B.4.

The icons displayed in the **IconPane** are representations for machines defined in the cell. The icons displayed are defined by each **WorkStation** instance. Assignment of the icons to a machine is explained in section 6.7.

The **UniEditor** class uses the interface of the **UniTranslator** class to launch the translation of the created UniSet program. **UniTranslator** is not part of the implementation of the **UniEditor** class, only instantiated by the **UniEditor** class.

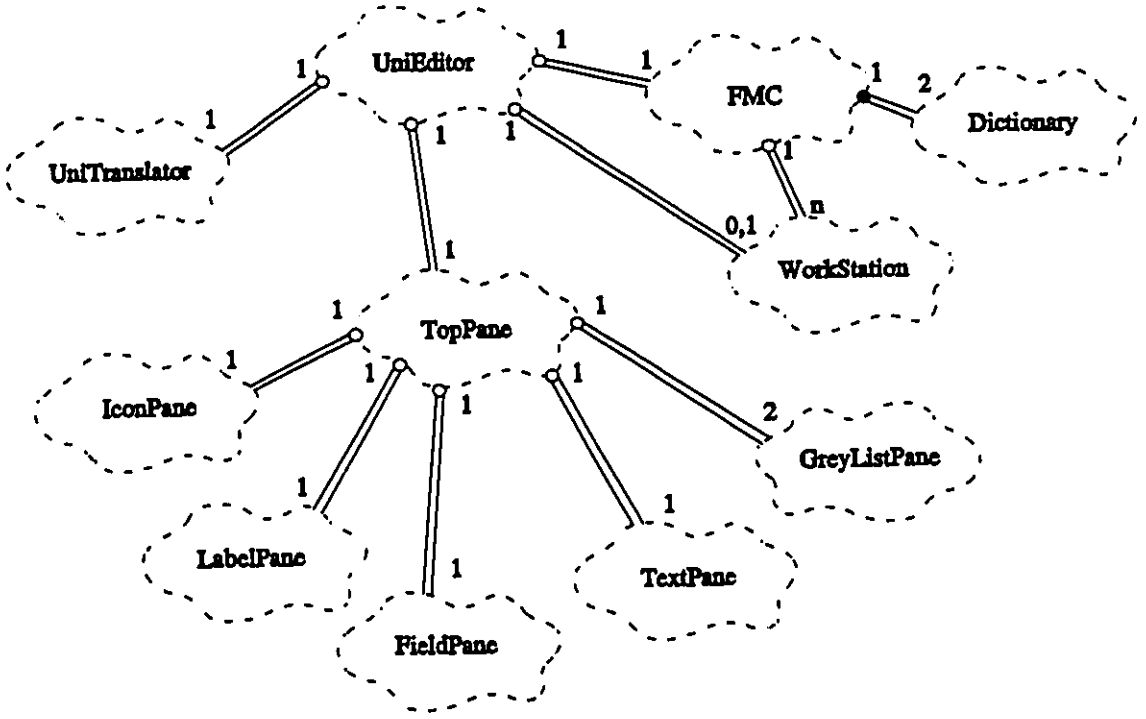

Figure 7.10 UniEditor Class Diagram

The **FMC**, and **Machine** class interfaces are also used by the **UniEditor** class. The **FMC** class must be visible to the **UniEditor** class since the contents of the window depend on the FMC instance. Each **UniEditor** window is instantiated and assigned an **FMC** during the *new:* message execution. Therefore the cardinality of this relationship is 1:1. An **FMC** may contain many number of machines and therefore there is a 1:n relationship between class **FMC** and class **WorkStation.** The **Machine** class is visible to the **UniEditor** class by requesting a machine through the interface of the FMC class. At any one time only one machine is visible to the **UniEditor** class.

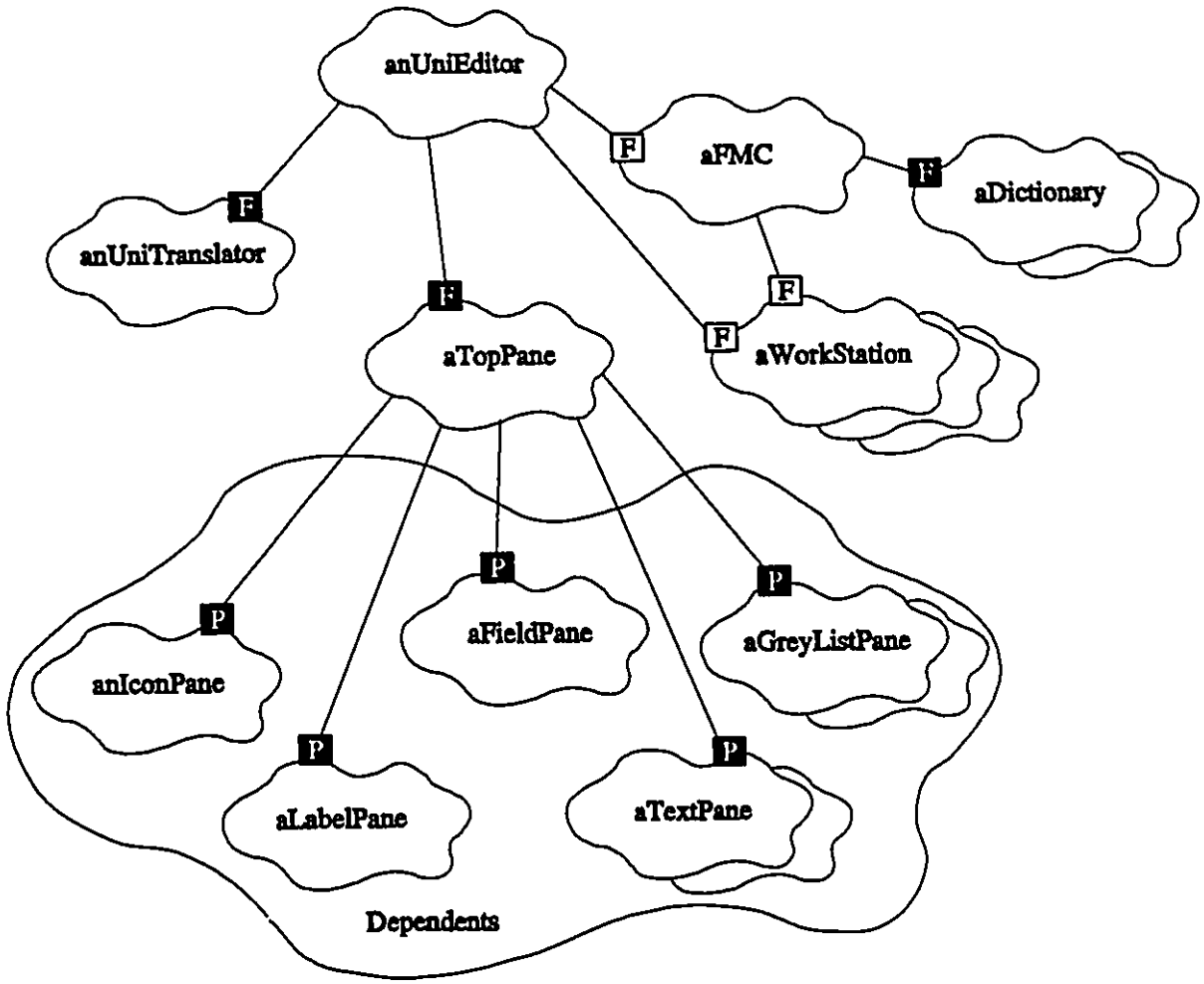

Figure 7.11 UniEditor Object Diagram

The **UniEditor** object diagram is shown in Figure 7.11. **UniTranslator, TopPane, FMC** and **Machine** instances are all fields of a **UniEditor**. The **FMC** and **Machine** instances are shared fields. The **LabelPane, IconPane, TextPanes** and **GreyListPanes** are dependents of the **TopPane** and are not directly visible to the **UniEditor**.

7.4 Concurrency and Synchronization Class Design

This section details a first attempt at designing classes to deal with machine interaction. The classes are not fully designed. They will need to be further developed upon the implementation of a cell.

The concept of Flexible Manufacturing Cells is based on the idea that individual machines could work together to perform a singular goal. In reaching the goal it is necessary that the individual machines must interact and cooperate. A robot loading a machine tool would be a common interaction. The action of a robot entering into the working volume of a machine tool has many potential hazards. To control the hazards the cell programmer must have at his disposal tools to start and stop operation of each machine.

Computer science treats the interaction of independently operating processes as a problem of concurrency. The solution to the interaction of machines relies heavily on the tools already developed for similar problems in concurrency.

Programming the interaction between machines requires the cell programmer to be able to easily shift between programming different machines. Signals are required to allow one machine to wait for another machine to finish execution before proceeding. The UniSet instructions provided for machine synchronization are *wait:* <aMachine> and *signal:* <*aMachine*>. The *wait:* instruction means the receiver must wait to receive a signal from the machine specified before executing the next instruction sequence. The *signal:* instruction sets

a signal line. A signal instruction should be issued to a waiting machine after a critical region of code is executed.

7.4.1 Synchronization Mechanisms

Much research and effort by computer scientists has centred on multi-tasking and process control. The problem of machine synchronization in the cell is an example of process control.

Semaphores are common tools for process control. A semaphore is used when two or more independent processes must access a shared resource. A semaphore controls access to a resource and ensures that only one process accesses a resource at any instance. Prior to accessing a shared resource a *wait* must be sent to a semaphore. If a resource is not being accessed a process can immediately access a resource. Upon completion of a task involving a shared resource the semaphore must be signalled to indicate the resource is free. A semaphore controls access to a resource by keeping count of the number of wait and signals it receives. If a process issues a *wait* while another process is accessing the shared resource the process must wait and its execution is blocked by the semaphore. As soon as the semaphore receives the signal indicating the resource is free it frees the longest waiting process to allow it access to the resource. Fortunately Smalltalk supplies a **Semaphore** class with all the necessary abstractions.

An example of a resource shared by two processes is a Robot that services a Lathe and a Mill. Rather than attempt to schedule the exact sequence of workpiece and tool changes for the Lathe and Mill it is simpler to have both machines request tool and workpiece changing. Prior to a request for the Robot the Mill and Lathe must access the semaphore. If the Mill

needs a tool change and the Robot is idle the semaphore will let the Mill request a the Robot to perform a tool change. If during the Mill's tool change the Lathe requests a work piece change the semaphore will know the Robot is busy and block further action by the Lathe. The Lathe will remain blocked until the tool change is completed on the Mill and the Mill has signalled the semaphore that it is finished with the Robot. The Lathe will then be released by the semaphore and given access to request the Robot to perform a work piece change.

7.5 Concurrency Class Design

Cell operation involves the functioning of independent machines. The machines though working independently must also be able to cooperate and synchronize to perform carefully orchestrated tasks. Much careful design and programming is needed to ensure efficient use of resources and to prevent common multi-process problems such as deadlock, interference or starvation. The classes **MachineDriver**, **UniSetController** and **SignalCommunicator** are required to ensure smooth operation during UniSet cell program execution.

7.5.1 MachineDriver

The **MachineDriver** class is intended to provide a software interface to each physical machine during UniSet program execution. **MachineDriver** instances should include all code necessary for communication and control with the associated machine. To send a signal to a machine would only require that the signal be sent to the associated **MachineDriver** instance.

The actual transmission of the message to the machine would be the responsibility of a **MachineDriver**. This abstract software representation of each machine would allow all machines to respond to the same protocol, thus the UniSet interface with each machine would be identical.

7.5.2 UniSetController

Class **UniSetController** is intended to provide overall supervision of UniSet program execution. During cell operation a **UniSetController** instance will follow the cell events, it will do this by executing the cell control program that is a result of UniSet program translation. All communication between cell constituents must pass through the UniSetController, responses to signals are determined by the cell control program. During cell operation machines will not send information directly to each other or directly to the **MachineDrivers** rather all communication must be sent through the UniSetController. Upon receipt of a signal the UniSetController will take the appropriate action, possibly informing a MachineDriver of some event.

7.5.3 SignalCommunicator

The class **SignalCommunicator** has been implemented to allow signals to be passed through it to a **MachineDriver** from the **UniSetController**. These signals may originate anywhere but will typically originate at other machines.

The **MachineDriver** is a separate process from the **UniSetController** and has its own tasks, mainly to supervise the machine communication. As the **MachineDriver** is working it may require a signal from the **UniSetController**. The synchronization required to pass the messages would require the **MachineDriver** and **UniSetController** processes to wait for each other. This is particularly undesirable for the **UniSetController** since it coordinates all the machines.

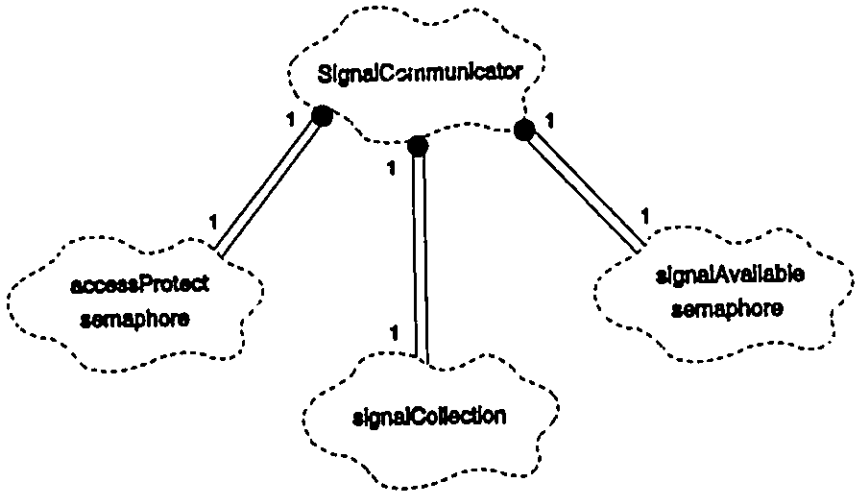

Figure 7.12 SignalCommunicator Class Diagram

SignalCommunicator class functions as a queue and allows the **UniSetController** to pass a signal to a **MachineDriver** without waiting for it to respond. **SignalCommunicator** receives messages and places them in a buffer contained in the instance variable *signalCollection*. Signals are kept in the buffer until the **MachineDriver** requests them. The **SignalCommunicator** must also ensure mutual exclusion to the buffer, both the **MachineDriver** and **UniSetController** cannot access the buffer at the same time without corrupting it. The semaphore *accessProtect* ensures mutual exclusion.

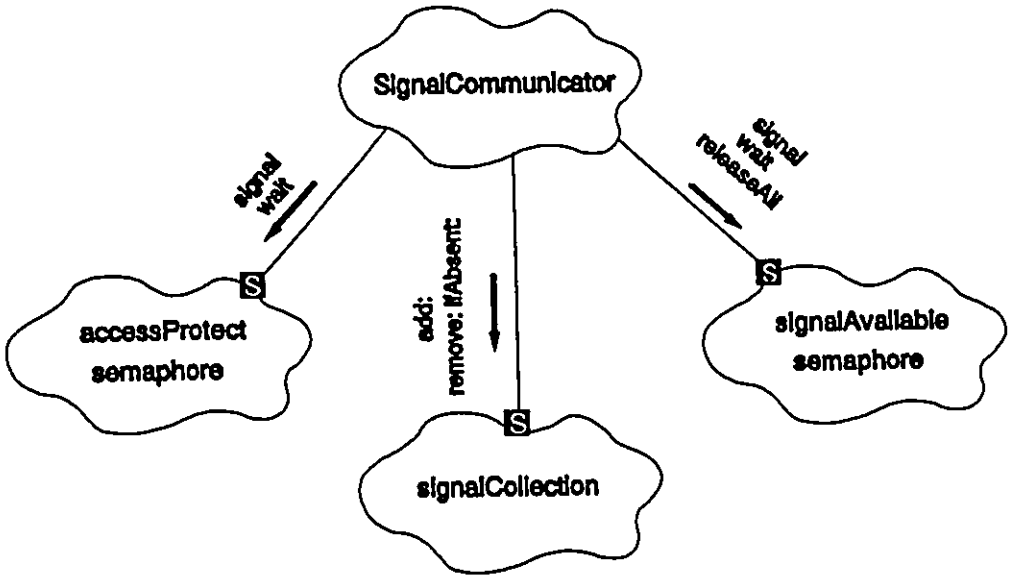

Figure 7.13 SignalCommunicator Object Diagram

If a machine is expecting a signal from another machine the signal may not come at the required time. The buffer ensures that signals that arrive too early are preserved for the **MachineDriver**. If the signal is not in the signal collection the **MachineDriver** must wait for the signal before proceeding. A second semaphore, *signalAvailable*, prevents a **MachineDriver** from accessing the buffer when it is empty, rather a **MachineDriver** waits for an incoming signal. Waiting for a signal is preferrable to polling the buffer since a process that involves polling still requires execution time on the computer processor, delaying other active processes. The signalAvailable semaphore suspends the **MachineDriver** process, releasing the process only when the correct signal arrives. Processor time is thus preserved for other more productive events.

The methods implemented in **SignalCommunicator** class are few. The methods include two interface methods and two private methods. The interface messages are *signal:* and

waitingFor:. The message *signal:* delivers a signal to the **SignalCommunicator** and places it in the *signalCollection*. Message *waitingFor:* tells the **SignalCommunicator** that the sender is waiting for the signal specified as the argument. If the signal is in the *signalCollection* it is immediately returned. If the signal has not already been added to the *signalCollection* the sending process is sent to wait at the *signalAvailable* semaphore.

Two private methods *initialize* and *searchFor:* are also implemented in **SignalCommunicator** class. *Initialize* is typical of all the many other initialize methods. It assigns the proper objects to the instance variables upon creation of the object. The method *searchFor:* searches the signalCollection for the desired message and returns the signal or 'nil'.

Method *searchFor:* at this time does not include much functionality, it has been provided so that it can be easily modified without modifying the other methods. Currently the signals are all kept in a collection but later it might be beneficial to keep them in a Dictionary, the searchFor: method would need alteration.

CHAPTER 8

CONCLUSIONS AND RECOMMENDATIONS

8.1 Summary

This thesis was undertaken with the intention of reducing programming and configuration efforts for flexible manufacturing cells. This has become necessary because of the proliferation of programming languages that have been developed for use with computer controlled machines and for use with FMCs and the lack of standard communication protocols between these machines.

The beginning of automation research began in 1952 with the creation of computer controlled machines. The most recent research on manufacturing automation involves the integration of multiple machines into flexible manufacturing cells. The cell is a desirable configuration of machines if it is flexible. Flexible manufacturing cells can be flexible in two ways, either by variations of a part with a single program or by being easily reprogrammable and reconfigurable.

The successful implementation of an FMC can yield benefits that include increased manufacturing time without operator supervision, reducing costs and increasing productivity. Ease of configuration also reduces cell downtime.

The current state of technology in manufacturing machines creates much difficulty for integrating machines into a cell. Machines from different manufacturers do not often use the same programming language. The cell developer must then program each machine in a unique language. Without common programming languages a cell controller cannot communicate with each machine, thus each machine must be programmed at its own controller. A cell is based on the cooperation of several machines into a single working unit. Cooperation is achieved through communication. Communication between machines in a cell is a major source of difficulty since there is a lack of common communication protocol.

In hopes of easing the implementation of a cell the major goal of this thesis was to develop a simple set of instructions that cater to a variety of manufacturing machines. The instructions were derived from a thorough study of the most prominent languages currently implemented on a variety of computer controlled machines. Implementation of the resulting instruction set, labelled UniSet, included the development of a programming environment. Presented in this thesis is a comparison of the programming languages studied and the resulting instruction set. Also detailed is the UniSet environment and its graphical user interface including an explanation of the software design.

Included as appendix A is a brief explanation of object oriented terminology and design notation. This appendix was included because the development of UniSet and the environment rely heavily on object oriented programming and its paradigms.

8.2 Conclusions

The major purpose for undertaking this thesis was to find substantial similarities between manufacturing programming languages and relate common functions from various machines. A study was undertaken to document the similarities between several programming language. The study, which is documented in chapter 4, concluded that there is significant common functions presented in varying programming languages. By comparing the common functions it was possible to create a set of instructions that could represent the various functions common to each language. A Unified instruction Set (UniSet) was generated.

The production of a programming environment that implemented and supported UniSet was also a major goal of this thesis. The creation of the programming environment evolved out of the Smalltalk programming environment and language from Digitalk. The selection of Smalltalk was the result of a search that demonstrated that an object oriented language could provide the best support for the development of the UniSet environment. The suitability of an object-oriented programming language results from the object-oriented chacteristics of data encapsulation and abstraction. These characteristics provide the means for each machine to be represented in the same manner, with the same protocol irrespective of the individual machine characteristics. Smalltalk, of all the object-oriented languages evaluated, provided the largest set of built-in features to support the development of the UniSet environment.

One of the major reasons for selecting an object oriented language was that the theory of using a common protocol for every manufacturing machine paralleled the object oriented paradigm perfectly. To the cell programmer a robot and a milling machine should have the

same programming interface and similar instruction sets. Thus the implementation of UniSet developed naturally into an object oriented environment for manufacturing programming.

The UniSet environment includes tools for cell configuration and programming. The environment is designed as a graphical interface employing icons and providing for mouse interaction. The **UniEditor** user interface was developed for cell programming. To configure the UniSet instruction set the **UniSet Browser** was created. The **DiskServer** interface allows language, cell and machine descriptions to be saved and retrieved from a disk.

A major feature of the UniSet instruction set and environment is its adaptability. It has been constructed with an open architecture to allow the inclusion of additional machine and language definitions. New features can be added to the environment without impacting existing features.

8.3 Recommendations

The development of UniSet and the UniSet environment has established the foundation for a complete cell programming and control environment. The research presented in this thesis should be expanded to include powerful tools to simplify programming and enhance cell control. The object oriented nature of UniSet allows tools to be developed generically and find application with any machine.

There are many more developments that can be added to enhance UniSet. Communication modules should be included in further developments. These modules would be objects that handle communication with each machine. Each instance of a communication module would serve one machine. If the communication classes were properly designed, according to object oriented principles, the communication protocol would be identical for every machine.

A fully developed UniSet environment would rely heavily on artificial intelligence for the implementation of error checking, simulation and collision avoidance. Error checking could be as simple as program integrity or as complicated as checking machine paths to prevent conflicts.

REFERENCES

1. D.F. Johnston, "Small Batch, Precision Manufacturing Using a Prototype Automated Cell", Reprinted from *Proceedings of the 2nd International Conference, Computer-aided Production Engineering*, Edinburgh, Scotland, April 13-15, 1987, NRCC No. 27454, pp 145-153.

2. Daniel F. Johnston, "Using the VMS Operating System and a VAX Computer to Control a Flexible Manufacturing System", National Research Council of Canada, NRCC No. 32104.

3. Roger Seifried, "Unattended Machining", Technical Paper presented International SME Conference, May 6-9, 1985, Detroit, MI, Society of Manufacturing Engineers, Dearborn MI, Paper No. MR85-468.

4. J. Temmes, P.Ruusunen and J. Lempiainen, "Assembly Cell Controls Using Flexible Software Tools and High-Level Control Configuration Language", *Control and Programming in Advanced Manufacturing*, K. Rathmill Ed., IFS (Publications) Ltd., 1988, pp 411-424.

5. P.K. Wright, D.A. Bourne, et al., "A Flexible Manufacturing Cell for Swaging" *Mechanical Engineering*, Oct. 1982. pp 76-83.

6. Yoram Koren, *Computer Control of Manufacturing Systems*. McGraw-Hill, Inc, 1983. pp 253-255, 1-2.

7. D. Kochan (Ed), *CAM: Developments in Computer-Integrated Manufacturing*, Springer-Verlag, Berlin 1986. pp 31.

8. Atef Fahim & Rudy Tolkamp, *UniSet - A Flexible Manufacturing Cell Programming, Simulation, and Management Environment*. INCOM '92 Preprints, pp 273-277.

9. Atef Fahim & Kyunghyun Choi, *Real Time Cell Control for Flexible Manufacturing*. 2nd IFAC WorkShop on Architectures for Real Time Control, Seoul, Korea, Aug. 1992.

10. R.H. Taylor, P.D. Summers and J.M. Meyer, "AML: A Manufacturing Language", *Control and Programming in Advanced Manufacturing*, K. Rathmill Ed., Springer-Verlag, Berlin, 1988, pp. 183-214.

11. David D. Grossman, "A Decade of Automation Research at IBM", Reprint from Robots VI Conference Proceedings, March 2-4 1982, Detroit MI, SME Technical Paper No. MS82-224.

12. David D. Grossman, "AML as a Plant Floor Language", *Robotics & Computer-Integrated Manufacturing*, 1985, Vol. 2, No.3/4, pp 215-217.

13. David D. Grossman, William M. Short, "AML - Much More Than a Robot Language", Technical Paper presented at Robots 9 Conference, June 2-6 1985, SME Technical Paper No. MS85-612.

14. Andre Sharon, Edward Carey, "AML + PC = CIM?", *Computers in Mechanical Engineering*, July 1985, pp. 30-34.

15. P.K. Wright, D.A. Bourne, J.P. Colyer, G.S. Schatz and J.A.E. Isasi, "A Flexible Manufacturing Cell for Swaging", *Mechanical Engineering*, October 1982, pp 76-83.

16. David Alan Bourne and Paul Fussell, "Designing Programming Languages for Manufacturing Cells", Robotics Institute, Carnegie-Mellon University, Pittsburgh, Pa, 1982.

17. Paul Kenneth Wright and David Alan Bourne, Manufacturing Intelligence, Addison-Wesley Publishing Company Inc., Reading, Massachusetts, 1988, pp 28 & 30.

18. David Alan Bourne, "A Numberless, Tensed Language for Action Oriented Tasks", Robotics Institute, Carnegie-Mellon University, Pittsburgh, Pa, 1982.

19. David Alan Bourne, "CML: A Meta-Interpreter for Manufacturing", *AI Magazine*, Fall 1986, Vol. 7, No. 4, pp 86-96.

20. David A. Bourne and Mark S. Fox, "Autonomous Manufacturing: Automating the Job-Shop", *Computer*, Sept. 1984, pp 76-86.

21. David Bourne, "A Multi-Lingual Database Bridges Communication Gap in Manufacturing", *Robotics and Factories of the Future*, Springer-Verlag, Berlin 1984, pp 545-552.

22. The APT Long Range Program Staff, IIT Research Institute, APT Part Programming, McGraw-Hill Book Co. New York, NY, 1987, preface.

23. Brian O. Wood and Mark A. Fugelso, "MCL, The Manufacturing Control Language", Presnted at SME 13th ISIR/Robots 7 Conference, April 17-21, 1983, Chicago, Ill.

24. A.P. Ambler, "RAPT: An Object Level Robot Programming Language", Presented at *Colloquium of Languages for Industrial Robots*, Institute of Electrical Engineers, London, Feb. 1982, D.A.I. Paper No. 172.

25. A.P. Ambler, R.J. Popplestone and K.G. Kempf, "An Experiment in the Offline Programming of Robots", D.A.I. Research Paper No. 70, University of Edinburgh, Edinburgh, Scotland, June 1982.

26. Jack Hollingum, "Working Towards a Common Language", *The FMS Magazine*, April 1983, pp 178-180.

27. Standards Council of Canada, Announcement of Meeting of ISO/TC 184/SC3/WG2.

28. Susan Bonner and Kang G. Shin, "A Comparative Study of Robot Languages", *IEEE Computer*, December 1982, Vol. 15, No. 12, pp 82-96.

29. Willian A. Gruver, Barry I. Soroka, John J. Craig and Timothy L. Turner, "Industrial Robot Programming Langauges: A Comparative Evaluation", Tutorial on Robotics 2nd ed., IEEE Computer Society, 1986, pp 560-565.

30. Tutorial on Robotics, 2nd ed., IEEE Computer Society, 1986, pp 447-454.

31. K.S. Fu, G.C. Gonzalez and C.S.G. Lee, Robotics: Control, Sensing, Vision and Intelligence, McGraw-Hill Book Co., New York, New York, pp 450-473

32. Tomas Lozano-Perez, "Robot Programming", Proceedings of the IEEE, Vol. 71, No. 7, July 1983, pp 821-841.

33. Shimano, Bruce E., Geschke, Clifford C. and Spalding, Charles H. III, "VAL-II: A New Robot Control System for Automatic Manufacturing", *Proc. of the International Conference on Robotics*, March 13-15, 1984, pp 278-292.

34. *User's Guide to VAL II*, Programming Manual, Ver. 1.4B, Unimation, May 1985.

35. *ASEA Programming Manual*, ASEA Robotics, June 1984.

36. Chang, Chao-Hwa and Melkanoff, Michel A., NC Machine Programming and Software Design, Prentice Hall, Englewood Cliffs, New Jersey, 1989, pp 22, 86.

37. David J. Larin, "Cell Control: What We Have, What We'll Need", *Manufacturing Engineering*, January 1989, pp 41-48.

38. M. Onori and M. Nystrom, "Adaptive Techniques for the Mark II Flexible Automatic Assembly System", in preprints of *7th IFAC Symposium on Information Control Problems in Manufacturing Technology*, INCOM '92, Toronto, Canada, May 1992, Vol. 1, pp 268-272.

39. Carole Cunningham, "On the Move to Off-line Programming", *Mechanical Engineering*, Oct. 1988, pp 77-79.

40. M.C. Bonney, K.Y. Young, Y.F. Yong, "Off-Line Programming Using the GRASP Robot Simulation System", in proceedings of *2nd International Conf. on Computer-aided Production Engineering*, J.A. McGeough Ed., Edinburgh, April 1987, pp 67-70.

41. E.J. Wright, "Some Experiences of Off-line Programming on ASEA Robots", in proceedings of *4th International Conf on Computer-Aided Production Engineering*, J.A. McGeough Ed., Edinburgh, Nov. 1988, pp 135-140.

42. Johnson, Ralph E. and Foote, Brian. "Designing Reusable Classes", Journal of Object Oriented Programming, June/July 1988, pp. 22-35.

43. Lalonde, Wilf and Pugh, John. "Pluggable Tiling Windows", Journal of Object Oriented Programming, September/October 1989, pp. 57-66.

44. *Smalltalk/V 286*, Tutorial and Programming Handbook, Digitalk, Inc. Los Angeles, CA, 1988.

45. Booch, Grady, *Object Oriented Design with Applications*, Benjamin/Cummings Pub. Co., Inc., Redwood City, CA, 1991.

46. Udo Graefe, Ajit Pardasani and Albert W. Chan, "Conceptual Architecture of an Object Library for Design, Control and Simulation of a Manufacturing Enterprise", in preprints of *7th IFAC Symposium on Information Control Problems in Manufacturing Technology*, INCOM '92, Toronto, Canada, May 1992, Vol. 1, pp 316-321.

47. ISO/TC 184/SC3/WG2 Minutes.

48. Prasun Dewan, "Object-Oriented Editor Generation", *Journal of Object Oriented Programming*, July/August 1990, pp 35-49.

49. Kurt Schmucker, "Using Objects to Package User Interface Functionality", *Journal of Object Oriented Programming*, April/May 1988, pp 40-45.

50. M. Fabian, B. Lennartson, "Control of Manufacturing Systems: An Object Oriented Approach," INCOM '92, Toronto, Canada, May 1992, Vol. 1, pp 47-52.

51. Gary T. Leavens, "Introduction to the Literature on Object-Oriented Design, Programming and Languages", ???, pp 40-53.

52. John H. Saunders, "A Survey of Object-Oriented Programming Languages", *Journal of Object Oriented Programming*, March/April 1989, pp 5-11.

53. Wilf Lalonde and John Pugh, "Designing is Hard: Object-Oriented Software is Different!", *Journal of Object Oriented Programming*, March/April 1989, pp 46-55.

54. Dave Thomas and Randolph Best, "Smalltalk + C, The Power of Two", Dr. Dobb's Journal, August 1989, pp 16-18,94-100.

55. Szukalo, Edward W. "A Unified Instruction Set for Flexible Manufacturing Cells", Fourth Year Thesis, University of Ottawa, Dec. 1990

56. Taylor, Christopher D.A., "Defining and Implementing a Unified Instruction Set for a Flexible Manufacturing Cell", Fourth Year Thesis, University of Ottawa, April 1989.

57. Quail, John P., "A Software System to Program an FMS Using a Unified Instruction Set", Fourth Year Thesis, University of Ottawa, Dec. 1988.

58. Lalonde, Wilf and Pugh, John R., *Inside Smalltalk*, Vol 1., Prentice Hall, Englewood Cliffs, N.J., 1990.

59. Goldberg, Adele and Robson, David. *Smalltalk-80 The Language*, Addison-Wesley Pub. Co., Reading, Mass., 1989.

APPENDIX A

OBJECT-ORIENTATION

This appendix is included for the benefit of those uninitiated to object-oriented programming (OOP). It explains object-oriented programming concepts with emphasis on their application to Smalltalk. This is followed by a description of the notation used in this thesis for communicating object-oriented designs.

A.1 Object-Oriented Terminology and Characteristics

The object-oriented vocabulary is not large but can be difficult to understand. The definition of each word depends on the definition of other words that also need to be defined within the object-oriented context. An understanding of the concepts of OOP requires an understanding of the individual terms and characteristics, however an understanding of these terms seems to require an understanding of the concept of OOP.

A.1.1 Terminology

Object/Instance

The software definition of *object* is more formal than the customary meaning of *object*. An object is tangible, visible or exists intellectually. An object can be a receiver of a thought or action. Examples of physical objects include a machine or a vise. Intellectual or intangible objects include a machining program or language translator.

According to Booch "*An object has state, behaviour and identity...* [45]". The state includes all of an objects 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. For example, a lathe may have properties that include 'bed_length' and 'tool_position'. 'Bed_length' is a static property because a lathe usually does not change the length of its bed. 'tool_position' is a dynamic property because a lathe's 'tool_position' changes frequently. The behaviour of an object characterizes how it acts and reacts when acted upon by other objects and by itself. The identity of an object is that which sets it apart from all other objects. The word *instance* is synonymous with object.

Examples of objects in the field of manufacturing are parts, machine tools and grippers. All these objects have their own state, behaviour and identity. For example the state of a part can be finished or unfinished. A machine tool can machine a slot as one of its behaviours. A part is identified by its serial number.

Within the context of OOP almost everything can be an object. Physical entities can be programmed as objects and objects can be created for performing specific tasks. Particularly important to the work in UniSet is the **UniParser**. The **UniParser** translates the UniSet code into native machine code. Although this seems like a function to the sequential programmer this too can be an object.

Class

The formal definition of a *class* according to Lalonde and Pugh is:

"*A description of a set of objects with similar characteristics, attributes and behaviours.* [58]"

A *class* is a generic description, the particulars of the objects it describes are not defined. A class describes the 'essence' of being a member of a set of similar objects. A class can best be compared to a data type in high level languages such as Pascal.

A simple example is a set of **Point** objects. Every **Point** object has an 'x' and 'y' coordinate. The 'x' and 'y' coordinates are all different but to be **Point** objects they must all have these two coordinates. Therefore, class **Point** would characterize all of its objects, or instances, as having an 'x' and 'y' coordinate. Each new point is declared to be an instance of class **Point** and its 'x' and 'y' values are assigned.

Message & Method

Lalonde states that a class contains objects with similar behaviours [58]. The behaviour of a class must be typical of all instances of that class and is described using the terms message and method. *Message* and *method* are usually considered interchangeable but there is a slight difference. A *message* is sent to an object, a *method* executes in response to a message.

With further reference to the class **Point** example. Class **Point** might include comparison methods. Given the existence of three instances of the **Point** class, 0,0 and 5,5 and 3,2, the following *messages* can be sent to the points.

3,2 above: 5,5 3,2 between: 0,0 and: 5,5

Class **Point** knows how to respond to the messages 'above:' and 'between:and:'. The statement on the left would return 'false' because the 'y' value of 3,2 is less than the 'y' value of 5,5. The example on the right would return 'true' since the 'x' and 'y' values of the point 3,2 are both greater than that of point of 0,0 and less than that of point 5,5. The methods of class **Point** include the code that implements the described behaviour.

Manufacturing Example

Using the field of manufacturing to provide an example of the above definitions should improve clarity. Class **Part** and class **Machine** are defined in Figure A.1. Class **Part** includes attributes and messages common to every part and class **Machine** includes attributes and messages common to every machine.

The attributes are labelled 'instance variables' to coincide with Smalltalk terminology.

The **Part** class instance variables are:

serial_No	number identifying the part.
customer	the customer name for whom the part is being machined.
program	the machining sequence for the part.
finished	a boolean value indicating whether the part has been fully machined.

The instance variables for the **Machine** class include:

name	identifies the machine.
tool_Position	contains the position of the tool that is in the machine.
speed	indicates the current speed of the tool.
part	contains the part currently being machined by the machine.

The messages that instances of each class will respond to are also part of the class description. The methods for the classes involved are explained below but code is not shown. An instance of class **Part** will respond to the following messages:

finished	a question returning 'true' or 'false' depending on whether the part has been completely machined.
program	returns the program that needs to be executed by the machine to produce the finished part.
customer	returns the customer's name who ordered the part.

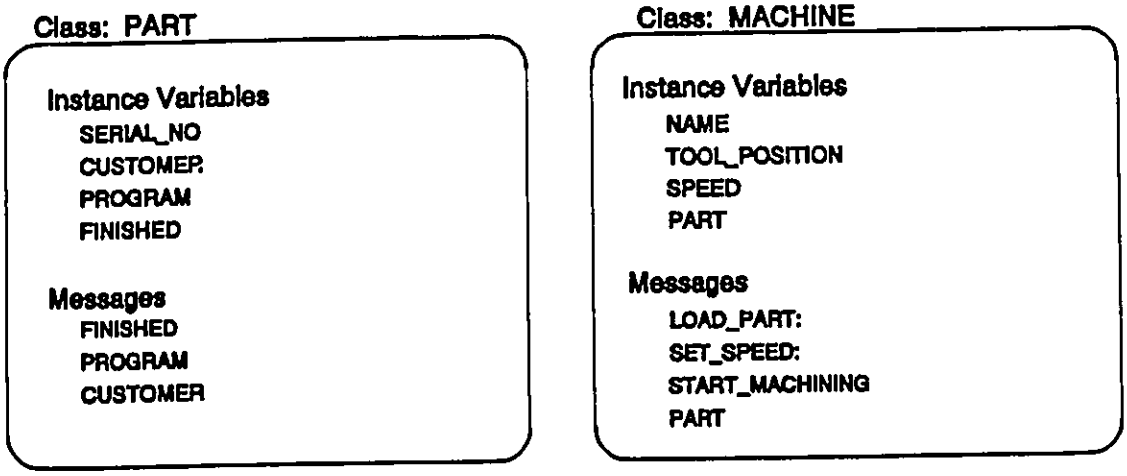

Figure A.1 Machining Example: Class Description

An instance of **Machine** class will respond to the messages:

load_part: accompanied by an argument indicating a part. Tells the machine which part it is holding by assigning to the instance variable 'part'.

setSpeed: accompanied by a number to set the speed of the machine. Assigns the number to the instance variable speed.

start_Machining initiates execution of the part program.

part returns the current part, the value of the instance variable 'part', being machined or nil if no part is in the machine.

So far in the example only classes have been described, no actual objects have been mentioned[1]. It is easy to imagine the instances of **Machine** and **Part** classes. Initially each instance is assumed to be initialized so that all the instance variables have appropriate values. The instance variable 'finished' for all parts would be 'false'. The 'part' on the machine would be 'nil' because a part has not yet been placed on the machine. The **Part** instances are given names 'part_1', 'part_2' etc and the **Machine** instance is named 'mill'.

The mill could be issued the following command:

mill *part:* part_1.

This would assign part_1 to the instance variable 'part' of the mill, the machine could then send to part_1:

part_1 *program.*

This would retrieve the machining program for the part, the mill could then *start_Machining* by following the program for part_1.

A.1.2 Fundamental Characteristics of Object-Oriented Programming

For a language to be considered fully object-oriented it must include four characteristics: inheritance, polymorphism, data abstraction and encapsulation. Languages

[1] In reality classes are instances of class **Class**. This detail, however, is best ignored for clarity of an introduction of the subject.

that contain less than all four characteristics, or partially support the characteristics, can be classified as object-oriented but usually with some qualifications.

Inheritance

Inheritance between humans is easily understood. Children routinely inherit physical features, personality traits and possessions from parents and grandparents. Object-Oriented languages contain a mechanism that allows inheritance between classes. Inheritance involves a *superclass*[2] and *subclass(es)*. Subclass(es) inherit the superclass' behaviour and structure. There may be any number of subclasses for one superclass. Inheritance may be multi-tiered, that is a subclass may itself have a subclass(es).

Object-oriented *inheritance* is a mechanism that allows a superclass to share its methods and representation with other classes. In Smalltalk all instance variables, class variables, pool dictionaries and methods are inherited by subclasses. The benefits of inheritance are many. 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 behaviour of the superclass are passed to subclass(es). Common behaviours and attributes should be coded into the superclass from which related classes can inherit. Inheritance acts as an

---

[2] Single inheritance allows a subclass to have only one superclass. Multiple inheritance is an unusual feature that allows a class to inherit from more than one superclass. This thesis will only be concerned with single inheritance.

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

Not only does inheritance ease modification it also promotes 'programming by difference' or 'exception'. "The new class is just like the existing class except ..." [59] is often a good way to describe new classes. In specializing a superclass the implementation of a subclass may:

1 ... include new methods.

2 ... include new class and instance variables.

3 ... override inherited methods.

There are two types of classes in a hierarchy -- abstract and concrete classes. Abstract classes do not allow instantiation and should serve only to group common structure and behaviour. Concrete classes allow instantiation. A class hierarchy should be designed so that superclasses are abstract and the subclasses at the bottom of the hierarchy are concrete classes.

Polymorphism

Methods and class structure can be used by subclasses through the mechanism of inheritance. *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 we 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 the its superclasses. If receivers are of different classes the same message can invoke completely different methods.

Receiver Object	Receiver Class	Mess-age	Argument	Operation	Result (Class)
3	Integer	*	4	3 * 4	12 (Integer)
4,6,1	Vector	*	3,5,4	i j k 4 6 1 3 5 4 (Cross-Prod)	19,13,2 (Vector)
4,6,1	Location	*	3,5,4	4 * 3 6 * 5 1 * 4 (Scale)	12,30,4 (Location)

Table A.1 Polymorphism

Table A.1 shows three instances of different classes receiving the method '*'. The result of each operation varies depending on the class of the receiver. To prevent confusion methods that have the same name should involve similar operations.

When conscientiously applied polymorphism 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. Code reuse is again encouraged.

In Smalltalk polymorphism is very valuable. For example, when trying to print a representation of an object the message 'printOn:' can be sent to instances of **String, Symbol, Character, Integer, Fraction** and **Time** among others. Instances of these classes all respond to this message by printing themselves in their own unique format. The methods for *printOn:* in each class know the details of how to print the instances.

Abstraction

The concept of abstraction deals with the external view of an object. An object is accessed through its interface, the internal details of an object are of no concern, only the results. Good abstractions allow the user to deal only with the external representation or interface of each object.

The abstraction chosen by a designer depends on his view or the intended use of an object. For example, the same abstraction of a machine tool is not needed for a machine tool programmer and a machine tool operator. Of most importance to the machine tool programmer is the language, memory capacity and the physical limitations of the machine tool. The machine tool operator must know where the emergency stop button is, how to tell if the machine is waiting for a tool change and how to restart the

machine tool after servicing. The features important to the programmer are not as important to the operator. The programmer can still program the machine tool without knowing the physical location of the emergency stop button. Thus the abstraction of the machine tool should be different for the programmer and operator.

Booch writes:

> "*An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer* [45]."

Data Encapsulation

Closely related to abstraction is data encapsulation. Data encapsulation involves data hiding. An object should be a black box so that users do not have and do not need access to the internal details of an object. Good encapsulation ensures that none of the internal details of an object are available to the object's clients. It is not good programming practice to make classes dependent on the internal representation of classes it uses.

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 is the same. Encapsulation also makes for safe software. Changes made to one class should not affect the behaviour of other classes.

The combination of inheritance, polymorphism, abstraction and data encapsulation provide unique characteristics to object-oriented programming. They encourage software reuse, simplify modifications and relieve the programmer of many tedious details.

A.2 Object-Oriented Notation

In sequential programming flow charts are an effective and accepted way of communicating a design. Object-oriented program designs have a much different nature. Currently there is no accepted documentation style for object-oriented programs. This section is devoted to an explanation of the notation used in this thesis for communicating the design of Smalltalk programs.

The notation used in this thesis is a subset of the notation presented by Booch [45]. The notation is very clear, concise and mainly graphical. Only a subset of Booch's notation is required because he developed a very general notation applicable to a variety of object-oriented languages including Smalltalk, C++, Object Pascal, CLOS and ADA. Smalltalk does not include all of the features of the other languages and therefore does not require the associated notation.

Object-oriented software requires two major designs. *Class structure* and *object structure* can both be presented to portray a clear understanding of the software. Class structure and object structure are closely related, because every object is an instance of a class.

An example should demonstrate the difference between class structure and object structure. A horizontal mill is a "kind of" mill. A mill and a lathe are both "kinds of" machine tools. The "kind of" relationship indicates inheritance. Inheritance occurs between classes and is indicated on a class diagram. A chuck and a tool post are "parts of" a lathe. Tool posts and chucks are not specializations of a lathe and should not be subclasses of a **Lathe** class. Rather a chuck and tool post are part of the implementation of a lathe. A lathe instance needs a chuck instance and a tool post instance. The "part of" relationship is between objects and is mainly expressed in an object structure diagram.

This section first describes the notation used in documenting a class structure followed by the section describing notation for object structure.

A.2.1 Class Diagrams

A class diagram graphically represents a class structure showing the existence of classes and their relationships.

Class Icon

A class is represented by a *cloud* outlined with a dashed line. The class name is placed inside the cloud. Figure A.2 shows a generic representation of a class.

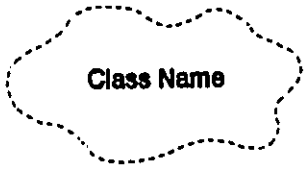

Figure A.2 Class Icon

Class Relationships

Figure A.3 shows the class relationship icon. There are two types of inheritance relationships. Compatible inheritance means the subclass is a slight variation on the superclass. Inheritance of a new type means that the subclass inherits from the superclass but its function and interface is significantly different from the superclass. Compatible inheritance is much more common than new type inheritance.

A using relationship indicates that a class actually relies upon the other class for part of its behaviour. Booch [45] provides a good example, a library and a book are definitely related. There is no library without books, but it would not be appropriate to say a library inherits from a book, there is no common behaviour or structure between the two. Rather a library uses a book.

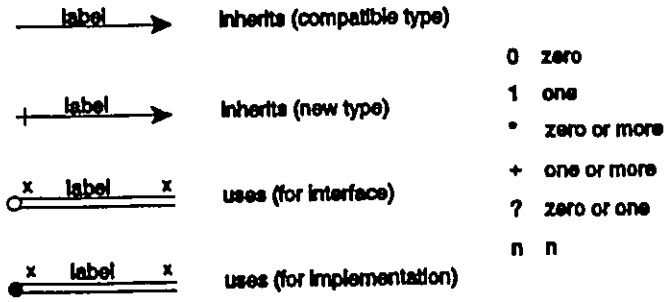

Figure A.3 Class Relationships and Cardinality

A using relationship designates that a class uses the resources of another. If the 'used' class must be visible to classes other than the 'using' class the relationship should be indicated as an interface using relationship. If the used class is hidden within the implementation of the using class the using relationship should indicate it is for implementation.

Cardinality indicates numeric values associated with using relationships. For instance, a lathe uses one chuck for its implementation and a chuck can only belong to one lathe. A lathe can have any number of tools but a tool can only be used by one machine. Figure A.4 provides a cardinality example. The '+' symbol near class 'B' means for every instance of class 'A' there is one or more instances of class 'B' and for every instance of class 'B' there is only one instance of class 'A'.

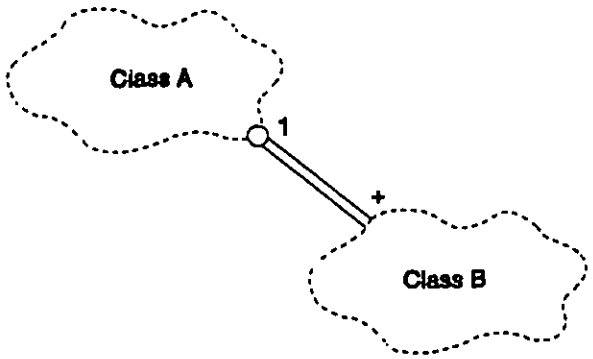

Figure A.4 Cardinality Example

A.2.2 Object Diagrams

Classes are generally static, objects are dynamic. Throughout the life of a program numerous objects are created and destroyed. A class diagram represents the design of classes at any instance in time. Object diagrams represent a time lapse portrayal of objects. Object diagrams display interactions that may occur among objects.

"*Class diagrams document the key abstractions in our system, and object diagrams highlight the important mechanisms that manipulate these abstractions* [45]."

Objects and their relationships are the most important elements of object diagrams. Object visibility and synchronization may also be included.

Object Icon & Object Relationships

The icon for an object is shown in Figure A.5. The icon is identical to the class icon except that the outline of the *cloud* is solid. The name inside the icon need not be a specific object name such as 'Lathe_9' but may represent any instance of class **Lathe** using the name 'aLathe'.

Figure A.5 Object Icon

A relationship between objects indicated by a solid line means that the objects can send messages to each other. A list of messages should accompany the line. The list consists of important messages that are transferred between the connected classes.

The relationships between classes may be adorned with extra symbols for indicating object visibility and message synchronization.

Object Visibility and Synchronization

Object visibility refers to how objects see each other. If an object is shared the visibility symbol will have a white background. Objects that are only visible to one object are not shared and the symbol has a black background.

The icons that indicate object visibility are shown in Figure A.6. There are three different types of visibility: *field*, *parameter* and *same lexical scope*. An object is a field if the object to which it is a field must be able to see it to send it messages. A field object is not part of the implementation of an object of which it is a field. Objects are in the same lexical scope if one object is used for another object's implementation. Smalltalk instance variables are represented either by field or scope visibility. An object is visible as a parameter if it is a dependent. Smalltalk supports dependents as a collection. Whenever an object undergoes a change that affects the dependents all the dependents are sent a message 'update:'.

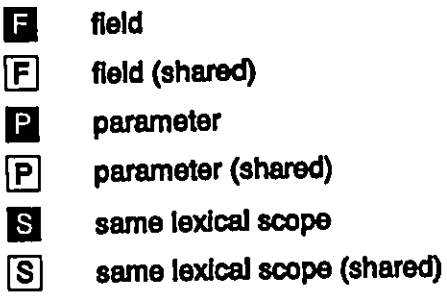

Figure A.6 Object Visibility

If object *part* is a shared field of object *lathe* then the symbol for a shared field should be placed at the end of the relationship line near the *part* object as in Figure A.7.

Object diagrams can include messages. Each message should accompany an a message synchronization symbol. The message synchronization symbols are shown in Figure A.8. For sequential systems an arrow pointing toward the receiver of the symbol is sufficient. The four other symbols are necessary for systems with multiple threads of

control. A synchronous message means the sender will wait indefinitely until both parties are ready to proceed. A balking message means the sender will abandon the operation if the receiver is not immediately ready. The timeout message means the receiver will wait for a set period of time for the receiver to become ready before the sender abandons the operation. An asynchronous message means the sender will initiate and interrupt the receiver regardless of whether the receiver is expecting the message.

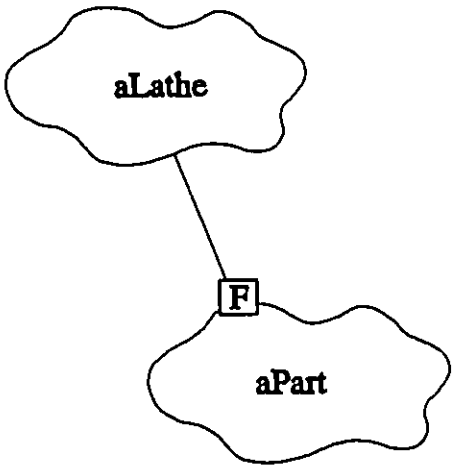

Figure A.7 Shared Field Example

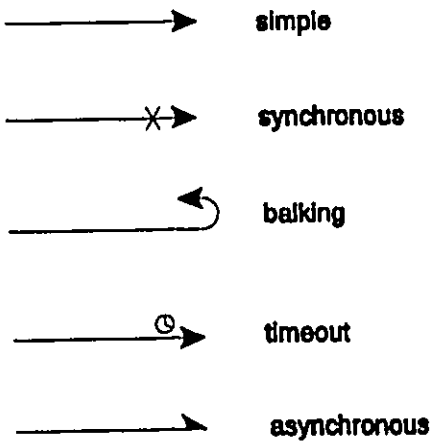

Figure A.8 Synchronization Symbols

Both class and object diagrams show detail at one level. A design for a complex system cannot be shown on one object and one class diagram. Typical software designs include many object and class diagrams. There should be one object and class diagram showing the overall software structure. More specialized object and class diagrams document each individual part. A class or object icon on a top level diagram may involve many classes, these should be shown in more detail on separate diagrams.

APPENDIX B

ADDITIONAL UNISET CLASSES

UniSet was created using the Smalltalk environment partly because of its many predefined classes. These classes eased the implementation of UniSet, especially the user interfaces. During the creation of UniSet additional user interface classes were defined because Smalltalk could not provide all the required functionality.

The additional user interface classes are presented in this appendix. They are not outlined in the body of the thesis because they were not part of the initial goal but a tool to reach the goal.

B.1 TopIconPane

The **TopIconPane** class was developed to allow the inclusion of icons anywhere within the extent of a window. A **TopIconPane** can be used in place of any **TopPane**. Class **TopIconPane** is just like class **TopPane** except that it includes an extra array of icons and any part of the window not covered by a pane is coloured. The similarity

between the two classes is a classic example of programming by difference. This naturally leads to class **TopIconPane** being defined as a subclass of class **TopPane.**

The definition of class **TopIconPane** includes two instance variables that are not defined in its' superclass. The first variable, *backGround* holds an integer describing the background colour of the pane. The second instance variable, *iconLocations* holds a symbol. The symbol should be an instance message in the window's model class. Upon execution the method must return an **Array** of icons to be displayed in the window. Label icons should not defined in this method, but included in the same manner as when using class **TopPane.**

Figure B.1 lists the UniDiskServer *open* method that instantiates a **TopIconPane** pane. There are three unique parts demonstrated in this method. On line 12 the background colour is declared with the message *backGround:* 5. The integer represents a colour.

In line 14 the message *iconLocate:* sets the variable *iconLocations* to the symbol *#locateIcons:in:*. The *locateIcons:in:* method is implemented in class **UniDiskServer** and shown in Figure B.2. This method creates a **Collection** of icons and determines their location in *aPane.*

Line 16 of Figure B.1 executes the method *createIcons* in class **UniDiskServer** and uses the value returned as the argument for the **TopIconPane** method *paneIcons:*. The *createIcons* method is shown in Figure B.3. The method returns an **Array** of **Forms.** The

```
1   open
2       "Open a Disk Services window for the UniSet environment."
3
4       | listLineHeight ratio aTopPane |
5   ratio := 3/5.
6   listLineHeight := ListFont height + 4.
7   aTopPane := TopIconPane new
8       model: self;
9       label: 'Disk Services ';
10      foreColor: 0;
11      backColor: 15;
12      backGround: 5;
13      menu: #updateMenu;
14      iconLocate: #locateIcons:in:;
15      rightIcons: #(resize collapse zoom);
16      paneIcons: self createIcons;
17      minimumSize: 40 * SysFontWidth
18          @ (15 * SysFontHeight);
19      yourself.
20  aTopPane addSubpane:
21      (ListPane new
22          ...
```

Figure B.1 UniDiskServer *open* method

Forms are initialized as *Display compatibleForms* in line 13. Line 17 sizes the form according to the string being displayed in the icon. The **Array** is passed to the **TopIconPane.**

In the *open* method the **Array** returned by the *createIcons* method in class **UniDiskServer** is saved as an element of the instance variable *iconArray* of class **TopIconPane.** The *iconArray* instance variable is inherited from class **TopPane.** It has expanded functionality in the **TopIconPane** subclass. Instead of a two element **Array** it has three. The third element is an **Array** of pane icons returned from the method

```
1   locateIcons: iconArray in: aPane
2   "Calculates the appropriate origins for the icons."
3       | aRectangle frame line iconColl listLineHeight |
4   listLineHeight := ListFont height + 4.
5   frame := aPane frame.
6   aRectangle := (frame origin + (0@(frame height - (4 * listLineHeight)))
            corner: frame corner) insetBy: 2@2.
8   line := aRectangle origin + (0@(aRectangle height // 3)).
9   iconColl := OrderedCollection new.
10  iconColl add: (iconArray at: 1);
11          add: (iconArray at: 2).
12  self space: iconColl over: aRectangle width at: line.
13  iconColl := OrderedCollection new.
14  iconColl add: (iconArray at: 3);
15          add: (iconArray at: 4);
16          add: (iconArray at: 5).
17  line := line + (0@(aRectangle height // 3)).
18  self space: iconColl over: aRectangle width at: line
```

Figure B.2 UniDiskServer *locateIcons:in* method

createIcons of class **UniDiskServer**. The first and second elements of *iconArray* are Arrays with the left and right label icons as elements.

The methods from class **TopPane** that display and control interaction with the icons had to be expanded in class **TopIconPane** to include the third element of the *paneIcons* array.

B.2 NoLabelTopPane

The UniEditor window includes the optional displaying of comments in a pane beneath the window. Smalltalk does not provide the functionality for a window to initiate

```
1   createIcons
2      "From paneIcons dictionary, creates instances of Icon class."
4      |index anArray icon aForm listLineHeight aScanner|
5   anArray := Array new: 5.
6   aScanner := CharacterScanner new.
7   listLineHeight := (ListFont height) + 4.
8   index := 0.
9   PaneIcons do:[:dict|
10    dict do: [:aString|
11      index := index + 1.
12      icon := Icon new.
13      aForm := Display compatibleForm new.
14      aForm width: ((ListFont stringWidth: aString) + 10)
15                height: listLineHeight.
16      aForm white.
17      aScanner initialize: (0@0 extent: (aForm width @ aForm
18                height))
19                font: ListFont
20                dest: aForm.
21      aScanner display: aString at: 5 @ 3.
22      aForm border: (aForm boundingBox).
23      icon form: aForm;
24          name: (dict keyAtValue: aString).
25      anArray at: index put: icon]].
26  ^anArray
```

Figure B.3 UniDiskServer *createIcons* Method

a new pane without redefining the window. To display the comment in its own pane a new window is instantiated. The window should not have its own separate label since it is meant to be perceived as part of the UniEditor window. To provide this functionality the class **NoLabelTopPane** was developed.

Class **NoLabelTopPane** is a subclass of **TopPane**. It does not have any new instance variables. Some of the methods are rewritten to display the window without the

label. The class can be used in the same way as **TopPane** except that it should not be sent the messages for assigning labels or icons.

B.3 LabelPane

Class **LabelPane** is for displaying a label in a pane. It is used by **UniEditor** to display a machine name immediately beneath the machine icons. Class **UniSetBrowserSide** uses it to display the language names on each side of the **UniSetBrowser** window.

LabelPane is a subclass of **GraphPane**. It has two additional instance variables, *label* and *labelMethod*. *Label* is the string currently displayed in the pane. *LabelMethod* contains a symbol for a method in the model's class. When executed it should return a string to be displayed in the label. This method allows the label to change depending on the model's state.

A **LabelPane** is defined the same as a **GraphPane** except the message *labelMethod:* must be executed during initialization. The pane is displayed with the label string in white centred on a black background.

B.4 GreyListPane

The **GreyListPane** class was developed to provide the functionality for the panes in the UniEditor window that display UniSet commands. To display the selected item in grey was a purely aesthetic alteration. More importantly extra functionality was added to allow use of a three button mouse in these panes. The methods *middleChange:*, *middleSelect* and *selectMiddleAtCursor* were implemented for this reason.

```
1   openOn: aFMC
2       "Open the UniSet cell programming editor."
3       | listLineHeight |
4   fmc := aFMC.
5   listLineHeight := ListFont height + 4.
6   (topPane := TopPane new)
7       label: 'U n i S e t  E d i t o r';
8       minimumSize: 200 @ 100;
9       model: self.
10  ...
11  topPane addSubpane:
12      (GreyListPane new
13          model: self;
14          name: #majorWordList;
15          change: #command:;
16          middleChange: #middleSelectCommand:;
17          menu: #viewComment;
18          framingRatio: ((3/4) @ (1/4) extent: (1/4) @ (1/2))).
19  ...
```

Figure B.4 UniEditor *openOn:* method

Figure B.4 includes a listing of how to initialize a **GreyListPane** instance. The initialization routine is identical to a **ListPane** except for line 16. The *middleChange:*

method accepts a symbol for a message in the model's class. This message will be executed whenever the middle mouse button is depressed inside the boundaries of the pane.

```
1   middleSelectCommand: aCommand
2       "A command has been selected with the middle button.
3        Display the changes but do not force selection
4        onto program text."
5   command := aCommand.
6   commandShown := false.
7   modifier := nil.
8   self newComment;
9       changed: #minorWordList
```

Figure B.5 UniEditor *middleSelectCommand:* method

The *middleSelectCommand:* method that is the argument in line 16 of Figure B.4 is listed in Figure B.5. The method changes the values of a few instance variables of the **UniEditor** instance and then on line 9 updates the dependent *minorWordList* pane.

B.5 TextDisplay

The UniEditor and UniDiskBrowser windows both require a pane that displays text without allowing the operator the ability to edit the text. The UniEditor displays the comment in such a pane and the UniDiskBrowser displays file contents in such a pane.

Class **TextDisplay** was created to provide a pane for displaying text without editing capabilities. It inherits from class **TextPane**. Its default dispatcher class is

TextDisplayer. The initialization of a **TextDisplay** pane is the same as a **TextPane.** Clicking a mouse button while in a **TextDisplay** pane causes the window to close.

APPENDIX C

FORMAL UNISET SYNTAX

This appendix provides a complete definition of UniSet syntax using the Extended Backus-Naur Formalism used in *Programming in Modula-2* by Niklaus Wirth, Springer-Verlag, 1982 and Smalltalk/V 286 Digitalk Inc. 1988. It is included to show formally what alphabet is recognized the within UniSet environment. Along with the alphabet, the syntax rules are a formal definition of how UniSet instructions can be combined in a program.

The Extended Backus-Naur Form (EBNF) is not as structured as the classic Backus-Naur Form syntax definition. This provides a more readable and understandable description. The following five rules define EBNF using EBNF syntax [44].

<rule> syntax = {rule}

<rule> rule = "<rule>" identifier "=" expression ".".

<rule> expression = term {"|" term}.

<rule> term = factor {factor}.

<rule> factor = identifier | string | "(" expression ")" |
"[" expression "]" | "{" expression "}".

The left side of each rule is defined by the rules on the right side. The right side defines the syntax in terms of other rule names and terminal symbols. Following is an explanation of the symbols used above.

1) Characters or character sequences enclosed by double quotes " or apostrophe ' are terminal symbols. Terminal symbols cannot be further defined, all successful parsing ends with terminal symbols.
2) Parentheses, (and), group alternative terms.
3) The vertical bar, |, separates alternative terms.
4) Brackets, [and], identify optional expressions.
5) Braces, { and }, identify expressions that may occur 0 or more times.
6) An *identifier* is a sequence of letters and digits beginning with a letter.

Following is an EBNF syntax specification for UniSet.

1) program = machineName expressionSeries [program]

2) expressionSeries = {expression "."}

3) expression = [variableName ":="] (unarySelector | modifiedExpression | keywordExpression)

4) machineName = identifier

5) unarySelector = identifier

6) modifiedExpression = modifierSelector "," modifier

7) modifierSelector = identifier

8) modifier = (identifier | keyWordExpression)

9) keyWordExpression = keyWordSelector ":" argument

10) keyWordSelector = identifier

11) argument = (variableName | SmalltalkExpression)

12) variableName = identifier

13) SmalltalkExpression = Refer to Smalltalk manual

14) identifier = letter {letter | digit}

15) letter = capitalLetter |
"a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" |
"i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" |
"q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" |
"y" | "z".

16) capitalLetter =

"A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" |

"I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" |

"Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" |

"Y" | "Z".

17) digits = digit [digit].

18) digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".

19) comment = '"' {character | "'"} '"'.

ALPHABET

a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ,, +, /, \, *, ~, <, >, =, @, %, |, &, ?, !, [,], {, }, (,), ^, ;, $, #, :, |, ?, !