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

A COMPUTATIONAL MODEL FOR SKILLS-ORIENTED ROBOT PROGRAMMING

by

Colin C. Archibald

A THESIS SUBMITTED TO THE
SCHOOL OF GRADUATE STUDIES AND RESEARCH
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

OTTAWA-CARLETON INSTITUTE FOR ELECTRICAL ENGINEERING
DEPARTMENT OF ELECTRICAL ENGINEERING
FACULTY OF ENGINEERING
UNIVERSITY OF OTTAWA
OTTAWA, ONTARIO, CANADA K1N 6N5

ISBN 0-612-04893-4

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

Acknowledgements

This research was carried out while I was on education leave from my position at the National Research Council of Canada. I would like to acknowledge the contributions of NRC, which were not only financial, but also for the providing research facilities and a place within a highly qualified group of robotics researchers. Thanks to Dan Johnston of the Institute for Advanced Manufacturing Technology at NRC for the name SKOMP, and suggestions on what might be considered useful to shop floor NC machine programmers. Evelyn Kidd, also at NRC, has made many valuable suggestions on how to improve this thesis.

Many people have contributed to this work through discussion and debate. I would like to acknowledge the contributions of my thesis supervisor Emil Petriu who has pushed me into creating something of genuine value. Also at the University of Ottawa, Moshe Krieger has contributed to many of the ideas especially on realtime programming. I also thank Trevor Jones at CRS Robotics, Inc. for many discussions on what is required by the industrial robotics industry.

Abstract

A complete computational model for industrial robot programming is presented. There are two main objectives in realizing this model. The first is to allow shop floor production engineers (application programmers) to create and modify sensor-based robot programs. The proposed iconic user interface provides a non-textual programming mechanism. The icons, which represent individual robot skills, are linked and parameterized to modify the behaviour of the skills. Use of a control flow mechanism, as opposed to data flow, makes the description of the robot operation as a set of skills immediately obvious. Linking the skill icons requires only a few control constructs which makes the interface usable on the shop floor. This system provides a mechanism for online creation and debugging of sensor-based robot operations.

The second objective is to enable the system programmer to create and maintain the robot skills using consistent and facilitated methods. This is the underlying software architecture that makes the iconic shop floor interface possible. It is an object-based method that provides functional abstraction of the sensors and machines. The objects include skills, sensor drivers, logical sensors, and machine drivers. The skills are defined in the form of templates that completely specify a sensor-based robot action.

Other significant results ensue from the two listed above, for example, the possibility of standardization in robot programming at the skill level. The ability to separate

the responsibilities of individuals with different capabilities is another objective that has the side effect of making robot systems development manageable.

The computational model presented is called "SKills-Oriented Robot Programming (SKORP)." In this model the skills execute exclusive of each other and therefore the computation for each skill can be represented independently. Skills are designed and documented using realtime design tools from the multiactivity paradigm. The SKORP model provides consistent and usable design methods for describing computation in embedded systems. These design tools are used by the system programmer to guarantee the realtime interaction of the software modules that compose a skill.

This research is directed toward industrial robotics in traditional and non-traditional habitats, but the model presented is equally applicable to any numerically controlled machine that either requires sensors or interacts with the environment in a complex way.

Contents

List of Figures

Chapter 1

Introduction

1.1 Overview

The remarkable growth of general purpose computers can largely be attributed to issues of *usability*. Going from punch cards as a human-machine interface to interactive graphical interfaces using hand-held mice has made the computer a commonplace tool. But the usability of industrial robots has not maintained a comparable pace. *Flexible* robots, which can be changed from one application to another, are widely reported by, and unfortunately limited to, the academic research laboratories. However, in industry the robots are fixed in a particular application and changing their function is considered a salvage operation. Even small modifications in the robot operations are a major expense. This is largely because the robots are still programmed at the machine level. They are not usable in the way that general-purpose computing equipment has become usable.

The motivation for the work reported here began with the recognition that almost all robots functioning productively are either programmed using an explicit language

(where all motions are preprogrammed) or on the shop floor using some form of *teach pendant* or *console*. The creation of the robot operation is *ad hoc*, i.e., it is directed at solving the current prevailing problem. Creating a complex system using *ad hoc* components that all fit together in different ways is an inefficient and unsatisfying process. It cannot be properly managed and does not receive a rigorous structured design.

Shop floor programming is accomplished by using a human-robot interface (HRI) that provides reduced programming capability suitable for the production engineer. The most utilized shop floor interface is the teach pendant. This is a hand-held electronic box with a small display and buttons for moving the robot manually. Figure 1.1 shows two commercial teach pendants. The robot programmer either moves the robot physically with his hands (small robots) or uses the teach pendant to position the robot in the desired location, then teaching these points to the robot controller. The sequence of motions is then stored.

More recent shop floor interfaces include consoles with many buttons (sometimes more than 100) each of which represents some functionality of the robot. Some of these interfaces offer the possibility of textual programming and, although programs are not usually created this way, editing of existing programs from a teach pendant has been found to be quite useful. Although this gives the shop floor programmer increased flexibility, the number of buttons has grown until the consoles have become unwieldy. Some look like QWERTY terminals with a video screen. Although a terminal on the shop floor provides maximum potential capability, production engineers cannot effectively write or even modify code on the shop floor, especially if the code is executing on a parallel architecture. The attempt to add capability at the shop floor has, in

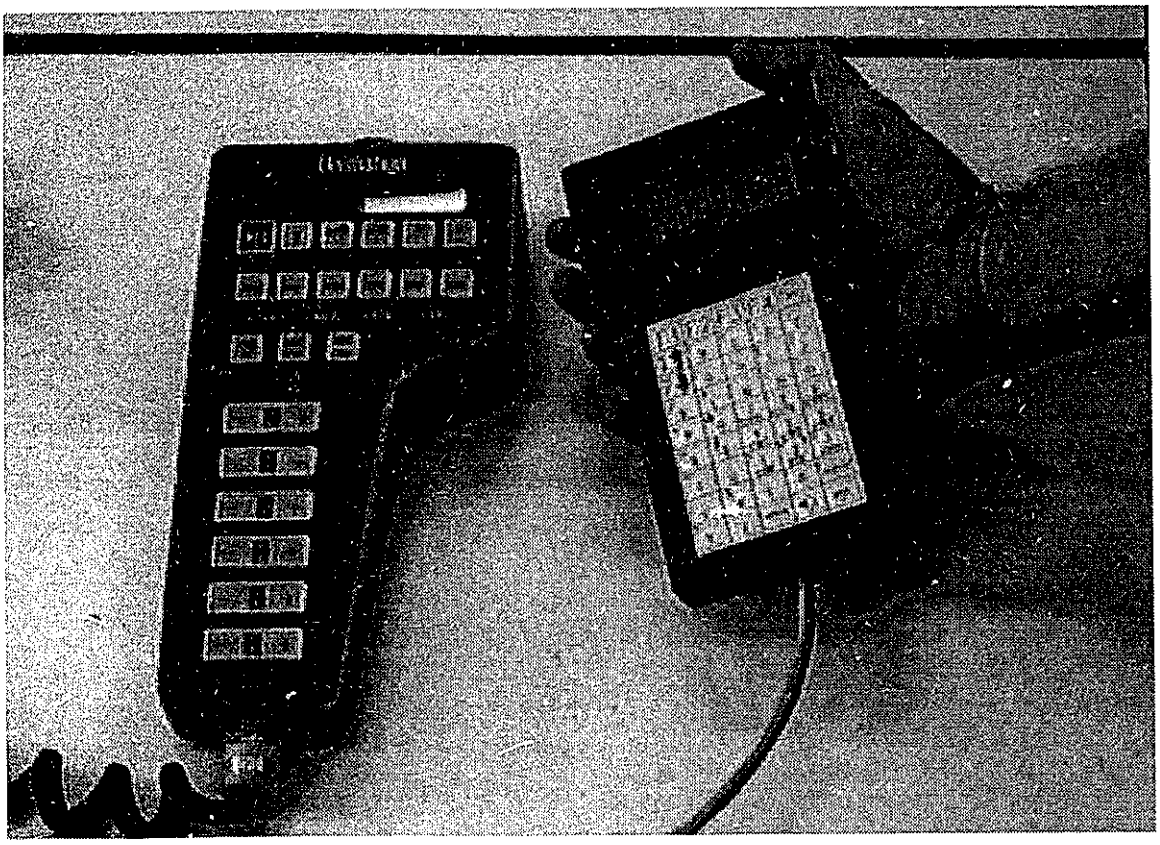

Figure 1.1: Two industrial teach pendants for shop floor programming.

fact, increased the responsibility at the single level of robot systems development that currently exists. It has also subverted the possibility of a manageable mechanism for the structured design of such systems.

There are two easily recognized avenues for industrial robots to become more effectual:

- Add sensory capabilities
- Reduce cost of programming

Currently most industrial robots have no external sensors. This restricts the robot habitat to a well-known and unvarying work space. A significant improvement can be made by providing sensor information that allows the robot habitat to be only slightly less than completely known. As the capabilities of the sensors and sensor data interpretation increase, the *a priori* knowledge of the robot work space can be decreased, thereby making more applications technically and financially feasible. A robot is a programmable mechanical instrument and sensors are also instruments. A major contribution of this research is the creation of a computational model specifically directed toward the control of mechanical devices that use information collected from sensors.

Programming robots that use sensors introduces several interrelated issues. For example, one has to consider (a) the collection and interpretation of sensor data, and (b) the realtime control of robots that use the results of sensor data interpretation in the control loop. These are difficult and multidisciplinary problems. Adaptive control requires expertise in realtime computing, control theory, sensing, signal processing, mechanical servoing, kinematics and dynamics [1]. Reducing the cost of robot programming and introducing sensors are closely related issues and should be considered

together. To date the sensors have been added to robots one at a time in an *ad hoc* way to meet the demands of an application. The software is developed in the same manner. A computational model is required that provides a consistent and manageable framework for programming robots that use sensors. The subject of this thesis is the development of such a computational model suitable for practicable shop-floor realization of robotic applications.

1.2 Robot Programming

Current robot programming methods are referred to as either *explicit* or *implicit* [2]. Explicit methods are sometimes called *robot-level*, meaning that the motions of the robot are expressed explicitly. Implicit methods are often referred to as task-level or *task-directed* [3], and the operations are expressed in terms of the objects being manipulated. In this case, the robot motions are computed from a higher-level description of the task, usually using geometric models of the environment and the robot and some reasoning and planning techniques [3, 4, 5, 6].

It has been reported that more than 100 different languages have been developed for robot programming [2]. The explicit languages, such as VAL II [7] and AML [8], have been successful commercially, although they have many deficiencies that were identified many years ago [9]. Explicit commercial languages have the following weaknesses:

1. robot specific
2. require a skilled computer programmer
3. frequent absence of constructs for data structures and flow control that are common in modern high-level general-purpose programming languages

4. require hardware specific to the robot manufacturer, designed to reduce product costs

5. very limited capability to integrate sensors and sensing algorithms

6. multitasking and multiprocessing unavailable

7. require operations to be expressed at a level of abstraction that is disappointing to the potentially productive programmer

8. lack of debugging tools

These languages have been commercially successful in spite of these issues because there are many high throughput manufacturing applications for robots that do not require sensors or complicated algorithms to be profitable. These languages have been compared and analyzed many times. (See [1, 10, 11, 12] for some comparative studies.) Some limitations on these languages are intentional. The individuals programming robots are not usually professional computer programmers. Industrial and mechanical engineers who are trying to find mechanical solutions on the shop floor require relatively simple languages, with a minimum of detail. Unfortunately the limitations on commercial robot languages have restricted the application of most robots to operations such as pick-and-place, spray painting, and welding of known parts in known locations. The use of robots with even simple sensors is currently so cumbersome that it is very restricted [13].

Implicit programming languages that have been developed, such as RAPT [14] and Autopass [15], are interesting, but there are fundamental barriers to these systems actually becoming commercially usable. These programming languages claim to provide

task-level robot programming environments [3, 16]. This implies that the robot activity is specified in terms of the resultant state of the objects being manipulated and that the program will automatically determine which actions to take and in which order, etc. This specious approach leads to impressive simulations but it has not found acceptance in the robot industry. Closely related to implicit programming is the large body of research that provides components of task-level programming. These include task planning, motion planning, workspace and process modeling, knowledge-base creation and maintenance, etc.

Off-line automatic task-level programming has had limited success in real environments mainly because the models required for these methods are imperfect. The best model of a sensor is the sensor itself. This is also true for the robot, the objects in the robot habitat, and the interactions among them. Modeling the exact behaviour of a robot that uses sensors requires exact models of the geometry of the robot habitat, the geometry of the robot, the kinematics and dynamics of the robot, the performance of the sensors, and the physical interactions of the robot tool with the object. Even if these models could be produced accurately, using them in a realistic way would soon become intractable. The value of a simulation becomes highly questionable if the models on which it is based are questionable or the models of the really difficult aspects of the problem are simplified to such a degree that they provide no insight into a physical solution. There is a prevalent view that simulation and modeling the environment is the best way to make progress in task-level robot programming. The result is almost never a physical robot that performs a useful function, and it has never been demonstrated that this is a commercially viable approach. Some stages of technology evolution, which are required to provide a foundation for further progress, have been

ignored. Explicit robot programming can be fundamentally improved by the creation of a computational model that can take advantage of modern computing architectures and software tools.

It is interesting to note that the academic community diverged from the industrial research community as microprocessor and sensor technology advanced rapidly in the 1980's. Figure 1.2 shows the progress, from top to bottom, of what happened. Industry created robots specialized for particular applications or classes of applications. The programmer's interfaces for these machines are also highly specific. This resulted from the need to provide turnkey solutions or at least easily programmable systems for end users. The academic community focussed on open architecture controllers. These controllers, many of them multiprocessor [17, 18, 19, 20, 21], provide programmer access to the control loop. Open architectures permitted the development of reactive and reflexive robot actions that use sensor data. Examples include recalculating the tool trajectory while the robot is in motion, and dynamic collision avoidance. These controllers have been slow to move into the robotics industry largely because they are extremely difficult to program and there has been little effort to develop mechanisms for software reuse. To create robotics applications using this type of programming requires an individual knowledgeable in multiple disciplines, including sensor data interpretation, realtime programming, robot control, and the details of the application itself. The various software engineering skills required are rarely found in a single individual and should not be required in a well-considered computational model for programmable machine control.

The compromise between the previous academic and industrial approaches is to develop computing technology that allows the robot programmer to *explicitly* program

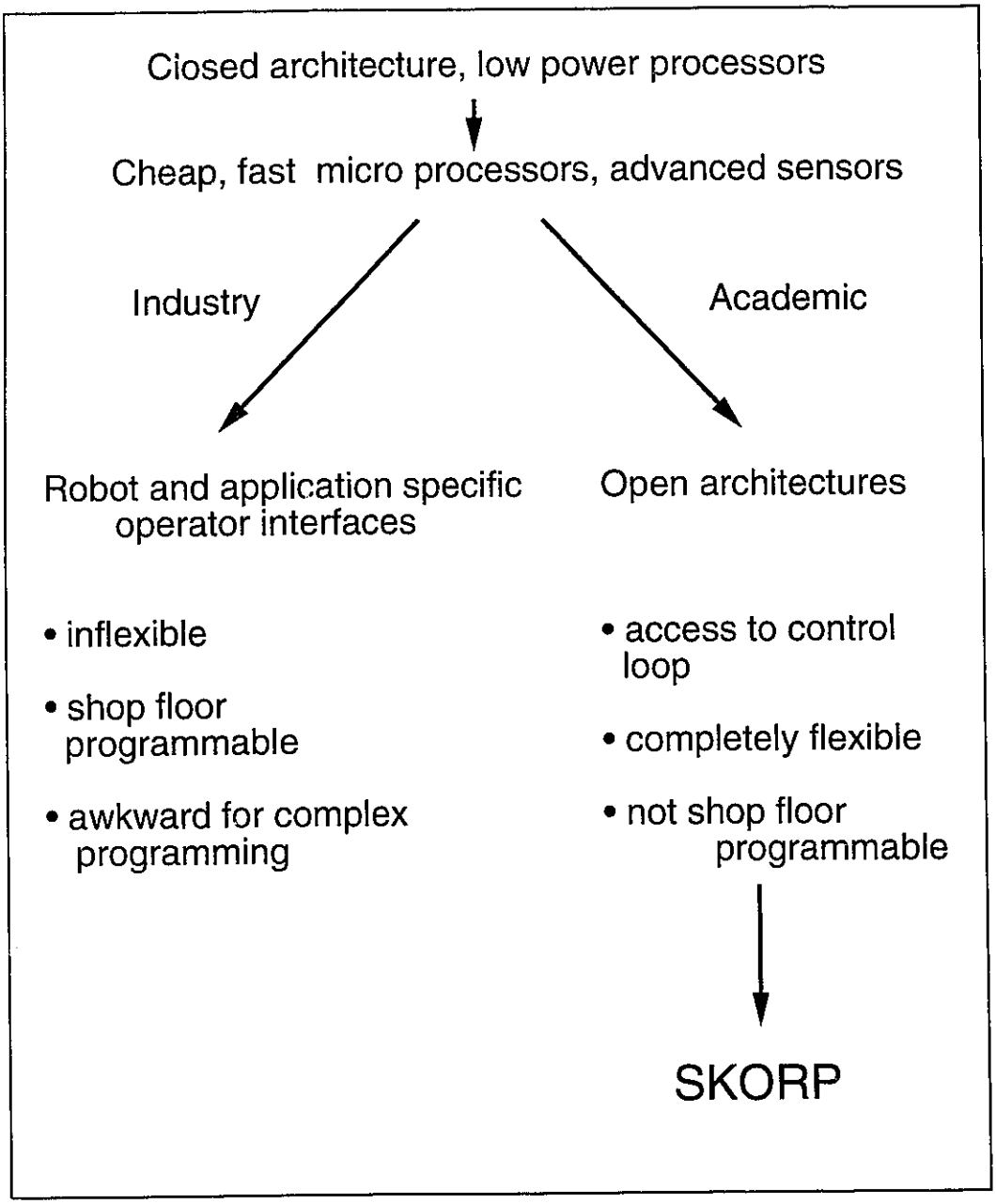

Figure 1.2: The progression of computing research for robotics – the industrial and academic trends.

a robot operation on the shop floor. To make this realizable, programming must be at a level of abstraction that permits the issues to be considered by the shop floor programmer. The programmer can produce an explicit program at the correct level of abstraction and debug, modify, and maintain at this level of detail. This is a general description of the next logical step – to facilitate the use of the open architectures developed in the last decade.

1.3 System Architecture

Designers of system architecture for industrial robotics must consider not only current hardware and software but also potential requirements. Systems that are designed for a specific fixed hardware architecture become obsolete very quickly. Consider the robot operating system (ROS) proposed by Zhang [22] in 1989. He recognized the requirements for modularity but did not anticipate the need to add additional computing power as the systems grew. Single processor hardware systems are known to be inadequate for the future requirements of sensor-based robot control [20]. Technology that permits the addition of computing power without disruption of the software design is essential [23]. It is surprising that these issues were identified clearly as early as 1986 (see [23] for example) and yet there has been very little progress toward a commercially viable robot architecture that is modular and expandable in both hardware and software.

It may be interesting to note here that the direction of the progress in robot programming has been heavily influenced by the NASREM architecture defined by Albus et al. [24]. This hierarchical model defined six levels of control as follows [25]:

1. Servo Level - performs motions that are small in a dynamic sense.

2. Primitive level - performs motions that are large in a dynamic sense.

3. Elemental Move Level - transforms goals described from the task point of view into goals from a manipulator point of view.

4. Task Level - is the "man-equivalent" level. Goals are planned based on a geometric description of the world, incorporating "common-sense physics."

5. Service Bay Level - coordinates groups of task level robots.

6. Mission Level - sets priorities for the activities.

When this model was first proposed the state of the art in computer programming was to solve all problems by successive decomposition which can easily be represented as a hierarchy. The fact that the level immediately above *elemental move* is called *task level* is a plausible explanation of why subsequently developed robot computational models stressed the need to move from robot-level languages to implicit languages, i.e., moving up in this hierarchy has been accepted as the appropriate direction for academic research in robot programming. For industrial sensor-based robot programming, a computational model for explicit robot programming using suitable abstraction techniques will have a larger immediate impact on the robotics industry.

1.4 Levels of Abstraction

An important issue in sensor-based robot programming, as in man-machine communication in general, is the level of abstraction in the interaction. A programming

mechanism, usually a formal language, must satisfy both epistemological and practical requirements. The robot programmer must be able to express all of the required concepts in an unambiguous way. In addition, the operations must be repeatable. In an adaptive robot action *repeatable* implies that the adaptation is consistent within the variations that may occur. The operations must also satisfy the practical requirements of execution speed, debugging capabilities, and rapid prototyping. In human-robot communication (HRC) this is complicated by the varying levels of interaction that are required. These varying levels are indicated by (although not restricted to) the required response times of the tasks being programmed. For example, the developer may have to program an active sensor requiring responses measured in microseconds, sensor data interpretation in the order of milliseconds or tens of milliseconds, and a graphical programming environment that must respond in the order of hundreds of milliseconds. Coincidentally, these three examples are in decreasing response time requirements and increasing levels of abstraction from the perspective of the robot programmer. All cannot be effectively programmed using the same computational paradigm. The microsecond level of programming is suitably implemented in machine language. The sensor data collection and interpretation are most suited to a higher-level language with adequate control constructs and a good capacity for creating abstract data types. C and C++ are now widely used for this level. In construction of a graphical programming environment, the emphasis moves away from timing considerations to the proper presentation of requisite information for the programmer.

The difficulty that arises is how to amalgamate several levels of abstraction requiring multiple programming paradigms. If all of the information contained in the lower levels is encapsulated and the robot programmer has no access to it, the system

will be too rigid and its capabilities will be denied to the robot programmer at the higher levels. However, if the robot programmer is responsible for all minutia at all levels, the programming effort will require someone knowledgeable in several diverse disciplines. This not only increases the cost of programming, but effectively eliminates the possibility of having manageable system design and development.

The robot programmer interface for prototyping of applications must balance the requirements of flexibility and usability. It is common to trade off these requirements. As the amount of detail that is presented to the user increases, the flexibility also increases but the usability decreases. This is delicate balance, that can never be achieved perfectly for all levels of sophistication of the users. Therefore, the system used to create a programming interface must have easy to use mechanisms for creating new interactive mechanisms. This concept of a user interface to create a user interface is relatively new. It is proposed here that a graphical programming interface can provide the mechanisms to create a balanced final product.

Graphical interfaces for presenting processes at the proper level of abstraction have been implemented for some application areas. For example, Paragon Inc. has developed a commercially available programming system called Visualization Workbench, which uses icons to link image processing transformations [26]. The results of each transformation are immediately visible in the resultant images, and modifications can be made interactively. Iterative modifications and immediate results create a powerful interactive debugging tool. Another product, LabVIEW from National Instruments Inc. [27, 28], is available for graphically building software device drivers. This system is very complex. The programmer has a graphical equivalent of the control structures found in modern high-level programming languages. These icons can be used to create

static graphical interfaces simulating the dials and controls on electronic devices such as oscilloscopes. In the robotics literature, those proposing a graphical interface are working with a robot simulation or doing off-line programming [29]. We will present a graphical programming interface based on abstraction and decomposition for on-line shop floor robot application programming and maintenance.

1.5 Robot Skills

A robot skill is the capability of a robot to repeatedly accomplish a physical action that can be described unambiguously. If this action requires the use of a sensor, the skill is referred to as a sensor-based skill. Robot skills that do not require sensors include *move to a predefined location, change tools that are located in a predefined location, pick or place in a predefined location*, and *open and close a gripper*. These skills alone comprise most of the currently existing industrial robot systems. Some examples of sensor-based skills are *approach to touch, move away from the scene, align the end effector normal to the flat surface, grasp the object that is moving, grasp the object in the field of view of the sensor, follow an edge*, and *rub in a specified direction with a specified force*. Note that each of these skills is fairly well defined in a few easily understood words even though they will execute differently depending on the current state of the robot habitat. In general, sensor-based robot skills are more difficult to implement than non-sensor-based skills and are more useful in that they adapt to the state of the environment in some way. The robot skill is the basic unit of decomposition for the design of a robot application. It is also the basic unit for implementation. The robot skill is the meeting place for the low-level system programmers and the high-level application programmers.

1.5.1 Debugging Robot Programs

It is well known that debugging tools are essential for efficient computer programming. This is also true for robot programming. Robot programs cannot be so thoroughly planned before implementation that they will execute correctly on the first attempt. Simulations based on idealistic robot models can detect a class of potential errors, but unexpected problems will become apparent only when the real robot actually attempts to carry out the intended operation in the target robot habitat. It is often necessary to try several higher-level strategies to achieve the desired behaviour. This is especially true with robot programs that use sensor data. Even when off-line programming is used as a preliminary stage of robot installation the iterative process of positioning and fine-tuning the motions is frustrating and lengthy. It is proposed here that an environment for effective on-line shop floor programming will reduce the dependence on the off-line stage and thus the overall cost of robot programming.

If it is accepted that an iterative debugging process will occur for both strategy and fine tuning, an environment that facilitates this process in the real robot habitat is required. It has been proven that the presence, but not the absence, of some errors in a program can be detected by preprocessing. This is even more significant in a physical environment where the inputs (sensor data) cannot be predicted exactly and the correct outputs (robot action) will vary depending on the inputs. It is this unpredictability that makes an on-line iterative debugging capability necessary.

1.5.2 Top-down Versus Bottom-up Robotic Application Design

There are two approaches to developing advanced robotic capabilities using sensors. These concepts, loosely referred to as *top-down* and *bottom-up*, are well known for software design and the general concensus has been that top-down is most effective. For robotics applications the top-down approach to design has some specific implications. Geometric models of the habitat, the objects in the habitat, and the robot are instruments of the design. Creating the models or importing them from another system is done in an early stage of system development, and all subsequent development interacts with these models. Often the last stage of the development is to take the system to an actual robot for testing. Some examples of such systems can be found in [4, 5, 6, 30, 31]. A major problem with the top-down approach is that the designer almost never knows what the primitive functional elements of the system are or if it is possible to create them to meet the design specifications. Successive decomposition, resulting in some small implementable functions, is possible only when the functions are known to be implementable. Although this is true in software systems that do not interact with the world physically, it is almost never true in robotic systems design.

In the bottom-up approach, the lower-level elements are developed and tested first. Models used are minimal to accomplish specific tasks. Kak et al. [32] have proposed that models should be "tuned" to the sensor(s). These "sensor tuned" models contain only enough information to achieve procedural and descriptive adequacy for a particular task. The bottom-up approach has been referred to as *physically grounding* [33] because the merit of the robotic task is based on what it can actually do *physically*, as opposed to theoretically.

Another notable difference in bottom-up robot program development is that some world model information is encapsulated in the skill definition itself, e.g., if a skill is defined to "align the end effector normal to a flat surface," the assumption that a flat surface exists is in the skill and this constitutes a partial model of the habitat. It is assumed that the programmer will not invoke the skill unless it is known that such a situation exists. Only that part of the environment useful for a particular skill is included. There is psychophysical evidence that most living creatures accomplish their objectives without accessing complete models, and only recently in the evolution of human beings has an abstract model been used to accomplish more complex tasks [33].

This paradigm of using minimal models of the environment and testing functionality from the bottom up led us to the development of reactive or reflexive robot skills. Reflexive behaviour means that a reflex-like motion is made based on relatively unprocessed sensor data. A reactive skill implies that a decision on how to respond is calculated after some interpretation of the sensor data. Some sensor-based skills have already been developed using *reflexive behaviour* [34]. Brooks' subsumption architecture [35] is a collection of reactive skills that automatically subsume each other on specific events.

There are many more skills in the physically grounded approach than in top-down design. Each physically grounded skill is designed quite specifically to interact with a class of objects or habitats and to perform in a specific manner. The resulting system is a combination of an event-response system, where the robot skills will react to the current state, and explicit robot programming, where the skills required for a particular operation are explicitly invoked. This leads to a hybrid system that combines both top-down and bottom-up design principles. This hybrid organization is used to advantage

in the design of the computational model presented herein.

Physically grounded robot skills (reflexive and reactive) that are can be linked to form complex robotic operations. Because these skills are developed bottom up and are tested in the physical habitat, they can be used confidently as tools for the top-down design of an entire application.

1.6 The SKORP Model

The essence of SKills-Oriented Robot Programming (SKORP) is that robot operations can be divided into discrete units of physical action called *skills*. Skills that use sensors are referred to as *sensor-based* skills. Skill modules are the highest level of abstraction in the bottom-up portion of the system design and the lowest level of abstraction in the top-down design. This meeting point will be discussed at length throughout this thesis.

The SKORP model is introduced here from the perspectives of the users of the model. Four individuals (or teams) who share the responsibilities of creating a functioning robotic application using SKORP are identified:

- robot systems analyst
- system programmer
- application programmer
- operator

The robot systems analyst identifies what skills are required for a particular application. He determines if these skills already exist and, if not, the requirements for

the skills are created at this level. The robot systems analyst is a specialist in the specification of robot skills and operations.

The system programmer is responsible for the actual creation of the skills. He uses the sensors and machines to create and test the skills in a laboratory or production design environment. The systems programmer is a specialist in realtime programming and control of sensors and mechanical devices.

The application programmer uses the skills that were created by the system programmer and the specification of the operation provided by the robot systems analyst to create the complete application. He uses an iconic programming environment where the icons correspond to skills. The icons are linked using a pointing device, such as a mouse, to develop the application from the predefined and tested skills. The skills are parameterized by the applications programmer to *fine-tune* them for the precise needs of the particular application. The application programmer, a specialist in production engineering, works on the shop floor.

The operator also works on the shop floor, but does no programming. He oversees the use of the programs in operation, starts and stops the applications, and makes minor adjustments to the parameters of the skills to maintain consistency of the operations that affect such things as quality control of the product.

In this research tools are provided for each of these four individuals. The robot systems analyst is given tools to design skills in a consistent manner. Skills must be designed in cooperation with the system programmer to ensure that they are either in existence or feasible to create.

The system programmer is given tools for the creation of the skills in a realtime subsystem. The most difficult aspect of his responsibility is to meet realtime deadlines.

Realtime design tools are presented specifically for the design of sensor-based robot skills.

The application programmer is provided with a graphical interface connected directly to a manipulator. Furthermore, an iconic interface that presents the skills of a robot operation as individual icons provides a mechanism flexible enough to grant access to all necessary information while leaving the unnecessary detail at the lower levels.

Finally the operator is given restricted access to the iconic interface to start, stop, and maintain the performance of the operations.

1.7 Objectives and Contributions

This research is motivated by the industrial robot user's frustration at the lack of tools for creating and using the sensor-based skills. Some of these skills were extensively studied at the National Research Council of Canada (NRC), including *inserting of a block in a hole using a force-torque sensor* [36], *following a moving object* [37], *pose determination* [38] and *bin picking* [39]. Using traditional robot programming techniques, each of these skills required approximately 18 months to create and each was developed in a different style or completely different architecture. Although interesting things were learned in these projects, the skills could never be used in conjunction with each other and it has been almost impossible to interest industry to commercialize them.

The threat of obsolesence in industrial automation is a further motivator. Microprocessors, sensors, and robotic manipulators themselves are being replaced with faster, smaller, and more efficient devices more rapidly than ever before. Robotic systems are

being installed that contain already obsolete components. One of the reasons for this unresponsiveness is the absence of an upgradable computational model that can be modified in a modular way. Companies that produce monolithic robotic systems that must be replaced in their entirity are inherently unresponsive to inovative technology components.

The creation of software for sensor-based robotics in modern industrial environments requires a computational model that provides

- modularity
- extendibility
- usability
- reusability
- accretion
- debugging facilities

These are the objective traits that have been adopted for the SKORP computational model. To achieve this, sensor-based robot programming has been divided into tasks for four different individuals and each of them is given tools to execute their responsibilities within a single computational model. A successful computational model for sensor-based robot programming must provide different programming methods for the very different requirements of applications programming and systems programming.

The production engineer (referred to as the applications programmer) is provided with a non-textual robot programming environment for use on the shop floor. The user interface must be quite simple to be effective. The systems programmer is provided

with tools for the consistent development of the individual skills. The tools must address the most significant difficulties of developing this type of realtime software, namely meeting realtime deadlines. The applications programmer and the systems programmer must be able to integrate their efforts seamlessly.

A layered software architecture is presented for integration of the components. This architecture is presented as a hybrid hierarchy with the systems programmer developing skills bottom up and the applications programmer developing applications top down.

1.7.1 Contributions

The contributions of this thesis are the following

- a computational model for industrial robot programming that separates the tasks involved in the creation of robot applications into manageable sections that can take advantage of the specialities of various developers
- study of methods used for robot application programming resulting in the concepts of the robot *skill* and the robot operation as a network of skills using the control flow model
- methods for the consistent design and implementation of realtime sensor-based robot skills
- an icon-based robot application programming environment for use on the shop floor by non-computer scientists
- the control flow method for specifying the steps in a robotic operation using icons that represent skills

- sensor-based robot skills, in particular, a skill to follow an edge using a wrist-mounted laser range finder

1.8 Thesis Outline

Chapter 2 gives a high-level description of the SKills-Oriented Robot Programming (SKORP) computational model including robot operations, skills, and the modules that make up the skills. The structure of SKORP and the methods that the robot systems analyst uses to design the skills and operations are given.

Chapter 3 will deal with the development of the individual skills on the realtime subsystem and describe the tools provided to the systems programmer. The emphasis is on the tools for realtime software development for sensor-based robot skills, i.e., the realtime interaction with sensors and machines.

The focus in Chapter 4 is on the applications programmer. The tools afforded to the applications programmer to create a robot application are described. The iconic shop floor robot programming environment is analyzed in terms of flexibility and usability.

The experimental implementation is described in Chapter 5. The workcell consisting of a PUMA 560 robot, a multiprocessor realtime subsystem, and a Macintosh-based application programming environment. This example of using the SKORP model in a complete system is described. Several skills that have been implemented and operations created from these skills are presented.

The implications of using a consistent model for robot system development are far reaching. Chapter 6 discusses the possibilities of standardization across many different robot platforms, programming at a high level from the shop floor, and other implications of using the SKORP model.

Conclusions and future directions for research in this multidisciplinary field can be found in Chapter 7.

Chapter 2

SKills-Oriented Robot Programming – SKORP

2.1 Introduction

As discussed in Chapter 1, the objective of SKORP is to develop a computational model for the efficacious development of industrial robot applications. Recall that four individuals (or teams) are responsible for creating these robot systems. This chapter describes the components of the SKORP model at a high level, specifically referring to the system architecture. The perspective of the robot systems analyst is emphasized in this chapter. The system programmer and the application programmer will be considered in the two subsequent chapters.

2.1.1 Related Research in Robot Systems Architecture

The concept of data encapsulation for robotic systems architecture has been discussed for more than a decade. In 1982 Mudge et al. [40] suggested that robot programming should be object-based. It is now clear that in these early works too many aspects of computing for robots were considered simultaneously. A consistent model for the implementation of all aspects of robot programming from machine level control to task planning was unsuccessfully attempted. The problem was identified and the general direction known, but researchers repeatedly attempted to force all aspects of robot programming into a single computing box.

The work of Zhang [22] was mentioned briefly in the previous chapter. He developed an interesting combination of a realtime open architecture robot controller and a high-level pattern-based language interpreter. The emphasis of his work is to create a Robot Operating System (ROS) that provides for multitasking at all levels. All of the tasks of creating a robotic operation are implemented using this operating system, including the motion control, teaching locations, interface to the keyboard, and file management. The issue of abstraction was recognized in his high-level "Open-Style Robot Programming Language." In this language the programmer can define macros and redefine the syntax of the language, but the integration of all the elements of the robot architecture under ROS prevents an effective separation of the various tasks. The highest level of programming still requires a skilled programmer working in a textual language.

Sensors now play a more important role in the model of the system architecture. Meijer et al. [41] proposed a system architecture that separates the sensing modules from the task planning, environment modeling, and control modules. They emphasize

the monitoring, diagnosis, and recovery from faults in the operation that require the original plan to be revised.

Much of the work on robot systems architecture has been centered around the creation and updating of world model information. For example, Roth and Jain [42] developed an elaborate method for creating a model of the *expected* environment that is updated and refined by sensor data.

Perhaps the most comparable work in the robotic system architecture literature to that presented here is by Ogasawara et al. [43]. They use environment models for telerobotic applications. They decompose the robot capabilities into skills and suggest the concept of using skills for telerobotics. The tele-operator can then invoke these skills without knowing about the underlying complexity. The skills are considered to be primitive motions only, and the teleoperator is expected to make the decision of when to invoke what skill. This approach is similar to the *supervised autonomy* of Burtnyk and Basran [44].

Industry is beginning to identify the complexity problem from a slightly different perspective. Glantschnig [45] identifies the problem as one of incompatible industry standards. An increasing number of complex components must now be interfaced into a manufacturing workcell. The sensors and machine controllers are often produced by manufacturers who assume that a particular operating system is being used and will supply only a vendor-specific interface. Glantschnig points out the necessity for doing a top-down design of these complex systems as opposed to the currently predominant, but unmanageable bottom-up development and testing. The solution suggested is to layer functionality and standardize interfaces. In this way the tools developed at each layer can be reused and interchanged effectively to create an architecture of communicating

systems. This general philosophy has been advocated by Sturzenbecker [16], among others.

The computer science community has provided some interesting ideas for solving problems that inherently require multiple disciplines. Zave [17] points out:

> "...each paradigm is too narrowly focused to describe all aspects of a large, complex system. For example, no one would want to program the database of an airline-reservation system using communicating finite-state machines, nor would anyone want to program its communication protocol using a data-definition language."

There is a comparable situation in robotic systems. A different computational paradigm is required for the high- and low-level programming. Ignoring this has been the single most important reason why the problems that were identified more than a decade ago have not yet been solved in any usable way. In 1988, Inoue [18] recognized that there would have to be multiple paradigms for developing a single robot system:

> "...to integrate robot system software, the real world interaction and the real time control are two major aspects which should be involved together. However at present it seems very difficult to unify them because the programming concept and methodology of each aspect differs too much."

This is the main impetus for the SKORP model. It is accepted that multiple paradigms are required and that these paradigms must fit together at a consistent and logical seam.

2.2 The SKORP Model - An Overview of the Mechanisms

Before describing the details in the SKORP model, it is necessary to give the reader an overview of the mechanisms and how they will eventually come together in the subsequent chapters.

2.2.1 Hardware Architecture

First, consider the proposed hardware architecture. There are no specific machines named, but Figure 2.1 depicts the general hardware architecture that the SKORP model requires. The separation of the responsibilities of the application specialist and the system specialist includes providing suitable computing hardware for both individuals. Separation of the realtime subsystem from the application programmer's workstation is imperative to the success of the model. Development tools and operating systems are necessarily different. The operator and the application programmer, require a workstation that has suitable development tools to create a user friendly interactive environment. Any of the popular personal computers on the market are suitable.

The realtime subsystem, although not specifically identified, is some sort of multiple processor architecture where the skills are implemented by the system programmer. Multiple processors are required to avoid computing bottle-necks that will always arise in sensor-based robot programming. This may be a transputer-based architecture or a more conventional multiple single-board computer configuration. All the sensors and machines are connected directly to this realtime architecture. The application

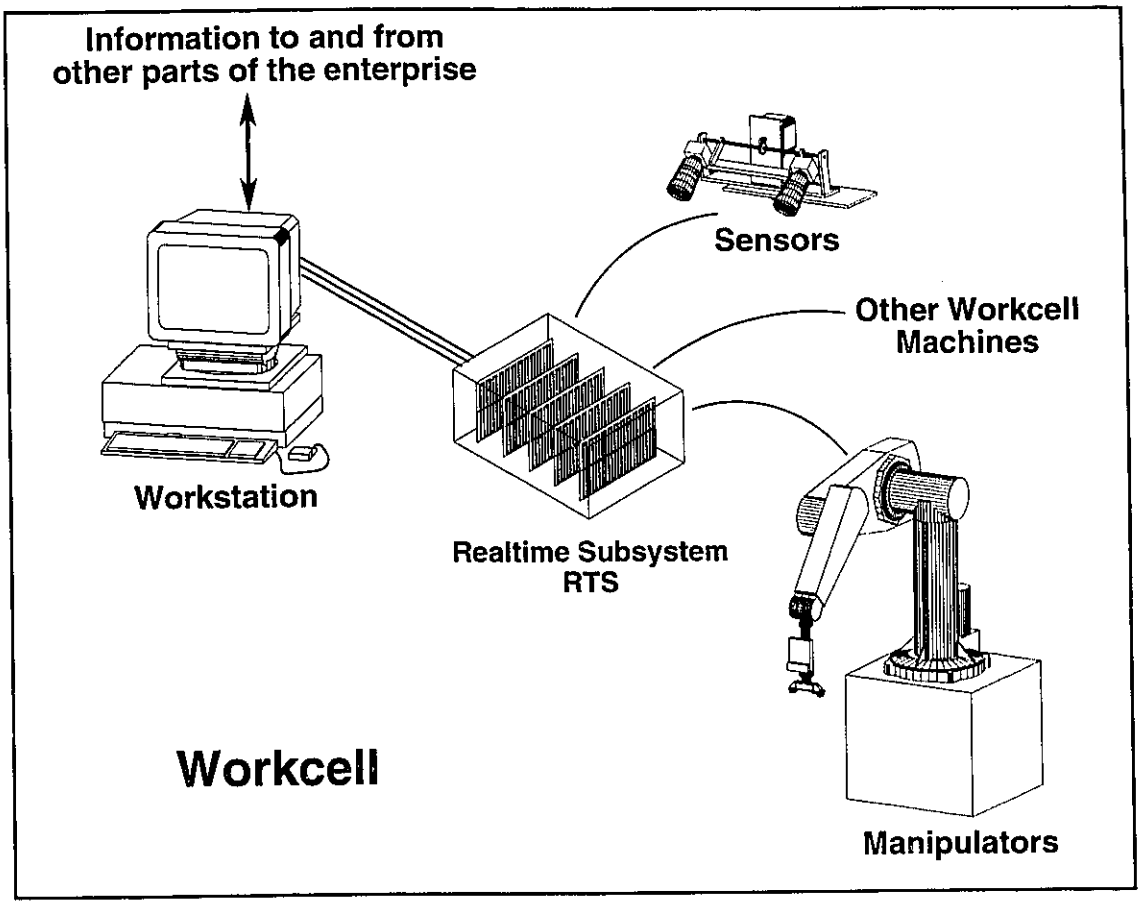

Figure 2.1: The separation of the realtime subsystem from the application programmer's workstation.

programmer's workstation is also connected to the realtime subsystem through either a serial or parallel link. The workstation, suited for higher-level programming, has all the tools for communication with the realtime subsystem and the other parts of the enterprise. Communication with the other parts of the enterprise may include various alarms, calls for human intervention in the process, and any production accounting data such as parts inventory, etc.

Figure 2.2 shows the relative responsibilities of each of the four types of users of the SKORP architecture. The robot systems analyst is ultimately responsible for the success of the application and has a design and supervisory role. His tools reflect these responsibilities.

2.2.2 Software Mechanisms

The following is an overview of how the four identified users of the SKORP model function with regard to the software development and use.

The robot systems analyst designs and specifies the required application. This description is in terms of robot skills, robot operations, environment configurations, and logical sensors [49]. An operation is a set of skills arranged by the application programmer using an iconic programming interface. The environment configuration is a description of the resources required for the operation.

The application programmer arranges the skills, which are represented as icons, in a window and links them together using a mouse. The parameters of the generic skill are changed to meet the requirements of the operations being created. For example, the generic skill *Rub* may be made into a specific skill *Polish Part* by changing the amount of force that the rub skill will apply to the surface and the speed at which the

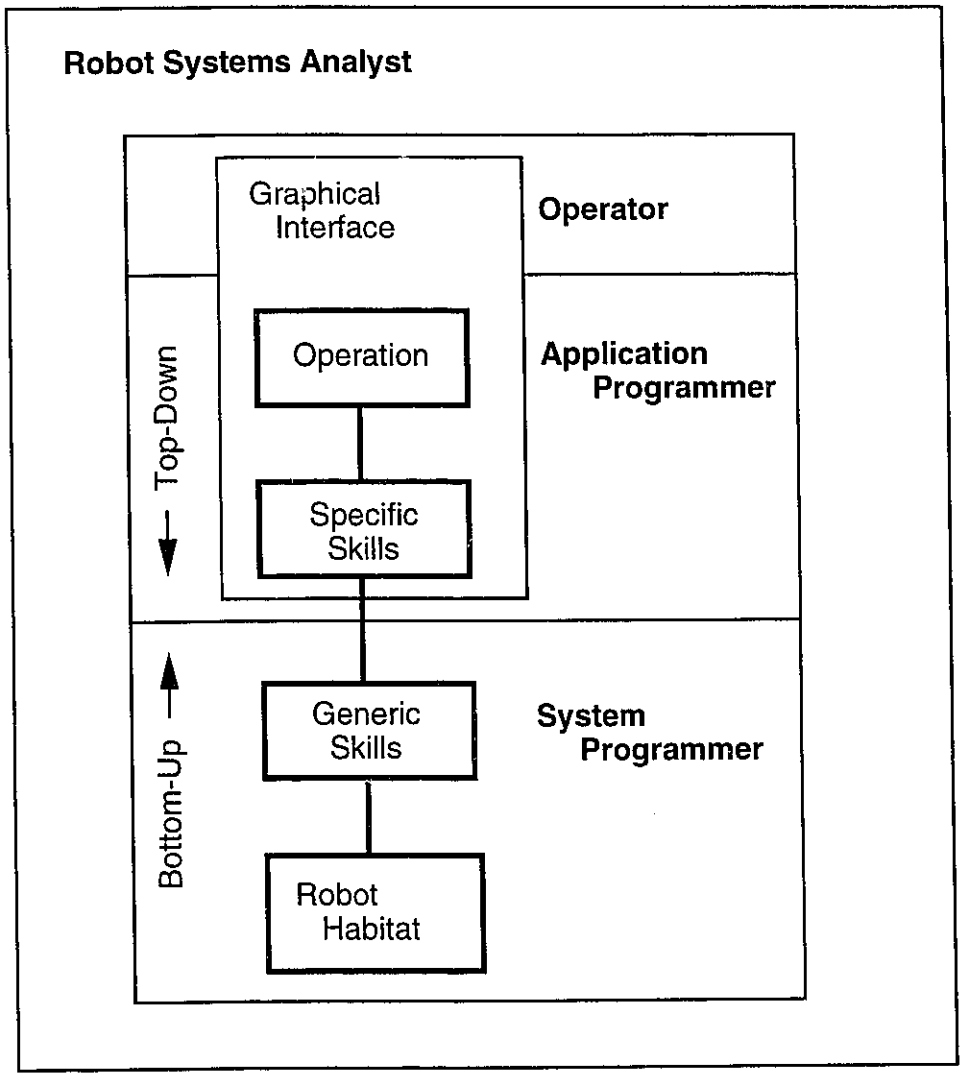

Figure 2.2: The relative roles of each of the four users of the SKORP model.

rubbing is performed. All of these modifications to create a specific skill from a generic skill are done from the graphical interface. The application programmer creates the operations in a top-down manner using the already developed and tested generic skills that were created bottom-up by the system programmer.

The system programmer, in the creation of the generic skills, uses tools that are described in detail in the following chapter. The SKORP model includes an object-based programming architecture for the development of generic skills on the realtime subsystem. The objects that are described include:

- skill and contingency methods
- logical sensors - sensor data interpretation
- sensor drivers
- machine controllers

The data in each of these objects is encapsulated, and can only be accessed by other objects using message passing. A precise functional definition of these objects and how they are created is reserved for the next chapter. They are introduced here so that the tools provided for the robot systems analyst will be clear.

2.3 Robot Systems Analyst's Specification Tools

As mentioned, the robot systems analyst specifies the robot operations, skills, environment configurations, and logical sensors. The operations are first described in an informal textual requirement. The skills are specified using templates for imple-

mentation by the systems programmer. The specification translates directly into the computational model for the skills.

2.3.1 Object Templates

In addition to defining the skills, the robot systems analyst specifies the elements of higher level objects including Environment Configurations, and Operations. Since Logical Sensors often become quite complex, requiring their own methods and using multiple sensors, they are also described in a template. All of these object templates contain slots for textual descriptions, data, and computational methods. The "method" is considered to be the *active* part of skills, drivers, and logical sensors. This distinction will become significant when the realtime computation is described formally in the next chapter. The design and specification process is simplified by this organization. Any instance of a SKORP system can be described using these templates, regardless of the details of the realtime subsystem and the workstation.

2.3.2 The Robot Skill Template

The central object in the SKORP model is the skill which is defined in terms of its properties, roles, relationships, and constraints. A robot skill is the capability of a robot to repeatedly accomplish any action that can be described unambiguously. The unambiguous description is in the form of a skill template. Since many of the modules in a skill can be reused in other skills, the template serves as a design tool that helps to organize the reuse of software. The template consists of textual descriptions, data, and software modules.

A Generic Skill Template contains

- GN Generic skill Name
- TD Textual Description of the skill
- PC PreConditions
- PT PosTconditions
- SD Sensor Drivers
- MC Machine Controllers
- LS Logical Sensors
- MD Model Data
- SM Skill Methods
- CD Contingency Descriptions
- CM Contingency Methods
- PE Parameter vector for Execution
- CP Completion Protocol

A generic skill template is thus described as:

$$GS = \{GN, TD, PC, PT, SD, RM, LS, MD, SM, CD, CM, PE, CP\}$$

where the generic skill name is the name by which the skill will be known to the applications and systems programmers.

The textual description of a skill is written by the robot systems analyst and used by the systems programmer as an overall objective in the creation of the skill. It is

also used by the application programmer to determine if this is the skill required in the current circumstances.

Preconditions are a textual description of what must exist in the robot habitat in order for this skill to function properly. Postconditions are the textual description of how the habitat exists after the skill is completed. Preconditions and postconditions are not used computationally.

The sensors and robotic machines are not specified by manufacturer and model if this skill can be executed by many different devices. This will become significant in the discussion on standardizing robot skills across multiple robots and sensing devices.

Some skills will require model data. These data are not specifically defined because different skills may require completely different information on the habitat.

Skill methods are the actions that actually carry out the skill. Many of these methods will themselves be objects that can be used by many skills. The methods may be directly involved in achieving the skill or they may be peripheral, such as a required collision avoidance method that is not directly part of the skill. These methods may be described in a general way by the robot systems analyst, but it is the responsibility of the systems programmer to create these methods.

Contingencies are the possible events that would prevent the skill from being completed in the expected way. These are textual descriptions of events that should be considered during the creation of this skill. The robot systems analyst determines what contingencies are reasonable to expect and how these events will be handled.

Contingency Methods are actions associated with the contingencies; they are to be executed in the event that a contingency occurs. The robot systems analyst describes these methods textually for creation by the systems programmer. Contingencies are

similar what is comonly referred to as *exception handling.* The term "contingency" is used to explicitly differentiate this concept from error recovery or fault tolerence. Contingencies are events that may be expected and methods can be provided to deal with these situations. We consider errors or faults to be events that are not expected and the methods to deal with these occurences are more general, such as halting the robot and reporting a fatal error.

The parameter vector is the list of variables that can be modified to create a specific skill from the generic skill. This will be the application programmer's access to modifying the behaviour of the skill. The most common parameter is likely to be the speed of the robot as it carries out a skill. Other parameters might include location names, and numeric parameters that are given in the terms appreciated by the application programmer, such as the amount of force to be applied in a force based skill, or the size of the testtubes in a laboratory automation skill. The inclusion of these parameters must be carefully considered. If too many interrelated parameters are placed on a skill it will become unwieldy, but if there are too few the skill will be either trivial or inflexible.

The completion protocol is a message that the skill returns upon completion of the skill. It indicates success, i.e., normal completion, a contingency has been encountered, or that the skill ended in an unrecoverable error.

Each skill does not require all of these attributes. The template is an encompassing set of attributes for all robot skills. Specific skills have some subset of these attributes.

The textual descriptions are not intended to be used computationally. The state of the robot habitat is continuous and multidimensional. A formal description of the pre-conditions and postconditions would make the invocation of the skills context sensitive

introducing undesireable complexity for the application programmer. This is discussed further in Chapter 4.

A simple database application can be created to store and retrieve the skill templates. An example of a generic skill *Approach to Touch* is given in Figure 2.3 to clarify this skill definition concept.

2.3.3 System Environment Configuration

A System Environment Configuration is the description of the current environment available for executing operations. The availability of both computational and physical resources are listed.

- SD Sensors
- LS Logical Sensors
- RM Robotic Machines
- GS Generic Skills

$$SEC = \{SD, LS, RM, GS\}$$

Not all systems will have the same set of resources. This template can be used to automatically ensure that the programmer is not calling upon unavailable resources. It differs from the skill template in that it is used computationally in the SKORP system to ensure that the required resources are available for the execution of a particular operation.

Generic Skill Template

Generic Skill Name:	Approach to touch
Textual Description:	Move the end effector in the positive Z direction of the tool until some object is encountered.
Preconditions:	The end effector should be near a surface.
Post Conditions:	The end effector is touching the surface.
Sensors:	Force sensor for end effector
Robotic Machines:	Revolute or prismatic robot capable of motion in the end effector coordinate system.
Model Data:	None.
Logical Sensors:	Z-force that returns the force at the end effector in the Z direction.
Skill Methods:	Read the Z-force at the end effector and continue to move in the positive Z direction in tool coordinates until a force is encountered.
Contingency Descriptions:	C_1 The end effector moves a large distance and encounters nothing.
Contingency Methods:	C_1 Return to the original location of the end effector.
Parameter List:	P_1 Approach Speed P_2 Maximum distance to move before declaring C_1
Completion Protocol:	0: Normal completion - Post Conditions exist 1: C_1 was encountered - return to original location. 2: Any other non-recoverable error occurred

Figure 2.3: A generic skill template for *Approach to Touch.*

2.3.4 Robot Operations

A robot operation consists of an operation environment configuration template (OEC) and a set of robot skills S_i arranged to accomplish a specific task within the defined structure of the habitat.

$$O = \{OEC, S_1 \ldots S_n\}$$

The operation environment configuration must be a subset of the system environment configuration for the operation to execute. The OEC can be created automatically by extraction of the requirements from each of the skill templates. Comparison to the system environment configuration determines the availability of the resources for the operation. This is useful when the application programmer includes a skill that requires an unavailable resource.

2.3.5 Logical Sensors

A logical sensor [49] is a software module that uses one or more sensor drivers to extract a specific vector of information from the sensor data. The logical sensors are included at this level to reduce duplication of effort in the extraction of useful information from the sensor data. The reuse of logical sensors is an important aspect of reducing the cost of skills development.

A Logical Sensor Template contains

- TD Textual Description of the information provided
- SD Sensor Drivers Required
- LM Logical Sensor Methods

- PV Parameter Vector for returning information

$$LS = \{TD, SD, LM, PV\}$$

Each logical sensor is written to obtain a specific vector of information from the habitat. Logical Sensor Methods are the algorithms used to interpret the sensor data. Usually there is only one. More than one sensor driver may be required if the sensor data are to be fused, or integrated. The parameter vector is sent to the skill when it is filled in.

2.3.6 The Operation Storyboard

Combining the skill descriptions, preconditions, and postconditions for each of the skills in an operation creates a storyboard for the robot operation. The robot systems analyst can use this storyboard to consider the operation at a high level of abstraction and to describe it to the operators or to a potential customer. When an operation is created by the application programmer, the storyboard should be returned to the robot systems analyst to verify that the operation performs the originally intended function. Except for the name of an operation, this is the highest level of abstraction that describes an operation.

2.4 Conclusion

The hardware components of the SKORP model separate the working domains of the system programmer and the application programmer. The robot systems analyst is responsible for specifying the skills and the operations using the templates. The skills

will be implemented by the system programmer, as described in the next chapter. The operations will be created by the application programmer as described in Chapter 4.

Chapter 3

Realtime Subsystem

3.1 Introduction

This chapter describes the organization of the layered architecture for the realtime subsystem in the SKORP model. The creation of skills on the realtime subsystem and the modules that make up the skills are presented in detail. As well, the tools provided for the system programmer to create the skills are described.

In sensor-based robotic systems the most critical issue in the creation of skills is meeting realtime deadlines. These deadlines are the required response times for a variety of events occurring in the system. Events generated externally are detected by sensors. Communicating software modules may also generate events internally. In general, events may occur periodically or sporadically and may require concurrently executing responses. Several tools have been created for the design of such complex event-driven systems. (See [50], for example.) A general purpose realtime development system soon becomes extremely complicated, often more complicated than small or restricted realtime applications. When using the design tools becomes more complex

than the product design, there is a negative marginal utility in employing these tools.

The realtime systems design problem is simplified here because the focus is on sensor-based robot programming. Robot operations are specified as a sequence of skills selected and parameterized by the application programmer. Since it is known that only one skill will execute at any time, there are a limited number of possible events that require servicing; This simplifies the problem further.

A *coordination language* [51] is a valuable tool for a systematic design of the individual robot skills. Such a coordination language is not a computational language but a formal way to represent and analyze the coordination of the computational modules in event-response systems. In this case, the modules that make up an individual robot skill must be coordinated. In fact, the coordination language is independent of the computational language and is used only to specify the interactions of the software modules with each other and with external devices.

The coordination language used here is based on the Process Activity Language (PAL) [52]. PAL has a graphical representation called Process Activity Network (PAN), and an associated timing tool called Activity Timing Chart (ATC). These activity coordination tools were developed by Krieger [53] for use in realtime multiprocessor system design [54] and are more recently being adapted for specific application areas, especially manufacturing and sensor-based robot control.

3.1.1 Related Research

The realtime subsystem presented in this dissertation is not a general-purpose real-time system. The process of creating a program for robot control is not the same as programming a general-purpose computer. The physical interaction with the robot

habitat is the most difficult and important aspect of realtime robot programming.

There have been two main approaches to creating robot software development tools. The first is to include everything in a single development tool, i.e., provide a tool that is used for all levels of the development. The second is to use off-line programming to try to develop operations on the computer without the robot. The objective is to reduce the time that the robot is out of production by creating the operations off-line that will be modified on line during installation.

Both these approaches present difficulties [55]. General-purpose languages, such as C or C++[56, 57], ADA [58], and PASCAL [10] have been proposed, often with libraries of robot control routines attached. These are excellent tools for system programmers, but provide no mechanism of dealing with the complexity of realtime robot programming. Using tools that are monolithic, i.e., provide a single environment for the development of both application software and realtime software, has the disadvantages discussed in the previous chapters, particularly the complexity issues. One way this has been dealt with is to create modules and force them together using various formalisms. Petri nets, and the variation of Petri nets called GRAFCET [59], have been popular for this [60]. The issue is still the same. A single environment for all levels of application development ignores the multidisciplinary nature of the problem.

Developing applications off line, and moving them to the robot habitat for fine tuning are equally unsuccessful, since habitat-specific issues are the most difficult and often these are unknown until encountered. Tools devised to solve complex problems must be founded on a method to address the most difficult issues for that class of problems.

The compromise is to develop applications on line, but to work with modules that

are already developed and tested. This has the combined effect of reducing the down time of the expensive robots and workcells and ensuring that the robot operations function properly in the intended habitat with no surprises when an off line system is moved into production.

Tools for the development of the individually tested robot modules are rarely described in the literature. Some attempts at this have been quite *ad hoc* with respect to the hardware and operating systems [61]. In this chapter, tools are presented for the creation of realtime skills and the modules that make up these skills.

3.2 A Layered Software Architecture

Many of the advantages of object-oriented programming are employed in this layered architecture design. Most important, it allows the programmer to arrange software so that it reflects the organization of the world with which it is intended to interact. This is an important consideration for complex embedded systems that must handle asynchronous events in hard real time. The design of such an architecture must also consider the eventual target hardware. It is clear that multiple processors are required [20]. Although modern single processors have very fast computation speeds, the primary issue is not the speed of computation but whether or not computation is possible at the time it is required to meet hard realtime deadlines. It is not claimed that the SKORP model is "object oriented" because some of the attributes that define "object-oriented" programming are not included, particularly inheritance properties. The computational model is based on encapsulated objects arranged in layers and is considered "object-based."

The computational model described here is not specific to any hardware. The ex-

perimental implementation is done using a Harmony[1] operating system with four 68020 processors, but the idea of implementation on a RISC architecture, particularly transputers, is being considered. The Harmony version of the SKORP realtime subsystem is described further in Chapter 5, where the experimental implementation is presented.

A generic skill template is comparable to a class of objects. Generic skills are objects in this class. Specific skills are objects that inherit the attributes of the generic skill but are parameterized to meet specific objectives. This allows one generic skill to be written, tested, and then used with a variety of parameter-setting combinations. Development of specific skills is progressively easier as libraries of these skills are built up over time, and each can be modified to create a similar skill. Specific skills are parameterized versions of generic skills created at the robot programmer's interface, as described in the next chapter.

In the SKORP model, objects communicate via message passing. They are invoked, invoke other objects, send requested information, and indicate that they are completed via message passing. The data in the objects is encapsulated, i.e., the data in the skills cannot be manipulated by any other object except through message passing. This enforces modularity and consistency in the software design.

Figure 3.1 shows the layered architecture. There are two types of abstractions. The highest level of abstraction is found in the Operations, followed by the more detailed layers of Specific Skill and Generic Skill. The second type of abstraction deals with the information from the robot habitat. The robot habitat is queried at the highest abstraction by a skill that invokes a logical sensor [49] that in turn uses a sensor driver. The detail of interfacing with the sensor is encapsulated in the sensor driver and the

[1] Harmony is a registered trademark of the National Research Council of Canada.

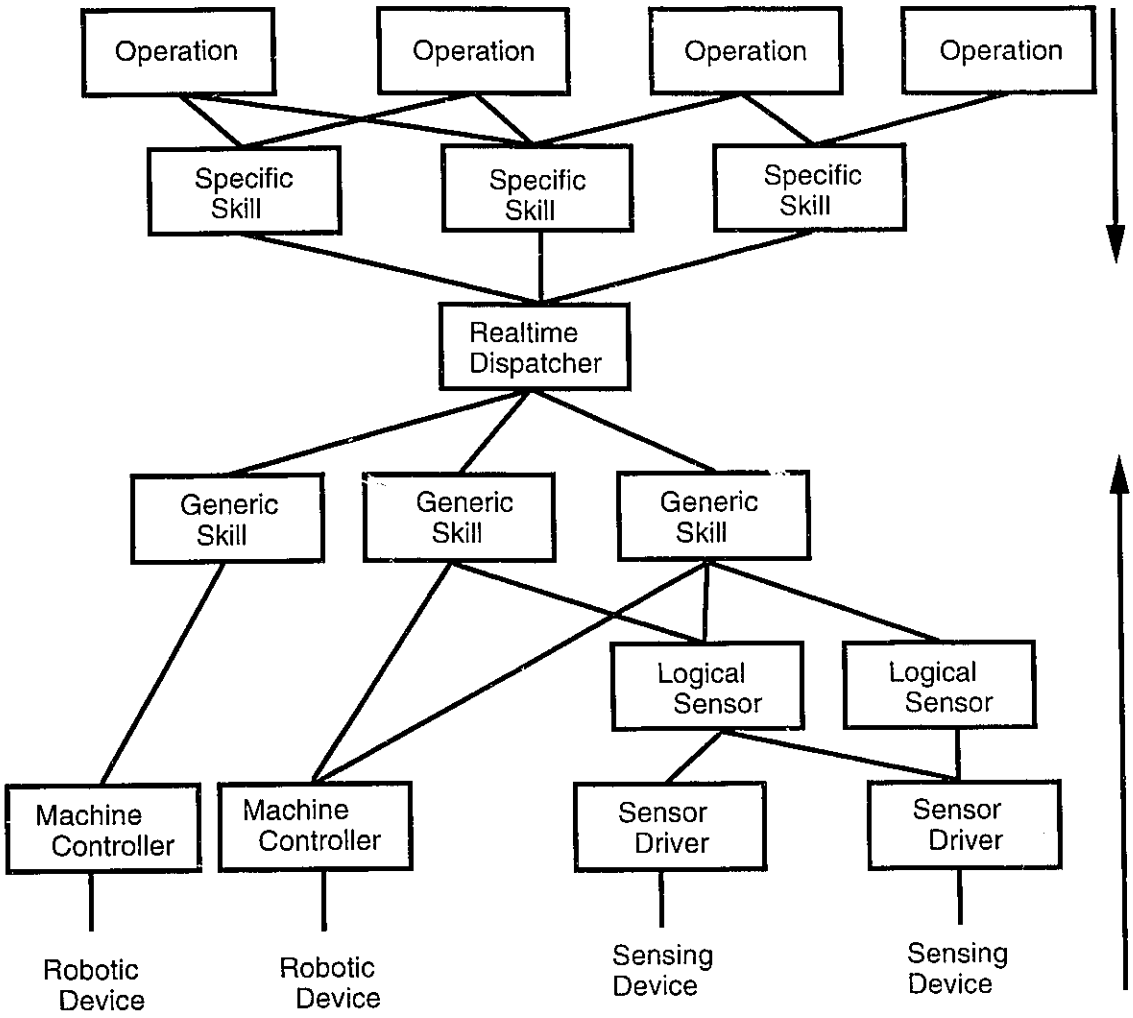

Figure 3.1: A schematic design of how the major objects in a SKORP architecture are layered. Central to the design is the Generic Skill.

interpretation of the sensor data is encapsulated in the logical sensor.

From Figure 3.1 it is easy to see how multiple sensors can be used in an individual skill. If the requirement is to combine the information at the level of the data, i.e., sensor fusion, this is done at the logical sensor level. If multiple sensors are used for a single skill but the information is combined after the sensor data have been interpreted, i.e., sensor integration, this is done in the generic skill layer.

The flow of information throughout the model is shown in Figure 3.2. Note that there can be multiple instances of logical sensors, sensor drivers, and machine controllers in this control loop. Objects are restricted to communication with other objects of specific types. For example, only Logical Sensors can communicate with Sensor Drivers. New modules can be added using variations of the communication mechanisms that exist for previously developed modules of the same type.

The control loop is always closed in the physical habitat of the robot. This is where the uncertainty in the feedback loop occurs. This distinguishes physical control from general-purpose realtime systems, since the complex physical interaction is the most difficult aspect of the entire system.

Realtime feedback of sensor data requires that the implementation of the skills allows access to both the sensors and the machines with a minimum of overhead computation. The levels of robot operation development, including the responsibilities of the applications programmer and systems programmer, are also indicated in Figure 3.2.

Central to the SKORP model is the generic skill. At the higher layers, the generic skills available are used to create specific skills, and operations. At the lower layers, the sole function of the objects is to contribute to the execution of the skill.

Skills are developed bottom up, which ensures functionality at the skill level. Oper-

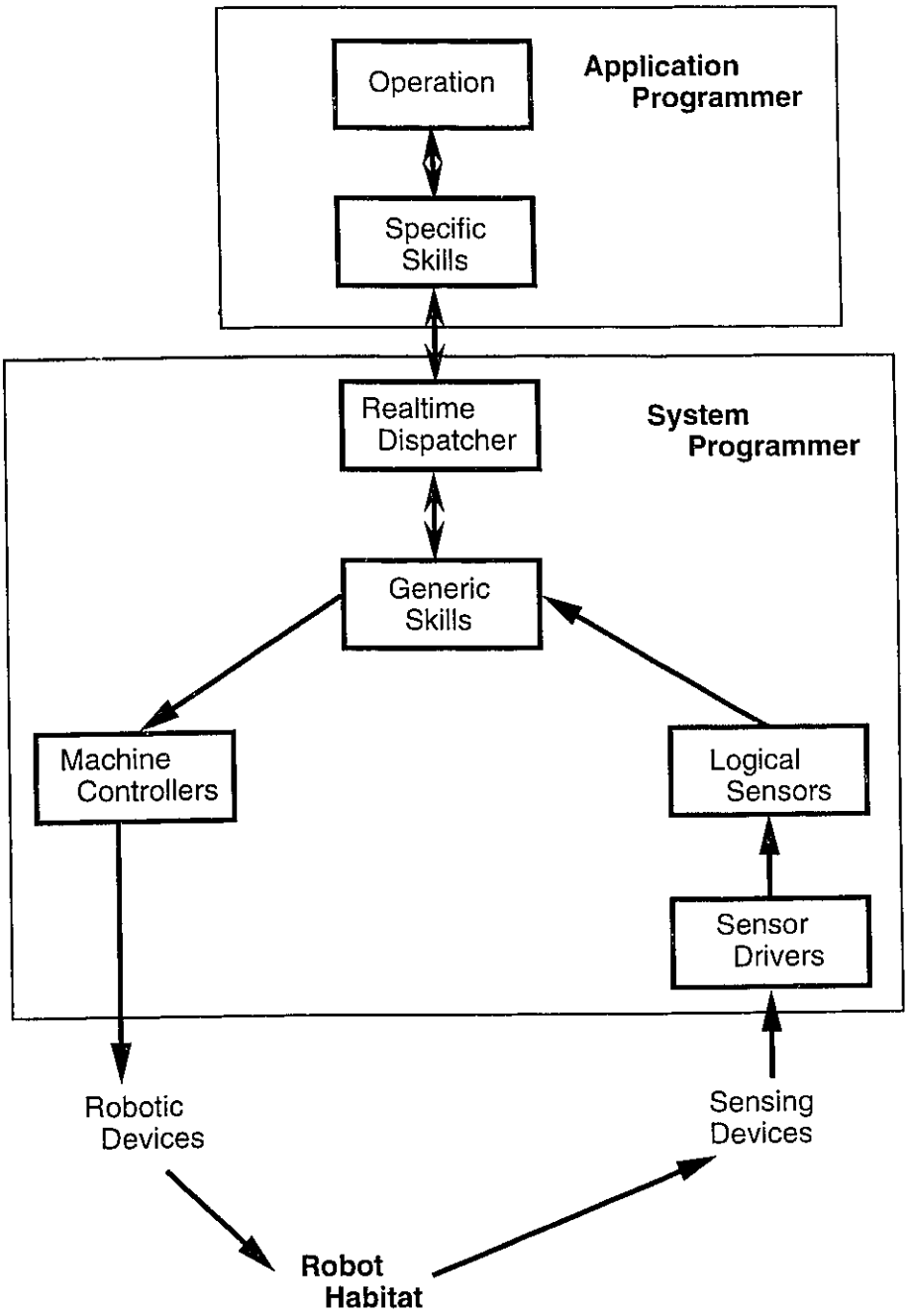

Figure 3.2: The information flow throughout the SKORP computational model. The layers created by the robot programmer are distinguished from those created by the systems programmer.

ations are developed top down; the application programmer examines the requirements and assembles the skills. Task planning is done explicitly by the robot programmer, as opposed to automated task planning, which has been proposed in the literature.

The computational model is functionally restrictive to enforce modularity and provide extendibility, reusability, accretion, and easy decomposition for design and debugging. It is specific to programmable machines using sensors. The SKORP model does not restrict the type of methods that can be included in a skill. Collision avoidance, path planning, singularity prediction, etc., as required by the skill can be implemented within the computational model. These additional skill methods are the responsibility of the system programmer. Also, there are no restrictions on the type of model data or the hardware architecture used in the implementation of the SKORP model. It is possible to use this computational model on any computer designed for realtime programming, although the realtime requirements will dictate a minimum number of processors.

3.3 A Coordination Language for Skills Development

A robot operation is defined as a set of skills that execute exclusive of each other according to sequential and branching constructs as determined by the application programmer. Since only one skill is executing at any time, the number and type of events that can occur during the execution are restricted. This means that the realtime requirements of a single skill will never interfere with those of another skill. The coordination language will therefore deal with only the events that are generated for

and by the modules that make up a single skill. Each of the events can be characterized as one of the following:

- A message representing a request is received.
- An expected external interrupt is received from a machine or sensor.
- An unexpected external interrupt is received from a machine or sensor.

The modules can generate an event by sending a message that represents a request, e.g., *go and get some sensor data, extract some information from the sensor data*, or *move the robot to a new location.* The types of software modules that make up a skill are designed around these events. Each of the software modules can be characterized as one of

- Sensor Driver
- Logical Sensor
- Machine Controller
- Skill Module
- Contingency Module

It is reasonable to restrict the type of module that can send requests to another type. For example, since the function of the Sensor Driver modules is to collect data from a sensor, it has no requirement for communication with a machine controller. This requirement-based communication between modules creates a layered and hierarchical architecture, where each module can communicate with only the connected modules

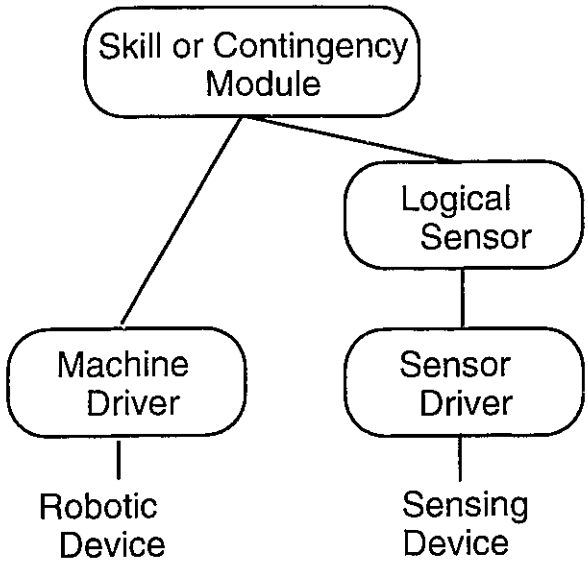

Figure 3.3: Organization of communicating software modules.

in the layer above and the layer below (see Figure 3.3). Parallel execution is also addressed at this level of design. Skill modules and contingency modules are shown in the same block because they always execute exclusive of each other, i.e., the skill is either executing its expected logic or the contingency logic. It is feasible for modules from all of the blocks in Figure 3.3 to be executing simultaneously on different processors.

3.3.1 PAL, PAN, and ATC

The coordination language PAL (Process Activity Language) is useful for specifying the computational requirement for skills, as this relates to the interaction of the modules and the external devices. There are two parallel representations: PAN (Process Activity Networks), which is a direct translation of PAL. A timing tool called an ATC (Activity Timing Chart), is used to represent the time requirements for interdependent modules.

PAN is used for drawing the modules of the skill to determine both their coordi-

nation during execution and which modules can be executed concurrently. It is also useful as a sketch to verify the high-level logic of the skill and create the PAL representation. PAL is used as a high-level coordination language. Before the implementation in the computational language the timing constraints must be verified using an ATC. This step requires information about each module, specifically the execution time or, if it is variable, the upper bound of the execution time. This requirement cannot be avoided in the design of realtime systems. The ATC can then be constructed from the PAL description. This is not a single pass process. During construction of the ATC it often becomes apparent that meeting the timing requirements requires changes in the original PAN expression. Although this may appear to be an inefficient design tool, an iterative design process seems to be the best way to ensure that both functional and timing requirements can be met. A simulator for "What if...?" design has been implemented using the PAL/PAN methods [62].

The PAN modules are represented as single upper case letters and global variables referred to as *process conditions* are represented as c_i. The graphical notation for PAN, shown in Figure 3.4, is modified from [53]. Note the differences between the Case and Concurrent constructs are the bold vertical lines that surround concurrent activities and the first item within a Case is always a condition. With these constructs it is possible to give structured expressions of functions executing on multiprocessors.

The modules use the client-server model for coordination. Each module is started when it receives a message that requests it to execute and ends by returning a message, usually including useful data, to the requester. The data in each module are encapsulated and only available to other methods via message passing.

PAL is written similar to high-level languages but, since it is used only for coordi-

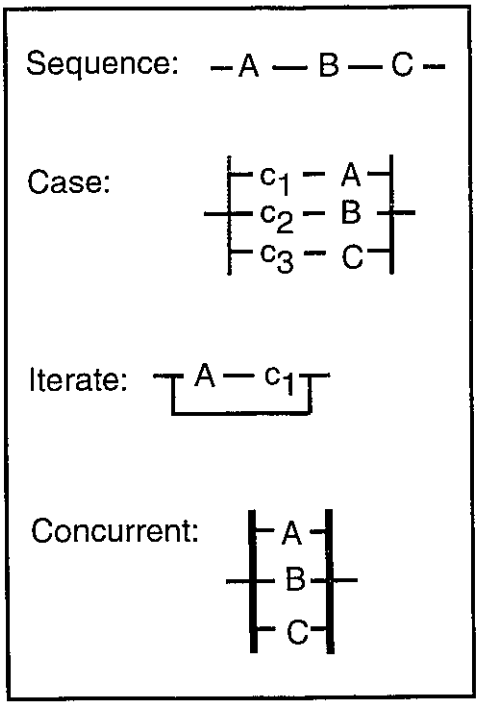

Figure 3.4: Graphical notation for Process Activity Diagrams.

nation, there is very little syntax. The entire syntax of PAL, as modified from [53], is as follows:

WTM(): Wait for a message

TXM(): Transmit a message

SPC(): Set process condition

RPC(): Read Process Condition

CASE: Similar to all Case statements.

A: Execute Module A

REPEAT....UNTIL() To express iteration.

CONCURRENT: Execute concurrently on separate processors.

For CASE, REPEAT, and CONCURRENT that affect a block of other statements,

the blocks are identified with BEGIN...END and are indented in the usual way.

The Activity Timing Charts are approximately equivalent to Gantt charts. Their function will be discussed in the next section.

3.3.2 An Example – *Approach to Touch*

Approach to touch is a sensor-based robot skill that can be described as follows: Move the end effector in the positive Z direction until a surface is touched or until the robot has moved a specified distance without touching a surface. The template for this skill was given in Chapter 2. This textual description of the skill is provided to the application programmer who is responsible for parameterizing the skill, i.e., specifying the speed of the approach and the maximum distance the robot should move before executing the contingency module.

3.3.2.1 A PAL, PAD, and ATC Description of *Approach to Touch*

Approach to Touch is a very simple sensor-based skill that can be used to explain the mechanisms of the coordination language but cannot adequately demonstrate the merits of using such a method when multiple sensors and mechanical devices are required. There are five software modules of interest in the design of this skill: the skill method SM1, the logical sensor module LS1, a sensor driver SD1, a machine controller MC1, and a contingency method CM1. (Refer to the generic skill template in Chapter 2.) There are two process conditions; pc-force and pc-reach. Both are Boolean variables. If pc-force is true, the logical sensor has detected that the robot has touched a surface, and if pc-reach is true, the maximum distance to attempt this touch skill has been reached. These are the two possible stopping conditions for the skill.

In the skill method SM1, a request to begin the skill comes from the dispatcher. SM1 iteratively requests the current Z force from the logical sensor and sets the process condition to TRUE (touching) or FALSE (not touching). If pc-force if FALSE, SM1 will send a request to the machine controller MC1 to move the robot and wait for the current position to be returned. SM1 will set the value of pc-reach to true if the maximum distance has been traversed. If either pc-force or pc-reach is true, this loop will exit and either send the successful completion protocol or execute the contingency module. If the contingency module is executed, a failure completion protocol is sent to the dispatcher.

The logical sensor LS1 requests force data from the sensor driver SD1 and extracts the force in the Z direction. The sensor driver can read new force data every 6 ms. The contingency module CM1 simply returns the end effector to its original position. The machine controller MC1 receives an external request for motion of the end effector every 28 ms and must respond to this request immediately.

Consider the PAN description. It is necessary to execute the robot control in parallel with the sensor data collection and interpretation. Although in this example the robot control will take place every 28 ms and the sensor will provide data every 6 ms, it is not desirable to force these two activities to synchronize by using a clock to delay one or the other, because the next robot that requires this skill will probably not have a 28-ms control loop. The objective is to implement the modules in the most general way possible, making them independent of each other. For example, if the software is ported to another robot, the machine controller modules would be replaced, but the remaining modules should not require modification. The PAN description of the method SM1 is shown in Figure 3.5 and the PAL equivalent can be found in

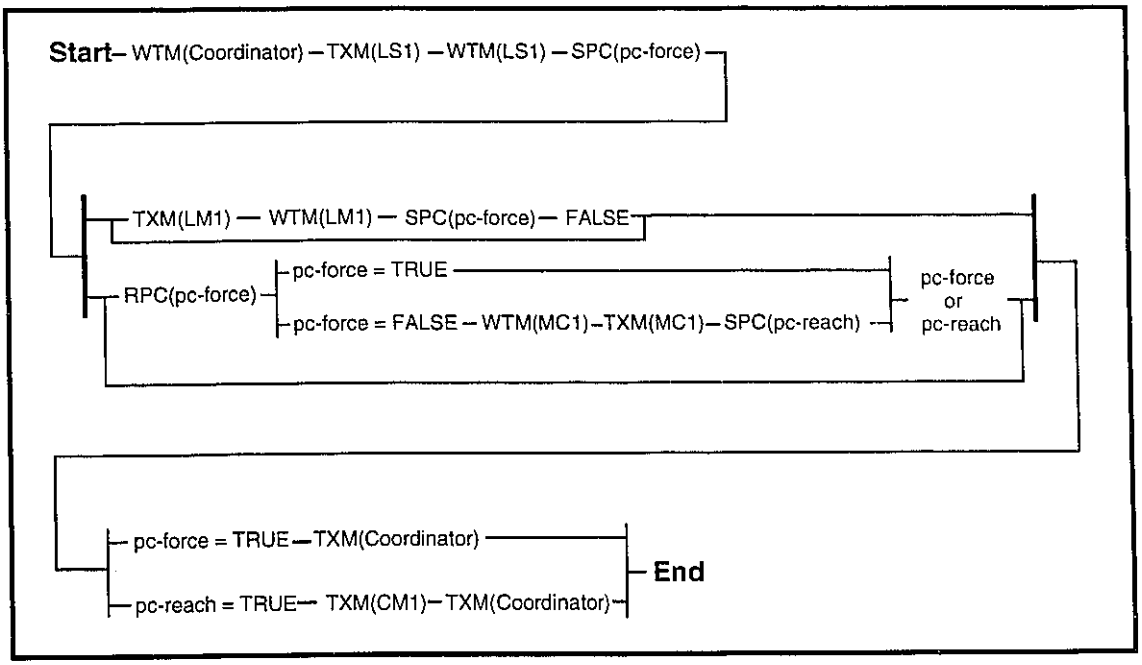

Figure 3.5: A PAN description of the skill module SM1.

Figure 3.6. From the concurrent activities in the PAN it is clear that this skill requires two processors.

The Activity Timing Chart ATC representation of the methods is shown in Figure 3.7. The sensor data collection should be done in parallel with the robot controller communication as shown in the timing charts. Because these events are not synchronized, which should be delayed must be decided. The robot controller communication occurs each 28 ms, while the sensor data collection requires only 6 ms in this case. It is necessary to communicate with the robot controller with minimal delay, so it was decided to use slightly stale sensor data for the robot control. As can be seen in the ATC, the robot communication will occur every 28 ms with no delay, but the sensor data may be up to 6 ms stale. If a single processor were used with no parallel activities, the delay between robot communication would be as high as 12 ms because a

Skill Module: SM1

```
BEGIN
  WTM( Coordinator )
  TXM( request Zforce from LS1 )
  WTM( Zforce from LS1)
  SPC( pc-force)

  CONCURRENT
  BEGIN
    REPEAT
      TXM( request Zforce from LS1 )
      WTM( Zforce from LS1 )
      SPC( pc-force )
    UNTIL( forever )
  END

  BEGIN
    REPEAT
      RPC( pc-force )
      CASE
        ( pc-force = TRUE ): Do Nothing
        ( pc-force = FALSE ):
          WTM( motion request from MC1)
          TXM( motion to machine controller MC1 )
          SPC( pc-reach )
      END CASE
    UNTIL ( pc-force OR pc-reach )
  END
  END CONCURRENT

  CASE
    ( pc-force = TRUE ):
      TXM(Completion Protocol 0)
    ( pc-reach = TRUE ):
      Execute CM1
      TXM(Completion Protocol 1)
  END CASE
END
```

Figure 3.6: A PAL description of the software module SM1 for the *approach to touch* skill.

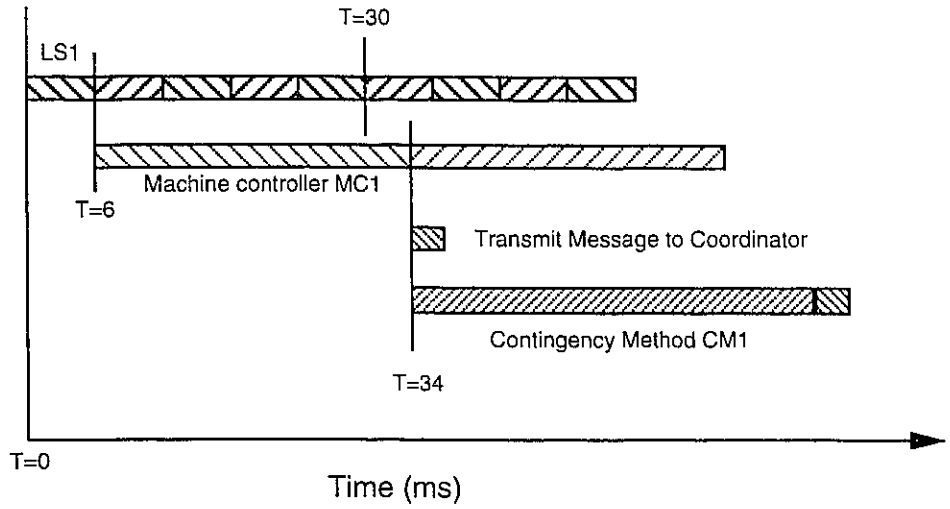

Figure 3.7: An Activity Timing Chart representation of the *approach to touch* skill.

complete set of sensor data must be collected. The 12-ms delay could occur if the data collection started immediately after the sensor began to deliver a complete set of data; this effectively wastes the first 6 ms and requires another 6 ms to collect a complete set of data. The delay of 12 ms for robot control using force sensors is unacceptable and unnecessary in a correct design. The ATC in Figure 3.7 shows the unlikely scenario where the sensor is read once and finds the robot is not touching an object, the robot is moved, and when the sensor is read the next time the robot is touching an object. Using the timing chart we can verify all the possible time constraints in this simple example.

3.4 Conclusion

Systems that are used for sensor-based robot programming must be designed with the most difficult aspects of the task in mind from the outset. Success depends on the successful integration of the sensors into realtime computing systems. The layered

system architecture presented for SKORP provides a consistent model, which addresses the issues of homogeneity and modularity which are required by industry, for the modular development of robotic applications.

A coordination language for the design of sensor-based robot skills has been presented. The realtime coordination problem has been restricted by the specific characteristics of robot controllers and sensors. The design problem has been further reduced by dividing the robot operation into discrete functional units called skills and dividing the skills into software modules of specific types. The specification of the software modules and their interaction with other modules is an iterative process that requires methods and tools that specifically address the most difficult issues in this type of system design. Response times to events generated internally and externally must be guaranteed for all practical situations at the design stage of skill development.

Chapter 4

Iconic Robot Programming

4.1 Introduction

The concept of programming robots using icons can be considered part of a wider trend to remove the details of complex operations from the users. In the consumer markets this has been seen with the introduction of power steering and anti-lock brakes on cars. In these cases the user communicates a request to the machine and some acting agent carries out the request without the user being aware of the details including how (or even if) sensors were used. The robot application programmer uses icons to communicate requests to the robot system. Using the icons as agents significantly reduces the complexity of programming robot actions when compared to traditional text-based programming languages. This allows the application specialist to concentrate on the applications as opposed to how the requests can be communicated to the robot system.

In this chapter the role of the application programmer is described, the non-textual programming methods previously developed are discussed, and the SKORP method of iconic robot programming is presented.

4.1.1 The Application Programmer

It is well known that an accurate description of the user is important when creating a user interface [63]. Only when the description of the user is well defined can a tool be designed for his use. When designing the interface for the robot application programmers one must combine the description of the user with the objective of minimizing the complexity of the interface.

The application programmer will be described in terms of the capabilities that he is expected to *use* while functioning in the application programming environment and the problems he is expected to solve. An important design issue is what he *should* be doing in this environment, as opposed to what he *can* do. This focus on the user is cited by Gould and Lewis [64] as the first principle of designing user interfaces for *usability*:

> "... designers must understand who the users will be. This understanding is arrived at in part by directly studying their cognitive, behavioral, anthropometric, and attitudinal characteristics, and in part by studying the nature of the work expected to be accomplished."

Many robot application programmers are capable of programming in modern textual computer languages, but examination of what is expected of these individuals working on a shop floor shows that textual languages are not appropriate tools. Programming in a textual language requires more concentration than can realistically be expected of anyone working on the shop floor. The tools themselves must not present an intellectual challenge.

Consider the physical working area of the application programmer. On the shop floor he is very likely standing up surrounded by numerous distractions associated with

the application being addressed and from other unrelated activities. These factors must be considered when determining the *attitude* of the users who require a usable tool that suits the shop floor milieu.

The application programmer must be provided with tools to completely define an application given a robot habitat and the objectives of the application. He must provide a solution to a range of possible situations within the restrictions of the application.

The application programmer is described as an individual who must use the following limited knowledge and capabilities:

- familiarity with interactive computer applications
- ability to use a point and click device, e.g., a mouse or trackball
- understanding of the robot capabilities
- knowledge of the physical process required for the application
- understanding of the concepts of skills-oriented robot programming

It is important to make the distinction between a human-robot interface and a human-computer interface. In classical human-computer interface design, the only unpredictable element is the human user. In a human-robot interface there are actually two interfaces: one from the human to the computer and one from the computer to the robot. (See Figure 4.1.) There are now two unpredictable elements in the system; the human and the physical robot environment itself. This is another argument for reducing the complexity in the graphical human-robot interface.

The human-robot interface must appear seamless to the user so that the amount of time the programmer spends thinking about the programming tools is small when

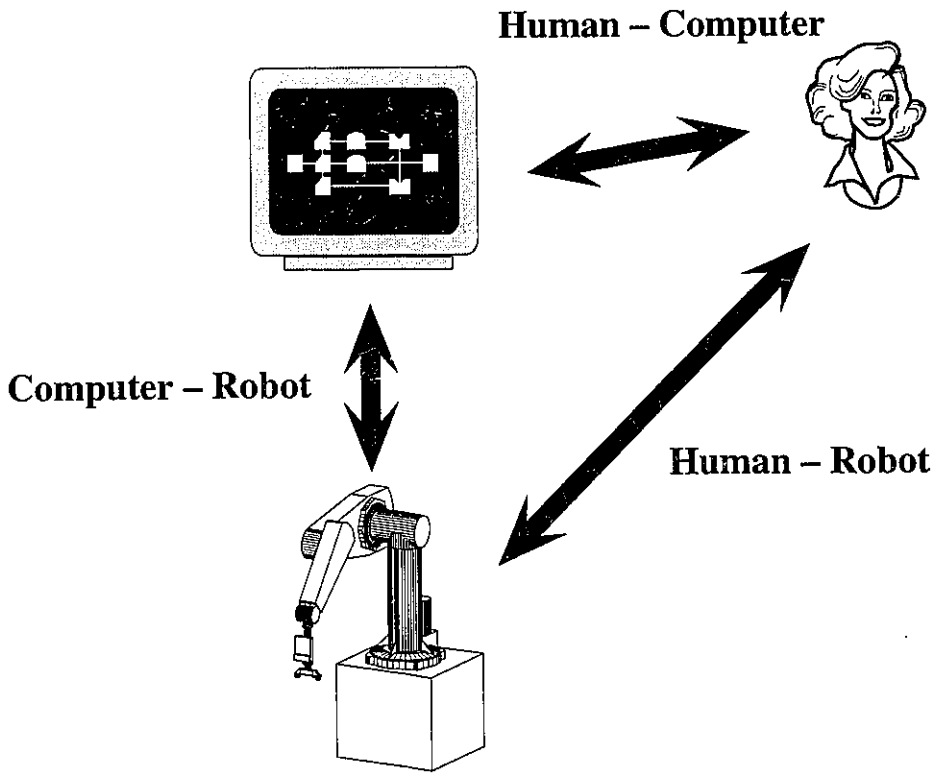

Figure 4.1: A human-robot interface as a combination of human-computer and computer-robot interfaces.

compared to the amount of time spent solving the problem. This is referred to as reducing the *tool time* [65].

It has often been assumed by user interface designers that users who cannot function effectively in an interactive environment are stupid [64]. It is worth examining more closely the suitability of the user interface design. In the human-robot interface design presented here, the complexity of the task and the atmosphere around the user are considered from the perspective of the user, as previously described. The application programmer is an expert in a particular application area. His concern is to create a program that will meet the physical requirements of the application. He may be a laboratory technician who knows the detailed requirements of a test procedure or a shop floor production engineer who knows the physical requirements of painting, glue application, etc. Currently the developer of robot applications is more likely to be a system programmer since available programming tools require knowledge of system details.

4.1.2 Mental Models

A significant design issue taken from classical human-computer interface design is the "mental model" that the user has of the system [63]. The mental model of a user interface refers to the mental representation of the system from the perspective of the human user. The information and processes embedded in the system that are not immediately visible to the user play a significant role in the user's mental model. These unknowns are hypothesized by the user and these hypotheses become part of the mental model. Novice users have much of their mental model represented as unconfirmed hypotheses. With experience using a system the hypotheses are either confirmed or

replaced with more complete information. A well-designed user interface should offer cues for the formation of correct hypotheses and make the mechanisms for confirmation of the hypotheses obvious. These cues are referred to as *affordances.* Cues that cause the user to form incorrect hypotheses are referred to as *false affordances* [65]. It is critical that each interaction mechanism in the user interface *affords* the user some capability that is either self-evident or easily remembered.

In the design of the human-robot interface, the mental model of the user is complicated by the unpredictability of the physical robot habitat. The application programmer must confirm not only the hypothetical behaviour of the user interface but also the hypothetical physical behaviour of each of the skills. The system programmer who creates and installs the skills has a different mental model of the functioning of the individual skills. Figure 4.2 depicts the mental models of the system and application programmers.

The system programmer has a model of what the robot will do based on the implementation of each of the skills. Since the system programmer is not an end-user of the interface, no information comes from the interface to update or reinforce his model. The effect of adding each skill to the system is already known in detail by the system programmer. The connection between the system programmer and the physical habitat is based on this intimate knowledge of the implementation of the skills.

However, the connection between the application programmer and the physical habitat is entirely different. The application programmer must feel that the behaviours exhibited by the robot are entirely caused by the work done in the graphical programming environment even though he has very limited knowledge of the implementation of the skills. This creates an illusory connection between the applications programmer

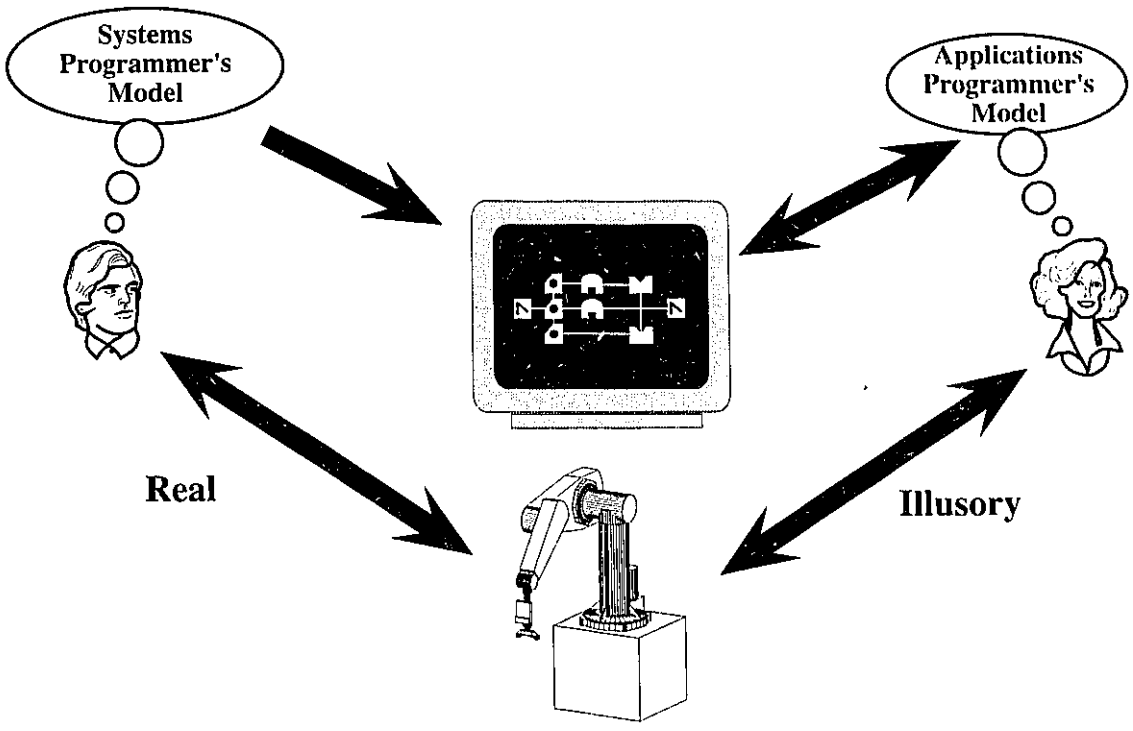

Figure 4.2: Application programmers and systems programmers maintain very different mental models of what happens within SKORP.

and the robot habitat. Each time a skill executes as it is described, the illusory "cause and effect" will positively reinforce the mental model held by the application programmer. Conversely, a skill that does not function correctly will have a negative impact on the application programmer's mental model of the system.

4.1.3 The Programming Environment

The objective of having an iconic robot programming environment is to remove complexity from the task of *explicitly* expressing robot operations. Predefined skills are made accessible to the application programmer in a graphical programming environment. This programming environment is constructed such that the application programmer specifies the actions of the robot using parameterized icons. He then uses a mouse to connect these icons to create an explicit specification of the operation. The final product is a directional graph containing icons and connecting arcs that specifies the operation. There is no textual programming, except for entering parameters in pop-up dialog boxes to alter the behaviour of the skills.

Each robot skill is represented as an icon that is placed in an operation window referred to as the *canvas*. A window that contains a selection of icons to be placed on the canvas is called a *palette*. The software tools provided to the application programmer are referred to as the programming *environment*, which is distinct from the physical workspace of the robot, referred to as the robot *habitat*.

The role of the application programmer is to decide which skills are to be used for an application, to assemble the skills on the canvas, and to parameterize the skills to meet the requirements of that particular application.

4.2 Related Research

There have been a number of attempts at providing a graphical interface for robot programming. Based on a study of human communication Leifer et al. [66] developed a method of robot task specification using text-graphic primitives. These primitives were originally developed to assist language-impaired human communication and have been adapted to represent robot actions. These actions are pictographs placed on a storyboard (canvas) to specify a sequential operation at the "intent" level. Although there are no tools to specify parameters of the behaviours or conditional execution of the actions, it is an ambitious and interesting study comparing non-lingual human communication with possible human-robot communication mechanisms.

A recent system developed by Maglica and Martensson [29] recognizes the complexity issues presented to the application programmer. This system uses functional icons arranged sequentially on a canvas. No conditional execution is provided, making the iconic representation quite simple. Nevertheless, this is an interesting system that shows promise for commercial use.

Gertz et al. [67] have proposed an iconic robot programming language called Onika. In this language, icons are colour and shape encoded. The colours serve as a visual cue for recognition of the icons by the application programmer. The shape is used for context sensitive syntax checking, i.e., similar to jigsaw puzzle pieces, it is not permitted to connect icons that do not fit together. Although this offers the advantage of preventing the programmer from connecting icons that do not make sense in sequence, it adds complexity to the task of creating a specification of the operation. First, it is not obvious when an icon is inappropriate. In some contexts it may be reasonable to connect two icons that would not make sense in another context. Second, it will

not be possible to introduce branching if the icons are spatially interconnected. Conditional execution of functions, i.e., disjunction in the flow of control, is imperative to the creation of interesting robot operations.

Another iconic language was developed by Mahling and Croft [68]. In this system icons represent actions, physical objects, and relationships among objects. This system quickly becomes complex with models of the objects and object relationships in the iconic representation.

At Asea Brown Boveri Corporation, Maier [69] designed a graphical programming system for control of mechanical systems. The distinction between data flow and control flow in graphical programming is identified. This distinction is blurred by the fact that the control flow representation of the operation requires data elements to be passed to each of the actions, and the data elements must be of the correct type. This complicates the representation of the control flow model for the user who needs to simply specify what actions are required and in what order. Maier's representation of control flow is completely different from the SKORP notion of sequential actions. In fact, it appears that Maier's model is more correctly called a high-level data flow model where the objects that the data flow to and from happen to be actions of the robot.

There are several other instances of using data flow models for control of mechanical devices and processes. A French national standard called Grafcet [59] has been used for sequential function programming. This standard is based on Petri nets [70, 71]. The essence of Petri nets is that a process can be contained in a directional graph where the nodes represent functions and the transitions represent states of the system. Branches, i.e., multiple transitions from a node, can either represent conjunction (parallel execu-

tion) or disjunction (conditional execution) of the subsequent functions. In this way both simultaneous and conditional sequences can be represented. The processes can be drawn in different ways but they often appear as ladders where the rungs represent actions and the flow of control is from top to bottom. Grafcet has been used to create a commercial programming interface for programmable logic controllers (PLCs) [59]. The capabilities of the PLCs are limited (when compared to robots) simplifying the programming requirements A Grafcet program containing both conjunction and disjunction is tractable only because of these limitations. The application programmer must deal with far less complexity in the habitat and can therefore deal with increased complexity in the programming environment.

Another high-level control language based on Grafcet was described by Schillinger and Kaufman in 1985 [72]. This appears to be the precursor to Maier's work [69] from the same laboratory. The 1985 description of Grafcet for functional programming was directed at process control, as opposed to discrete manufacturing. In process control it is essential to represent parallel activities at the highest level of design. Sequential execution alone would not be a feasible representation, and Grafcet appears to be an ideal tool for both design and programming in these areas. It was realized, however, that when one is programming robots in complex physical habitats, "Petri nets do not reduce the skills an application engineer needs [69]."

Using Grafcet was also explored in the United States as early as 1988 [73]. Although Grafcet appears to be a good design tool for some types of systems it does not meet the criteria as a basis for a robot application programmer described in this thesis, and there is no evidence that it has been accepted by robotics industries.

There are a few general purpose graphical programming languages available that

have been suggested as appropriate for programming robot applications. Prograph [74] uses the data flow model of computing. The iconic programming method used in Labview [27], patented by Kodosky [28], is also based on data flow. Both of these systems can collapse a subgraph of icons into a higher-level, hierarchical icon. This introduces the concept of increasing abstraction going up in the icon hierarchy. Because of this abstraction, it has been proposed that an application programmer can use these data flow methods for high-level programming. However, icons in data flow models require input from a previous icon and output for the subsequent icon, making them inherently context dependent. Context dependency introduces the additional requirement for the application programmer to place the icons only where specific data types can be received and passed. This additional layer of complexity, which also exists in Gertz et al.'s system [67], is not consistent with the goals of the application interface as described herein. Although the data flow model is a powerful mechanism for programming when the objective is to manipulate data, it is not appropriate or necessary when the complexity of the application is in the manipulation of the objects in the physical robot habitat. In the application programmer's interface that is described in the next section, control flow as opposed to data flow is used. The advantage is that no data is passed from one icon to the next, imposing complete functional and data independence of the icons. This will allow the application programmer to connect any two skills that are logically correct for the application. This distinguishes the SKORP model from the methods of iconic programming described above. There is no attempt to create an iconic programming language where the purpose is to transform data. The objects to be manipulated are physical, as opposed to data.

Figure 4.3: A single skill icon for the *Rub* skill.

4.3 The Application Programmer's Interface

4.3.1 The Icons

Each skill is represented by an icon that can be placed on the canvas and moved around within that window. The icons contain visual cues, including colour, shape, and text, to make the recognition of each one easier. One or two words of text in or below the icon has been found to be most helpful in recognition. Although Maglica [29] chose to use only symbols in the icons representing actions, user interface designers are now advocating the use of text in cases where there may be a large number of different symbols [65].

It is very difficult to create meaningful symbols for a large number of physical actions and it is frustrating for the application programmer to guess what a symbol depicts. "American Sign Language" (ASL) was considered as a source of symbols for the skills, but the verbs in ASL are generally expressed with hand motions that cannot reasonably be depicted in symbols. Potential users were asked for feedback on how the icons should appear, and the consensus is to have a few simple shapes with colour and text as cues for recognition. See Figure 4.3 for a sample single icon for the *Rub* skill. In the future we will introduce a combination of icon shape, colors, text, and tiny pictures inside the icon to determine if this facilitates recognition of the skills.

Each icon has one or more input connectors at the left of the icon and one exit

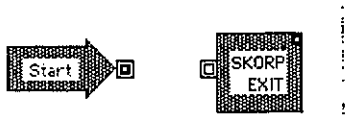

Figure 4.4: A single skill icon for the *Rub* skill.

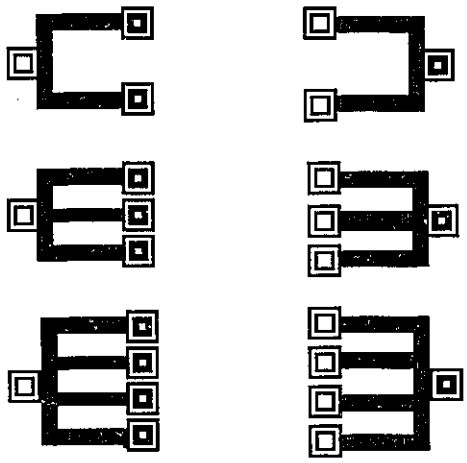

Figure 4.5: CASE and MERGE icons used for iteration and conditional branching.

connection on the right. All connectors for icons that have multiple input connectors are functionally equivalent. The skills always execute in the same manner regardless of which input connector invokes them. There are a few special icons that are exceptions to the above rules. A START icon has no input connector and an EXIT icon has no output connector (see Figure 4.4).

A CASE icon has a single input connector and multiple (two to four) output connectors. CASE icons are used for iteration and branching based on the result of the execution of the previous skill. There are also MERGE icons, which have no function except to provide symmetry to the operation and to provide extra input connectors when needed. See Figure 4.5 for a depiction of the CASE and MERGE icons.

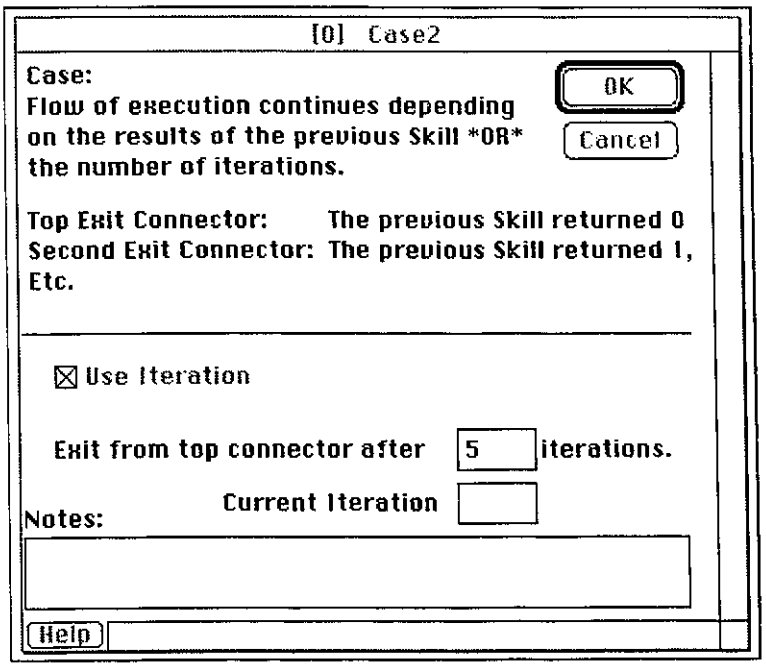

Figure 4.6: The dialog box for the CASE icon.

4.3.2 Popping Open an Icon

Each skill icon can be "popped open" by a double click, which presents a dialog box. This dialog box is always available behind the icon to remind the programmer of its functionality. The dialog box for the CASE icons allows the application programmer to specify the iteration or branching conditions. Figure 4.6 shows the dialog box for the CASE icon. Skill icons have a dialog box that allows the application programmer to modify the physical behaviour of the skill. See Figure 4.7 for the dialog box for the *Rub* skill. This customized window presents all the options for this skill. It is from this interactive window that the application programmer has indirect access to the sensors. Modification of the parameters of the skill directs the algorithms that interpret the sensor data to respond in sensor data-dependent ways.

A HELP button at the bottom of the dialog box opens another window that con-

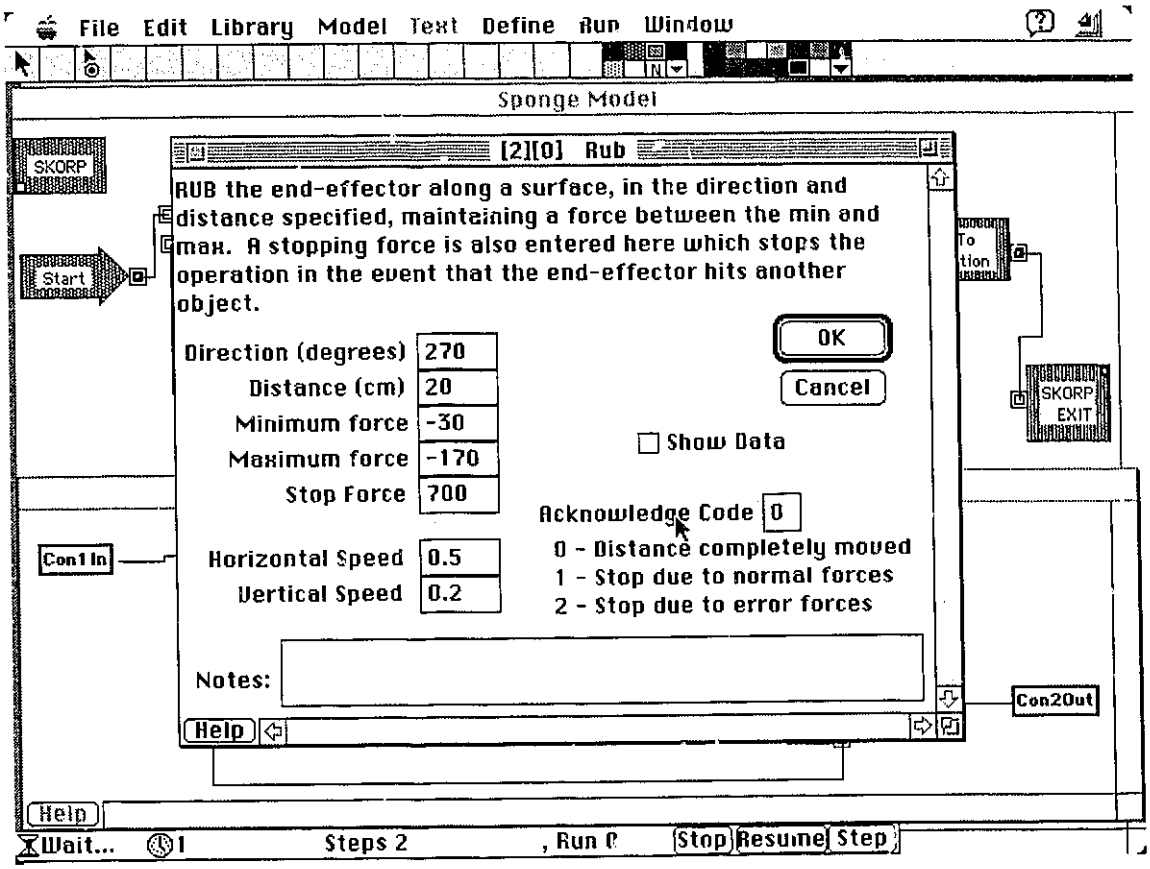

Figure 4.7: Popping open the skill icon reveals the dialog box: the *Rub* skill is shown here.

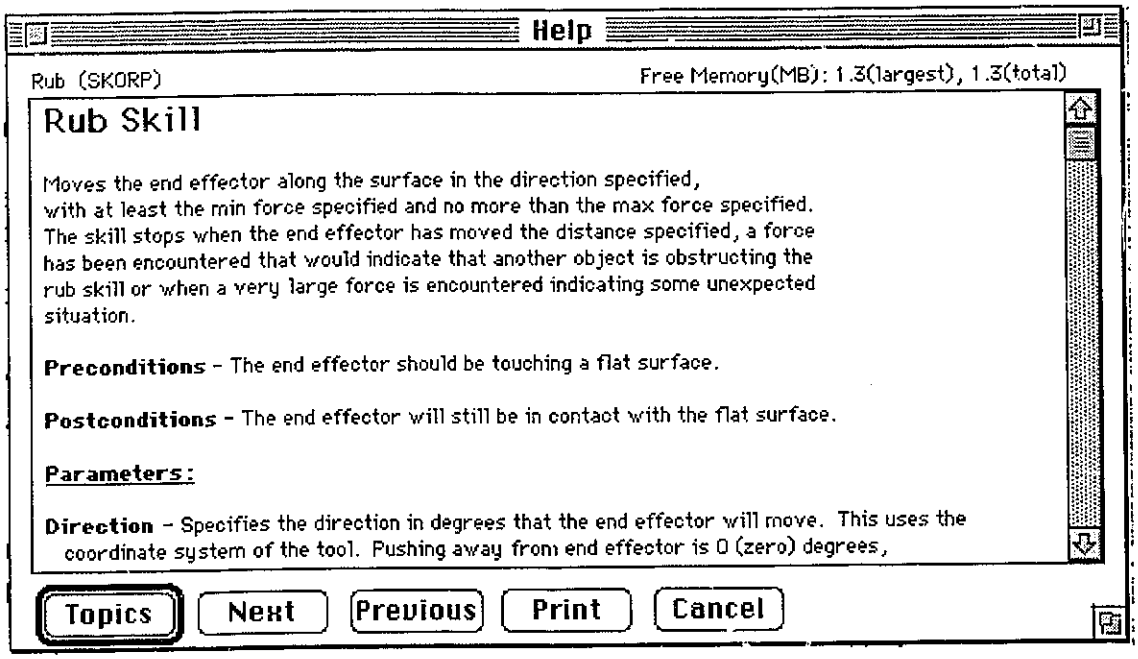

Figure 4.8: The HELP window presented when the HELP button is clicked in the dialog box for the *Rub* skill.

tains a full textual description of the skill, including preconditions and postconditions, to help the programmer decide when to invoke the particular skill. The objective is to make all the necessary information available but not to encumber the experienced application programmer by presenting more information than is immediately necessary. A portion of the HELP window for the *Rub* skill is shown in Figure 4.8.

Besides the Help button, which describes the skill textually, there can be other buttons that provide more information in textual or graphical form or allow the programmer to get information from the robot controller. These options may include fetching the current status of the robot, including its location, or displaying sensor data, as described in the next section.

4.3.3 Sensor Data Visualization Windows

One of the most difficult aspects of developing sensor-based robot skills is the lack of immediate visualization of the sensor data. The results of the interpretation of the sensor data may also be useful. This is particularly difficult if the target system executing the sensor-based skill is a multiprocessor, since the sensor data are rarely transmitted to the host machine so that it can be analyzed visually.

In SKORP, any skill that uses a sensor offers the robot programmer the option of popping open a data visualization window. Examples of where this is necessary are abundant, especially in the development and debugging stages. Suppose range data are being collected from a laser range finder during a realtime feedback skill and the interpretation of the data is incorrect because the range finder is moving while collecting data. The only easy way to determine the cause of the error is to look at the sensor data and the results of the sensor data interpretation. There are practical restrictions on this because of communication bandwidths but remote sensor-data visualization is becoming viable as communication technology develops.

4.4 Connecting Icons into Operations

The icons are provided to the application programmer on a special window called a palette. Miniature versions of the icons and their textual names offer cues to the selection of the skills. Figure 4.9 shows a sample of a skill palette. This window can be scrolled to locate the appropriate skill. The icons are dragged from the palette and dropped onto the canvas.

The icons are functionally mutually exclusive, i.e., only one skill icon executes at

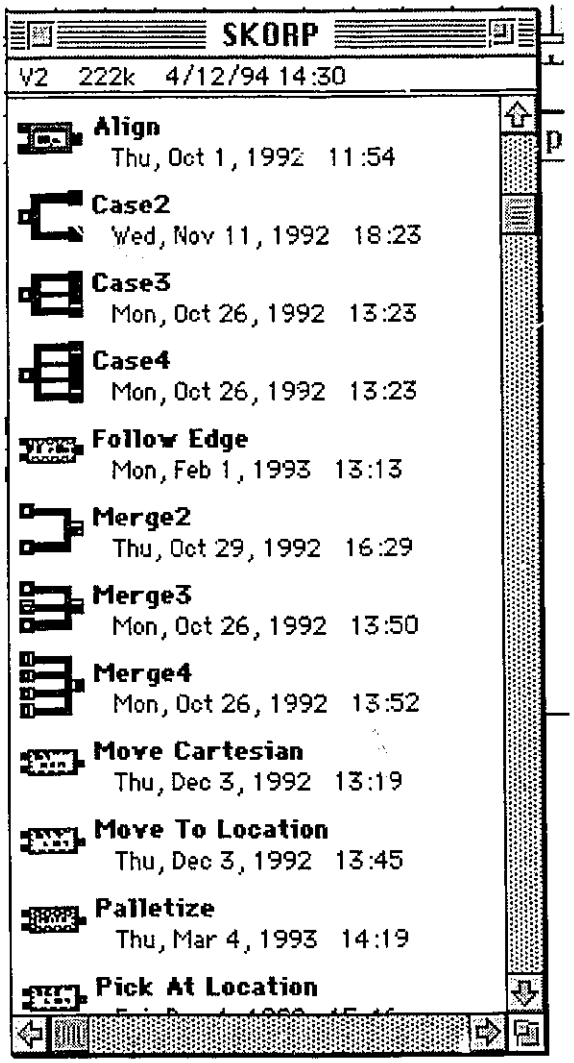

Figure 4.9: An example of a palette containing the miniature skill icons available to the application programmer.

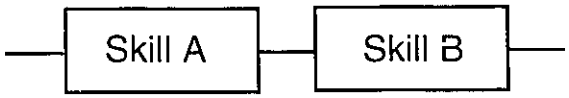

Figure 4.10: Simple sequential execution of skills.

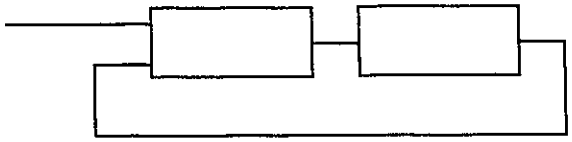

Figure 4.11: An infinite iteration.

any time. The progression of the execution of the operation can be thought of as passing an execution token from icon to icon along the connector lines. Moving a token from place to place functions exactly the same as playing a board game. Almost all potential users are familiar with these games. Taking advantage of the user's non-computer experiences is one way to provide cues to the functionality of the user interface [75]. Execution order of the icons is therefore intuitive. Input connectors are on the left and output connectors are on the right. Branching and merging are also obvious.

4.4.1 Control Constructs

There are only three control constructs: sequence, iterate, and case. The sequence construct is as described above, generally from left to right in the window and always from an input connector on the left of the icon to the output connector on the right of the icon. See Figure 4.10.

Iteration can be represented in two ways. Infinite iteration simply wraps the output connector around to the input connector of a previously executed skill. Figure 4.11 shows this infinite loop. Although it almost never makes sense to create an infi-

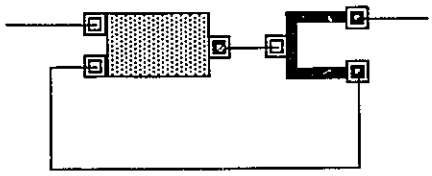

Figure 4.12: Iteration using the CASE icon.

nite loop when one is programming computers that do not interact with the physical habitat, many industrial robot tasks have no exit, i.e., the objective is often to define an infinitely repeating operation that is only interrupted by an operator.

Loops that eventually exit are controlled by the CASE icons, as described previously. On the operation canvas they appear as in Figure 4.12. Nested iteration and inter-nested iteration are equally valid.

Perhaps even more important than the control constructs that exist in the SKORP model are the constructs that do not exist. There is no mechanism for conjunction. Since there is a single execution token, a MERGE can never indicate the requirement that more than one previous skill must have been completed.

There is no possibility of the application programmer representing parallel execution in SKORP. Although it is often necessary to have parallel execution on the realtime subsystem, this must always be contained within one skill.

The operations represented as a connected graph do not represent a language in the formal sense [76]. There are no computational context dependencies. The graphs cannot be parsed since there is no grammar. This is an intentional design consideration. The order of the skills is the responsibility of the application programmer. The preconditions for a skill should be present when the skill is invoked. There is some inclination to attempt to parse all robot programs for safety reasons. However, pars-

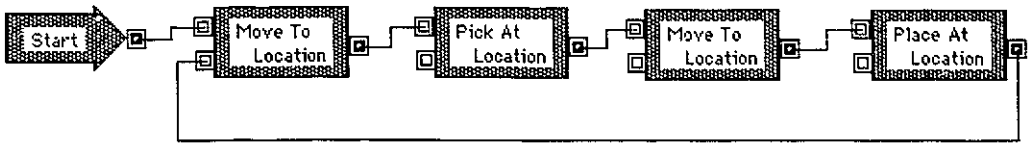

Figure 4.13: An iconic description of a simple pick and place operation.

ing will never *guarantee* that the habitat will be safe and may serve to introduce a false sense of confidence in an unsafe program. It is not feasible to *guarantee* anything about a robot program before it is actually executed. The safest method of developing operations in a partially known habitat is to test the actual programs while the robot moves very slowly. This can be done skill by skill. As the application programmer gains confidence in each skill the speed can be increased for that skill.

4.4.2 Example Operations

A few examples of the appearance of the application programmer's interface will make this description clear. Suppose the application programmer wants to describe a pick and place operation using skills. Figure 4.13 shows the simple pick and place operation. The START icon does some initialization of sensors and controllers and provides a starting place for the operation. Three skill icons are shown: *Move To Location*, *Pick*, and *Place*. The control is from left to right except where the *Place* skill exit wraps around to the initial move, creating a repeat loop. There is no internally generated exit, except in the case that one of the skills fails to execute.

A slightly more complicated scenario is depicted in Figure 4.14. Suppose the environment consists of some transport mechanism that will deliver any one of three parts. The task is to identify which of the objects is arriving and perform a *Pick* skill and

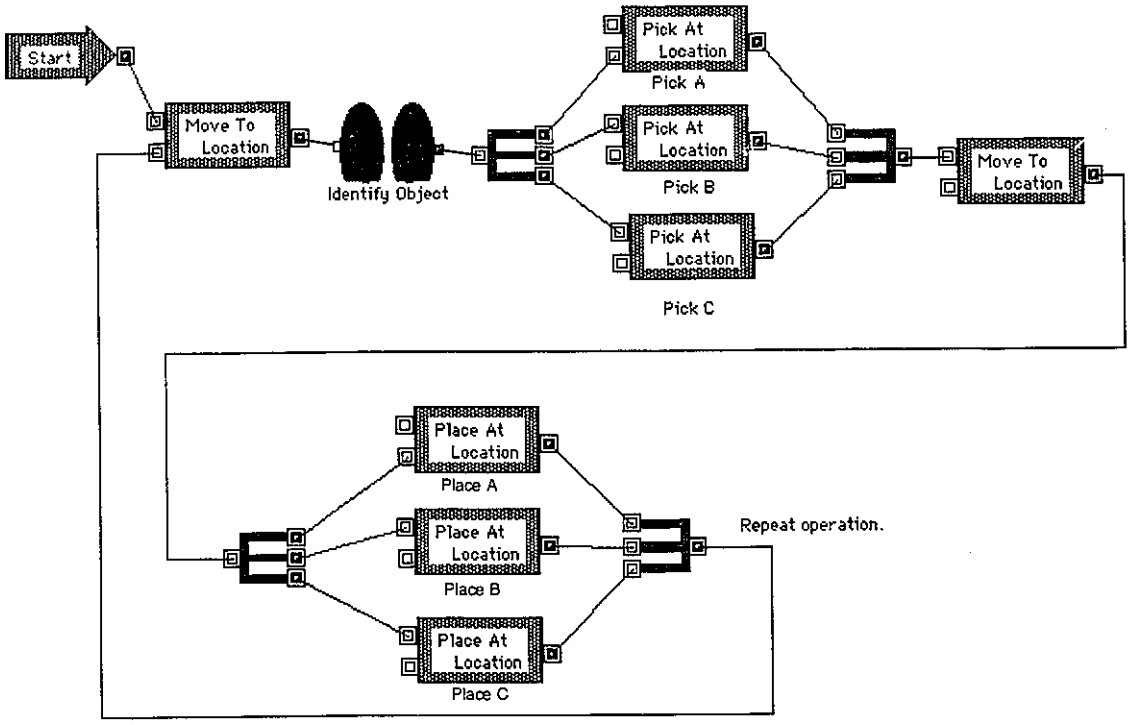

Figure 4.14: A pick and place operation that uses a sensor to identify the object and does a selective pick and a selective place.

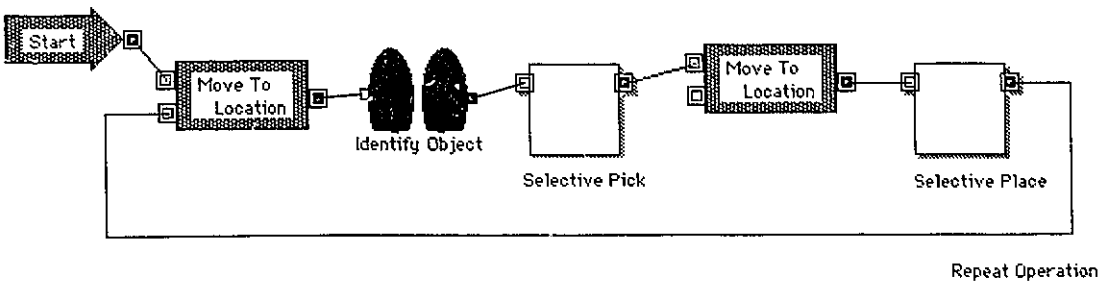

Figure 4.15: Pick and place using a sensor as in the previous figure, with the selective pick and selective place portions collapsed into a complex icon.

a *Place* skill specific to that object. The CASE and MERGE icons are used for the branching. As can be seen in the figure, the number of icons can begin to become cluttered on the canvas. The execution path is moved from right to left after the second *Move to Location* skill.

Although it is easier to produce a figure on paper by doing this wrap-around, the application programmer may find it easier to scroll the window so that only a portion of the operation is visible at any time.

4.4.3 Hierarchical Icons

Another option to producing an entire operation in a single window is to further abstract the operation by combining a group of related skills into a single icon. For example, in Figure 4.14, the *Pick* skills and the CASE and MERGE icons could be easily abstracted into a single icon. Similarly the *Place* skills can be combined. This result of this abstraction is shown in Figure 4.15. The icons that represent an abstraction of other icons are indicated by shadows, and are very obvious to the application programmer who can pop these open to reveal the underlying icons at any time.

4.4.4 Execution of the Canvas

The canvas serves as a window where the operation is constructed by connecting skill icons. It is also used to execute the operations. By selecting *Run* from a menu or with a single key-stroke the operation begins execution. The START icon establishes communication with the realtime subsystem. The workstation then proceeds to send commands to the realtime subsystem (RTS) that invoke the skills as the execution token is passed from icon to icon. The RTS responds after the execution of each skill with a message called an *acknowledge code*, indicating which of the possible states the habitat is in after the skill completed. Usually there are only two possible acknowledge codes: normal completion or fatal error. In some cases there are intermediate states that indicate that the skill completed in a way that will be of interest to subsequent skills. This acknowledge code is used by the application programmer to branch within the operation, thereby recovering from expected error conditions or simply executing different skills according to what was learned by the previous skill. An example of this was seen in the pick and place application in Figure 4.14, where the sensor was used to determine which of the parts was available for acquisition.

When the operation is executing, an execution control bar appears at the bottom of the canvas, as shown in Figure 4.16. This bar permits the application programmer to stop the operation, pause at the end of the currently executing skill, resume execution, and step through the operation skill by skill.[1] This is a very valuable debugging tool. The dialog boxes can be opened and the parameters adjusted while an operation is paused. This allows for very rapid *trial and error* to select parameters for the skills.

[1]Stopping the operation from this interface is not an emergency stop. Emergency stop buttons are always provided by the robot manufacturer and will be connected directly to the robot controller.

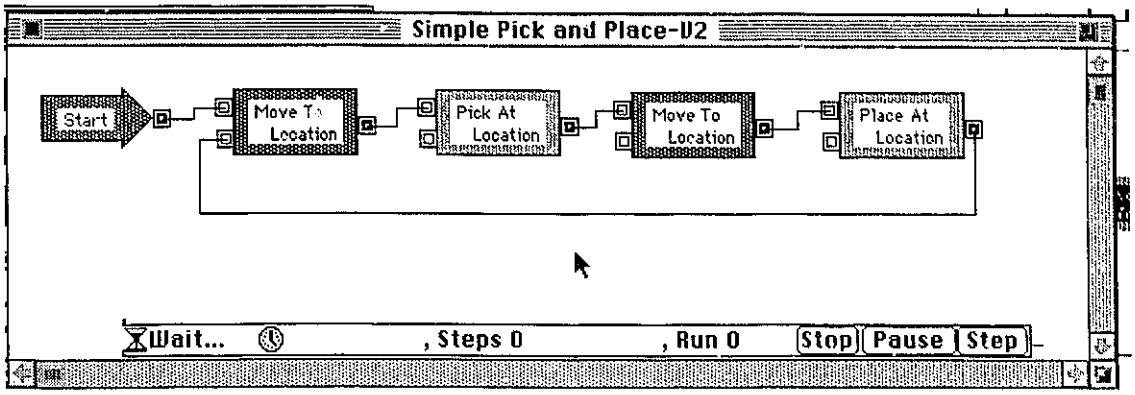

Figure 4.16: During the execution of the operation an execution control bar appears.

The instructions to the application programmer will often indicate a safe starting value for the skill parameters and recommend that they be varied incrementally until the skill is tuned to the needs of the application.

Chapter 5

Experimental Implementation

5.1 Introduction

The experimental setup consists of a PUMA 560 robot, a Macintosh personal computer, a multiprocessor realtime subsystem, a force-torque sensor, and a laser range finder. Figure 5.1 is a photograph of the laboratory at the NRC showing each of these components. A schematic of the apparatus is shown in Figure 5.2. This schematic represents the realization of the high-level diagram shown in Figure 2.1.

This chapter, which describes the current implementation of SKORP, is divided into three sections with references to the appendices. The creation of the iconic programming interface is described in Section 5.2.

The realtime subsystem was implemented using the Harmony operating system running on multiple processors and communicating with the Macintosh. This implementation is described in Section 5.3.

A number of interesting sensor-based skills have been created. Some of these became interesting research projects in themselves. With a laser range finder for non-

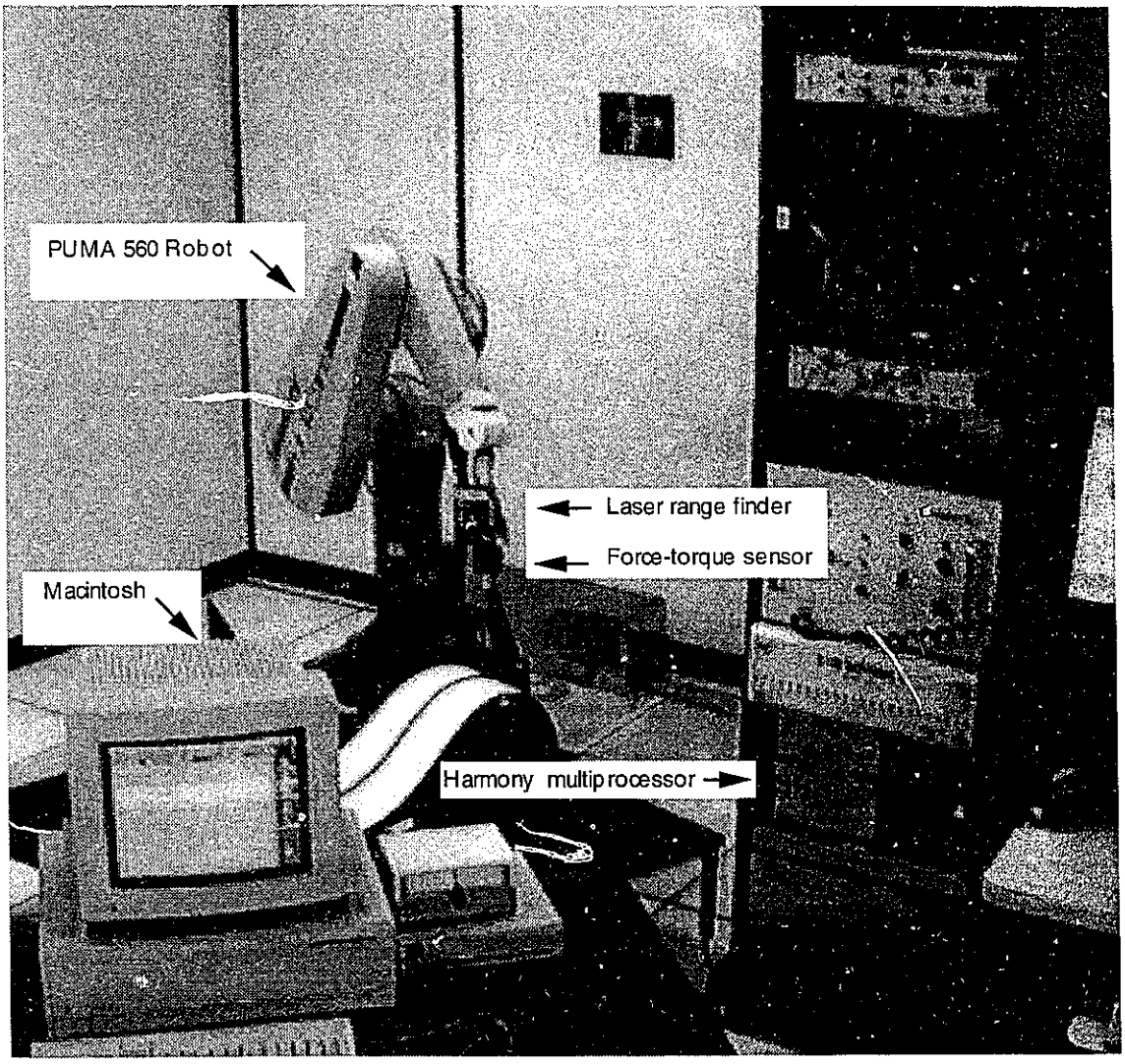

Figure 5.1: The PUMA robot, the MacIntosh computer, and the controlling hardware for the SKORP implementation.

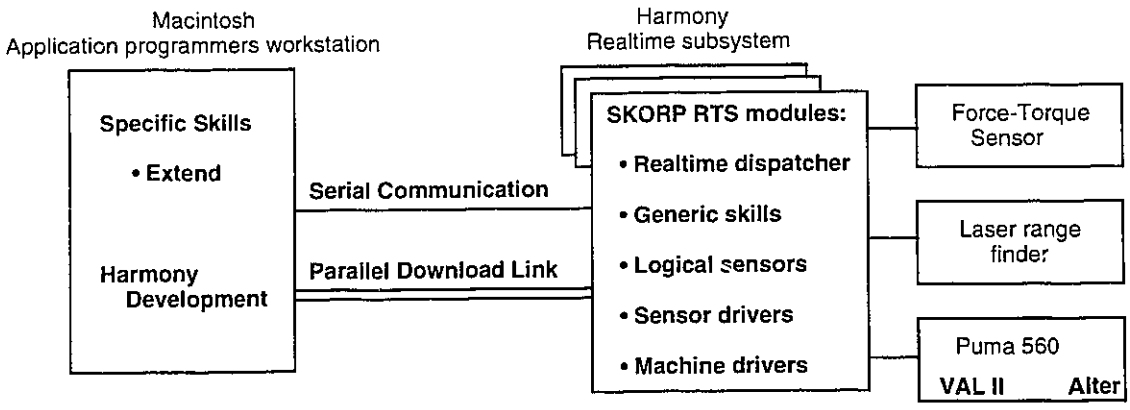

Figure 5.2: The PUMA, Harmony, and Macintosh apparatus schematic.

contact sensing and a force-torque sensor for tactile skills, the combination of the realtime subsystem and application programmer's interface has been demonstrated in difficult application areas. Some of the skills and operations that have been implemented are described in detail in Section 5.4.

5.2 Iconic Programming Implementation

The application programmer's interface was developed on the Macintosh with tools provided in a commercial software package called Extend [77], a package intended to be used for discrete and continuous simulations. Some of the tools provided in Extend are for

- creation of icons and dialog boxes
- communication mechanisms between icons
- communication between the icons and a serial port

These are the tools that were used to create the application programmer's interface. The extremely complex package allows many different types of simulations to be created. Since Extend is not used for its intended purpose, most of the capabilities are ignored.

5.2.1 Creating Icons and Dialog Boxes

An interface is provided in Extend to create icons. The behaviour of the icons is written in a formal language called ModL. This language is similar to C language with simplifications for ease of use and several additions for customizing message-based communication between icons. The *Rub* skill will be used to illustrate how icons and dialog boxes are created.

Figure 5.3 shows the window where the dialog box was created and Figure 5.4 shows the variety of interactive devices that can be placed in the dialog box. Each of the items added to the dialog box is given a variable name that is used later in the creation of the ModL code that defines the behaviour of the icon.

The ModL code for a particular icon, the shape of the icon itself, and the information presented to the user when the HELP button is pushed, are all implemented in the same multipane window. Figure 5.5 shows an example of this multipane window during the development of the *Rub* icon. The panes each have different functions. The upper left pane is used to create the icon, including its input and output connectors and to identify the areas of the icon that are used to animate the execution of the icon. Drawing tools and a colour palette are provided to create the icons. The upper right pane is used to enter the information that will be presented when the user requests help for this skill. The lower left pane displays the variable names that are associated

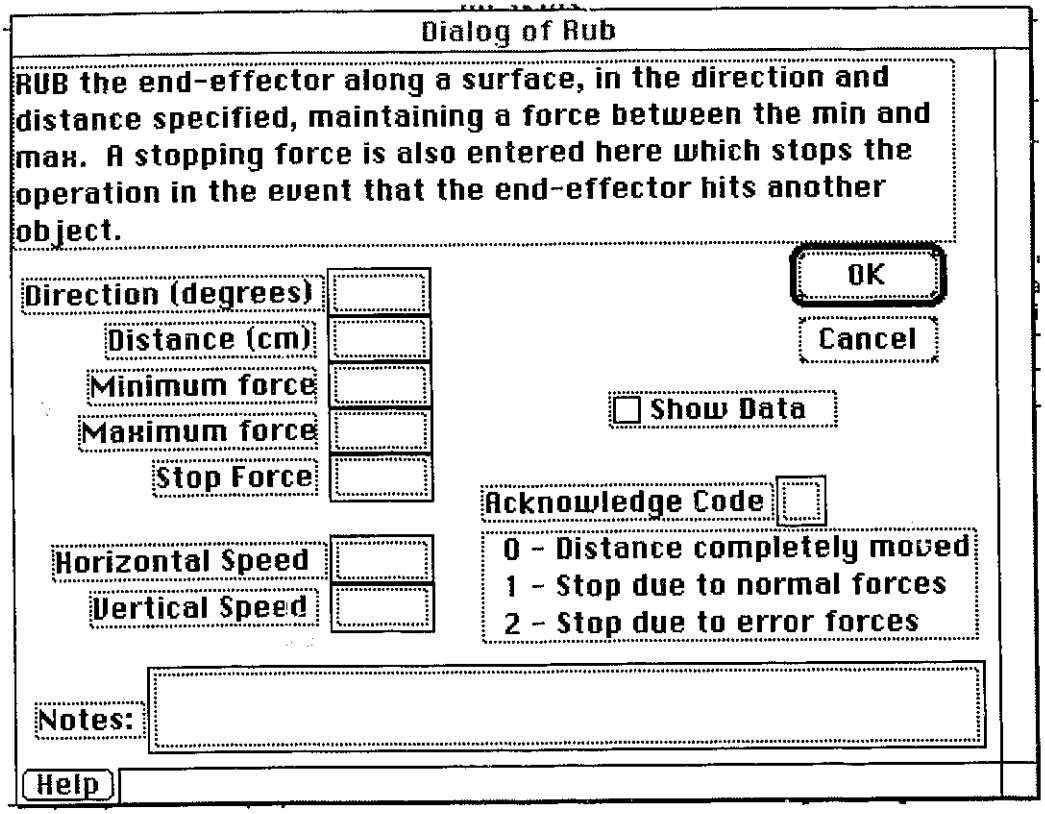

Figure 5.3: The dialog box for the *Rub* skill was created in this interactive window.

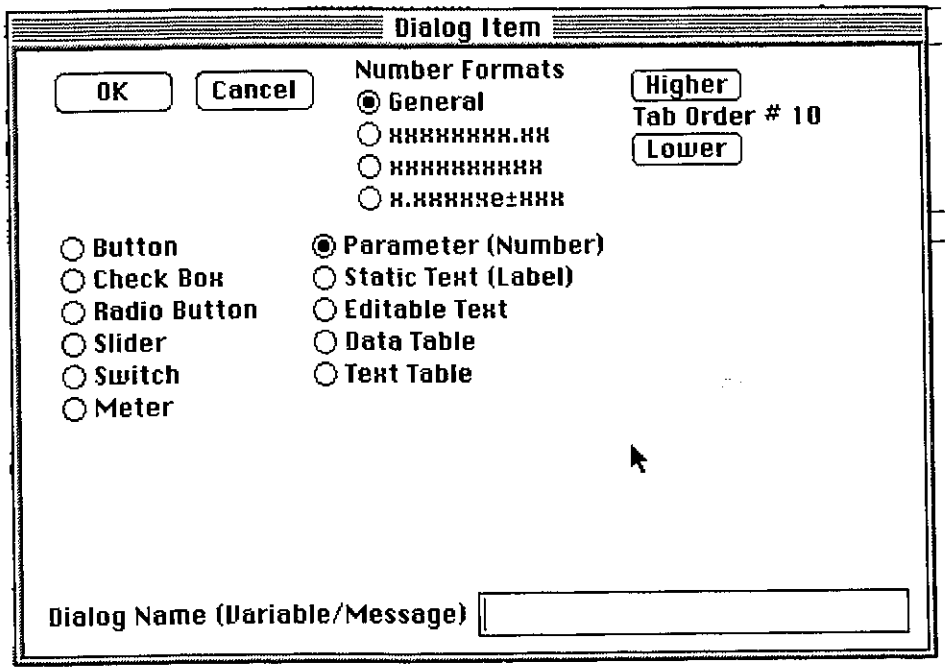

Figure 5.4: Each of the items placed in the dialog box is given a variable name.

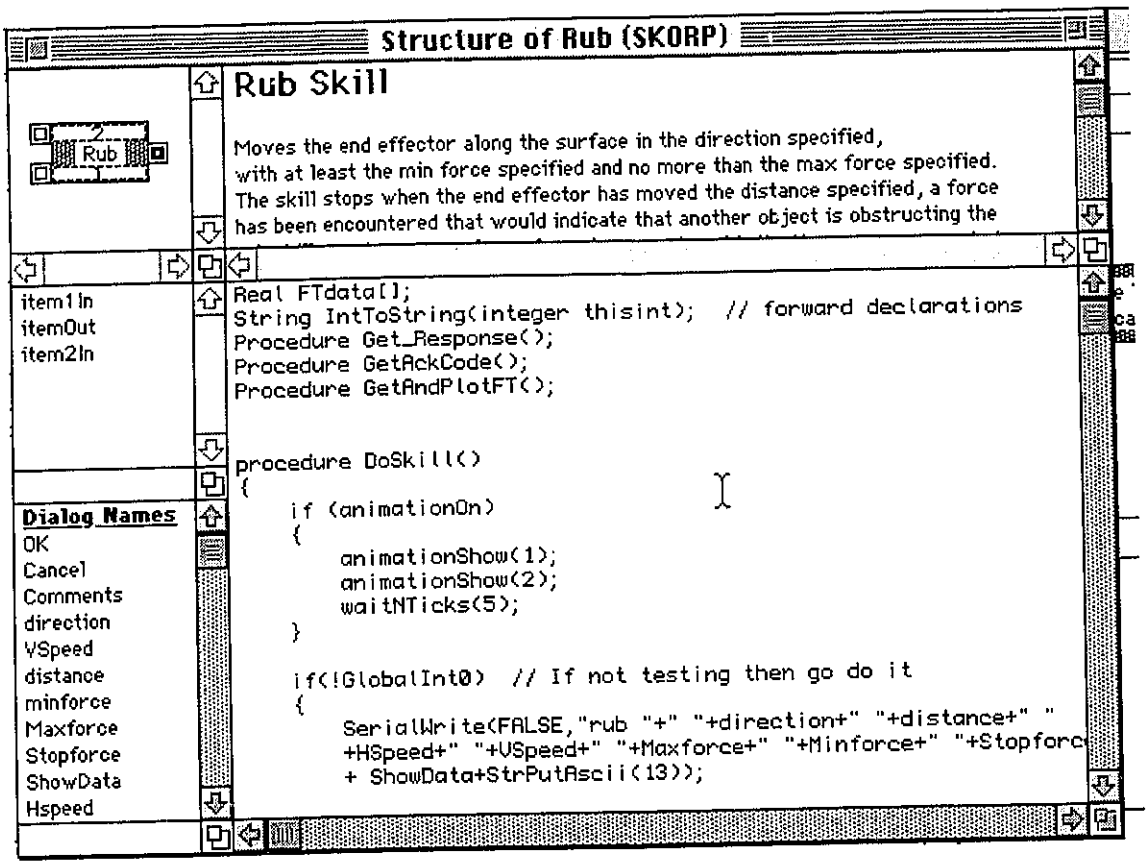

Figure 5.5: The multipane window used to creating the icons, HELP text, and the ModL code.

with the dialog box elements. The middle left pane shows the variable names for the input and output connectors. The lower right pane is a ModL programmer's editor. Using the ModL language the behaviours of the icon are described in this pane.

The icons in Extend communicate via message passing. In the SKORP application programmer interface, the skill icons use this message passing mechanism only to pass the execution token from skill to skill. When a skill icon receives any message at any input connector it has received the SKORP execution token. It will then proceed to

execute the skill by sending a string to the RTS and waiting for an acknowledgement that the skill was completed. The acknowledgement is in the form of a numerical code that indicates the conditions under which the skill completed execution. An acknowledge code of 0 (zero) indicates that the skill completed normally. When this acknowledgement is received, the acknowledge code is placed in a global variable and a message is sent via the output connector, indicating that this skill has completed. Note that each of the icons is completely independent of the others except for the message passed to the next icon. In fact, this is not really a message, since it contains no information except to signal the completion of a skill. A global variable that contains the acknowledge code is used by the CASE icons for conditional branching. From the time an icon receives the execution token until the acknowledge code is received from the RTS, the icon will change colour to indicate to the application programmer which skill is currently executing.

The development of the skill icon and the specification of the behaviour of the icon are responsibilities of the system programmer. When the skill is implemented in the realtime subsystem and the communication between the workstation and the realtime subsystem is defined, the system programmer then adds the icon to a library. The skill libraries are used by the application programmer to select the necessary skills for an operation.

The ModL code has a similar structure for all the skill icons. The ModL code for the *Rub* icon can be found in Appendix A.1. Although this language is not widely used, it is quite readable to anyone familiar with the C language. The ModL code for the CASE icons is quite different, so the code for the creation of the CASE icon is included in Appendix A.2. The main behavioral differences in the CASE icon are that

Figure 5.6: The cursor will automatically change into a connect tool as it passes over an input or output connector.

it has no communication with the RTS and it examines a global variable to determine which of its output connectors will be used to pass the execution token.

When creating an application, the application programmer places the skill icons on the canvas and connects them. The cursor will automatically change into a connecting tool (shown in Figure 5.6) when placed on an input or output connector. There are very few interactive mechanisms for the application programmer to learn and remember. This is critical in practice, since there may be long periods of time when the application programmer does not work in this environment at all. Re-learning complex interaction mechanisms is frustrating and should be unnecessary.

It was mentioned in Chapter 2 that the icons can be collapsed into a hierarchical icon. This allows the application programmer to develop reusable sub-operations. It also provides a way to fit an entire operation on a single screen. For example, Figure 5.7 shows an operation in the application programmer's environment. The icon called *Rub in Square* has a shadow, which indicates that it contains a sub-operation. Clicking on one of these hierarchical icons reveals the underlying layer of icons.

It is interesting to note that application programmers who have looked at the concept of collapsing icons prefer to think of these hierarchical icons as super-skills as opposed to sub-operations. This is reasonable if one considers the mental model of the application programmer. These non-computer scientists prefer to maintain a mental

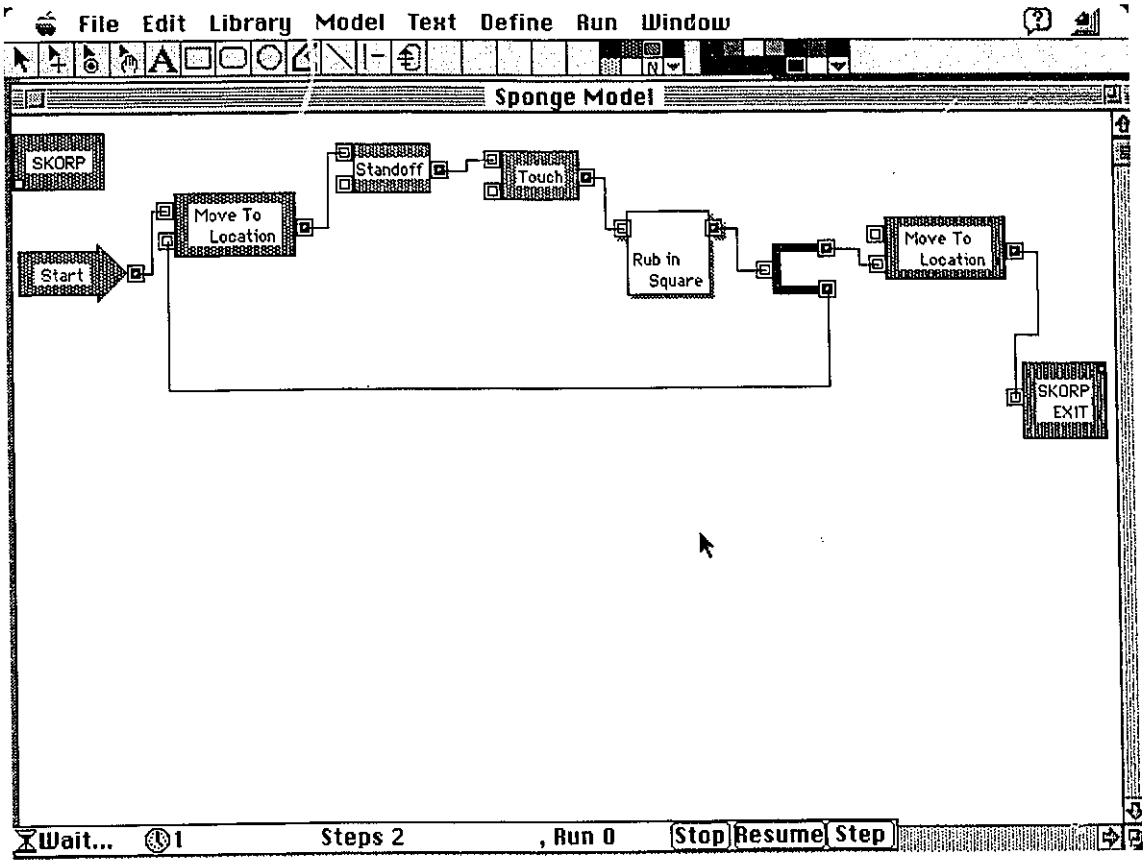

Figure 5.7: View of the iconic programming environment on the MacIntosh.

model of the entire operation represented in a single plane, and a super-skill is more easily considered to be a single entity than a sub-operation.

Figure 5.8 shows the skill icons that represent the *Rub in Square* sub-operation. These *Rub* icons can then be opened to reveal their dialog boxes. As these windows are opened they stack on top of one another, with the topmost window active at any particular time. Figures 5.9 and 5.10 demonstrate this by showing the dialog box and the HELP window for the *Rub* skill.

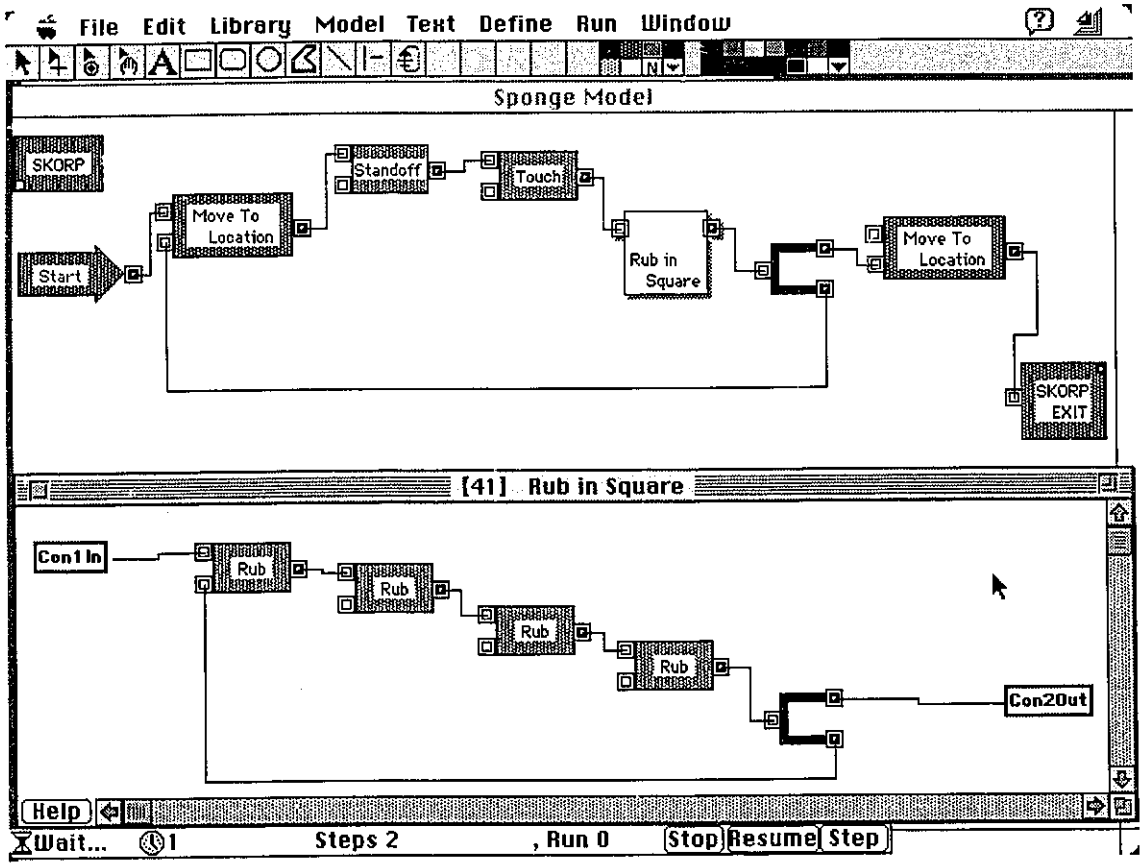

Figure 5.8: The collapsed icon in the previous figure is opened to reveal the lower layer of icons in this operation.

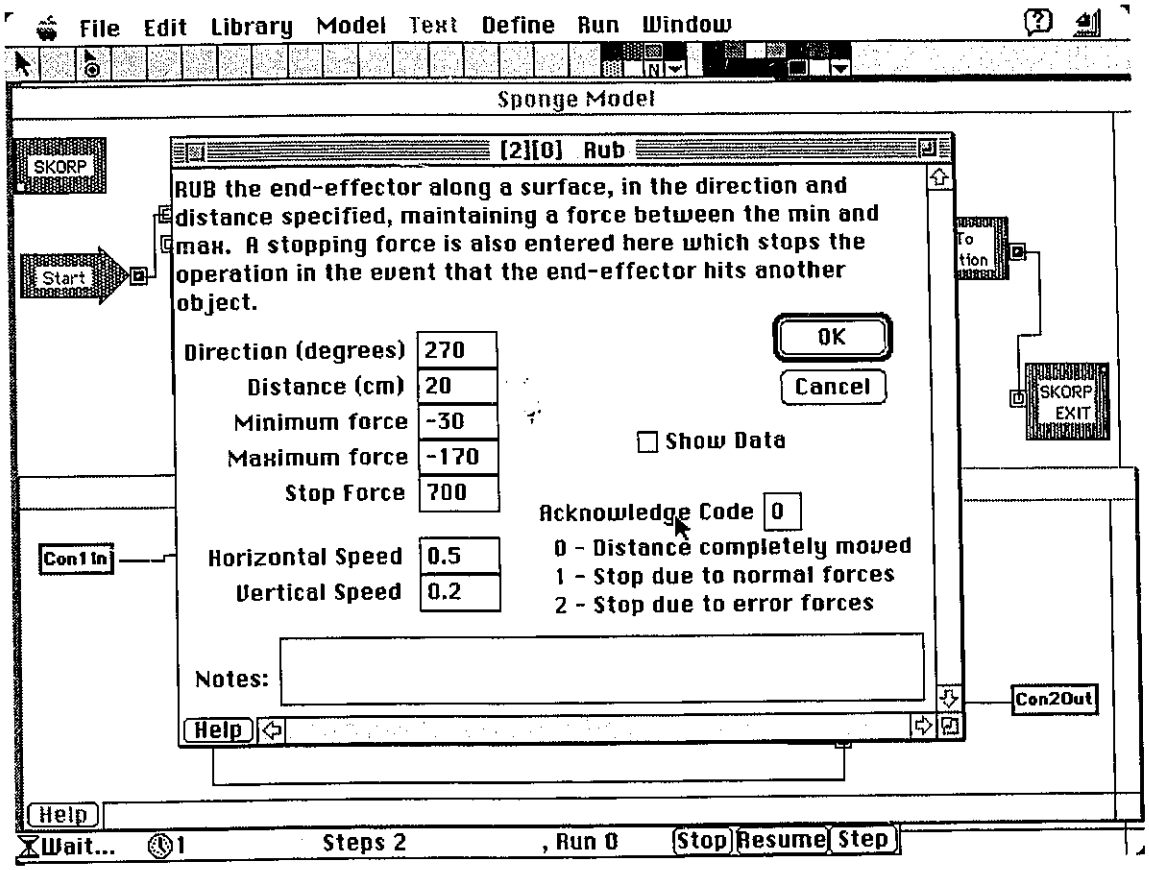

Figure 5.9: Popping open one of the skill icons reveals the dialog box on top of the other windows that are now inactive.

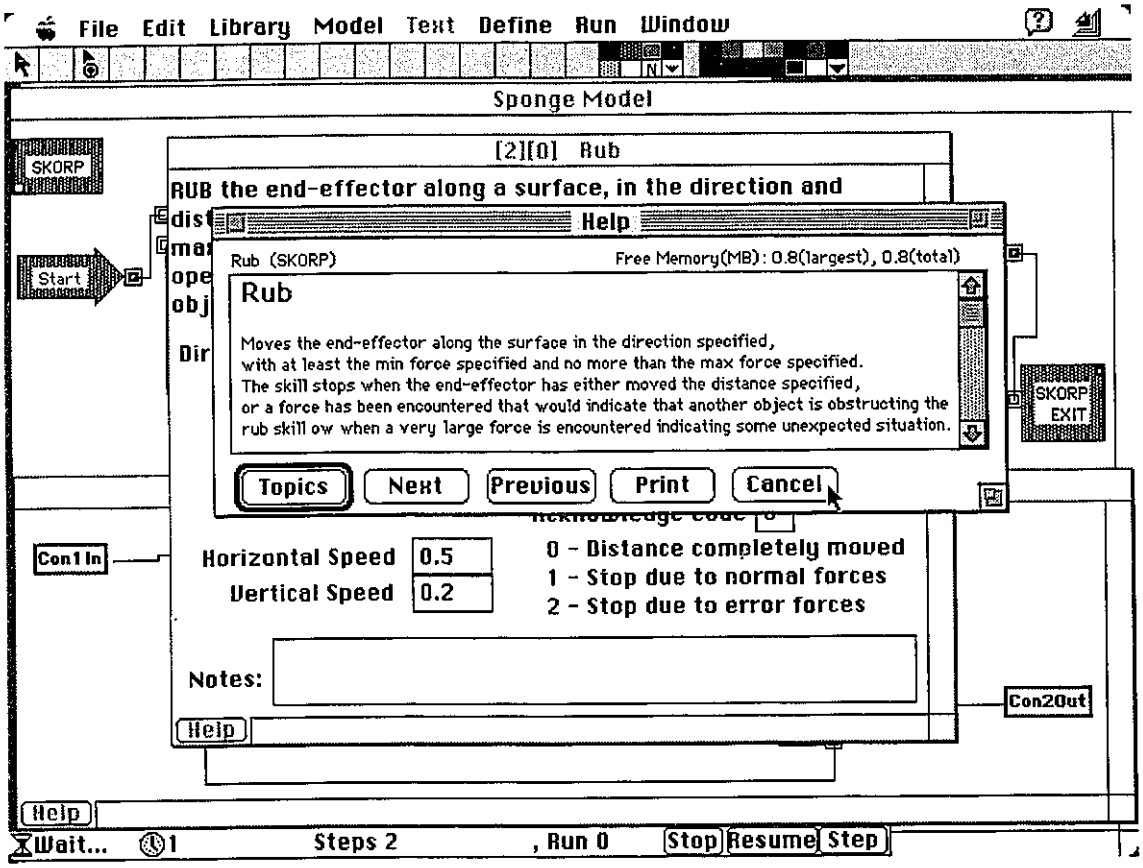

Figure 5.10: The HELP button reveals all information on the skill and its parameters.

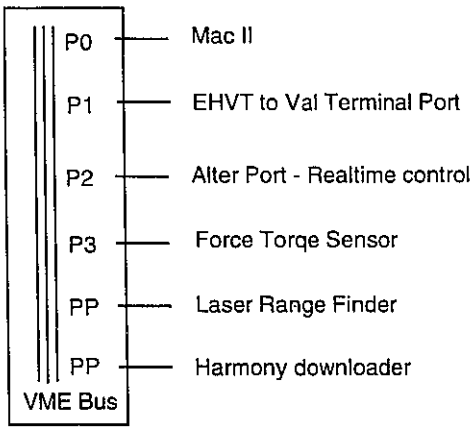

Figure 5.11: The I/O configuration of the four processor Harmony realtime subsystem.

5.3 Realtime Subsystem Implementation

The realtime subsystem described in Chapter 3 is implemented using the Harmony operating system. The particular Harmony system used here has four single-board MC68020 processors that share a common address space on a VME backplane. Each of the processor boards (P0 - P3) has an RS-232 serial port and there are two additional parallel port boards (PP) on the same backplane. Figure 5.11 indicates the purpose of each of the serial and parallel ports.

5.3.1 The Harmony Operating System

Harmony is a multitasking, multiprocessing, realtime operating system developed at the NRC. Gentleman et al. [78, 79] developed Harmony with the original intention of providing an open architecture for realtime robotics applications. Several case studies of laboratory research projects using Harmony for robot control can be found in [17]. Harmony-based applications are designed as a set of communicating tasks. The multiprocessing capability can be exploited to achieve optimum system utilization and

throughput by suitably distributing the application tasks on the various processors at link time in order to obtain a correct load balance. Determining a correct load balance is difficult and this difficulty led to the creation of the tools described in Chapter 3, namely, PAL, PAN, and ATC.

Harmony is written in C language with a few critical sections in assembly language. Harmony does not support an on-board application development environment. Executable programs for each processor are compiled and linked on the Macintosh using the MPW C language [80].

Since there are four processors in this instance of the architecture, up to four tasks can be actually executing in parallel but many tasks can share each processor. Harmony supports preemptive, priority-based FIFO scheduling of tasks. This allows a task at the top of the priority queue to be scheduled for execution if it is ready. The currently executing task is preempted if it is either blocked, as discussed below, or a task with a higher priority becomes ready.

Communication among the tasks is achieved by message passing. There are four message-passing primitives implemented as functions:

- Send
- Receive
- Try receive
- Reply

Synchronization of the tasks is achieved by the blocking behaviour of these primitives. The scheme used in Harmony is referred to as the *rendezvous* paradigm [81]. The Send and Receive primitives are blocking mechanisms, i.e., the task that executes

a Send blocks while waiting for a reply, and the task that executes a Receive blocks while waiting for a message to be sent. The Try Receive and Reply mechanisms are non-blocking. Both Send and Reply contain messages, i.e., user-defined data structures.

Tasks are created and destroyed using the Create and Destroy primitives. On system start-up the task called Main on the first processor (P0) begins execution and this task creates sub-tasks on any available processor which in turn may create sub-sub-tasks, etc., resulting in a tree of tasks. Tasks normally communicate with their parent and child tasks only. A special task called a Server is commonly used to provide a service to all the other tasks. This task is accessible to all tasks on all processors.

The blocking behaviour of the rendezvous paradigm for inter-process communication can result in an undesirable effect called *subroutining*. In a multiprocessor client-server model the obvious objective is to have the processors executing tasks in parallel. Using the Send-Receive-Reply blocking primitives the client (sending task) blocks waiting for the reply from the server (receiving task). This is the subroutining effect, so called because it is only as effective as executing the client and server tasks on a single processor using subroutines. This can easily be avoided using a special task called a *courier*. Since both Send and Reply contain messages, the client avoids using the blocking Send primitive by creating a task that does nothing but send messages to the client and server in an endless loop. In this way the client and the server can execute in parallel. Figure 5.12 shows a courier task being used between a client and server. The arrows indicate the directions of the Send primitive. (An example of using a courier will be given in the next section.)

This brief introduction to Harmony primitives and a knowledge of C language will allow the reader to understand the examples of the code that implements the SKORP

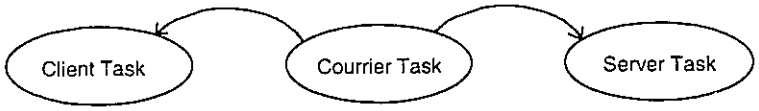

Figure 5.12: A courier task between client and server tasks.

realtime subsystem.

5.3.2 Software Organization of the RTS

It is very important to place the Harmony tasks on processors that will not conflict with each other. Although in general any task can be placed on any processor, the physical connection to the serial ports will dictate on which processor the external communication functions reside. The other obvious consideration is that the most time-critical tasks should be spread across the processors. It has been found that a reasonable estimation of this can be done without using timing charts, but the distribution of tasks always needs to be verified before implementation. The most time critical functions are almost always tasks that communicate with the actuators and sensors. It is usually the external devices that present the most difficult hard realtime deadlines.

In Harmony, the distribution of tasks across processors is decided by the system programmer. Tasks never migrate; they exist on one processor only. The task template files indicate where each task resides. The task template files for the current implementation are included in Appendix B. The distribution of the most interesting tasks can be seen in Figure 5.13.

In SKORP, skills execute exclusive of each other and because of this all the skill modules can be implemented on the same processor; they will never be in conflict with each other. All the skill modules have been implemented on the first processor (P0). Skills are implemented as subroutines within the main task. The logical sensors are

Processor 0	Main (including skill modules) Terminal server for interface to Macintosh Debug tasks Logical sensors for range data
Processor 1	Serial interface to PUMA terminal port Sensor driver for range finder Machine driver for PUMA through terminal port
Processor 2	Serial interface to ALTER port Machine Driver for PUMA through ALTER port
Processor 3	Serial interface to force-torque sensor Sensor driver for force-torque sensor

Figure 5.13: The distribution of tasks among the Harmony processors.

implemented as tasks, and the sensor drivers and machine drivers are implemented as servers. The code for one or two interesting examples of each of the types of modules is included in the appendices, as discussed in the following sections.

5.3.3 Logical Sensor Implementation – An Example

In Appendix C.1 the code implementing the logical sensor *edge-slope* is presented. The objective of this module is to

- fetch a range profile using the range finder sensor driver
- calculate the location of the step edge in the range profile
- calculate the angle of the surface near the edge
- reply to the client with the edge point and slope information

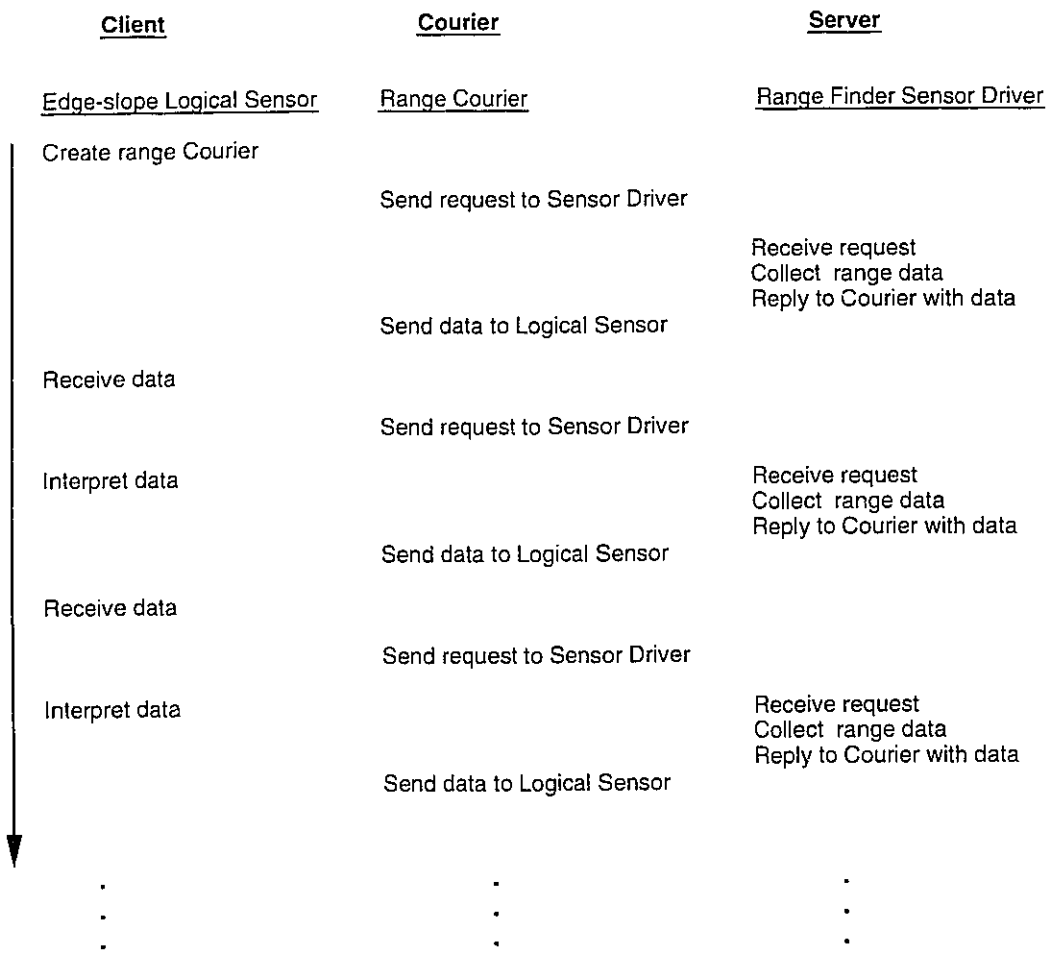

Figure 5.14: The logical sensor collects data with the help of a courier task.

An example of a skill that uses this logical sensor is the *Follow Edge* skill described in detail in the next section.

The interesting aspect of this logical sensor is its use of a courier to collect the range data. The communication among the tasks involved in range data collection is shown in Figure 5.14. When the logical sensor is created, it immediately creates an instance of the range courier. The range courier sends a request to the range finder sensor driver which in turn fetches a range profile and replies with the profile to the logical sensor.

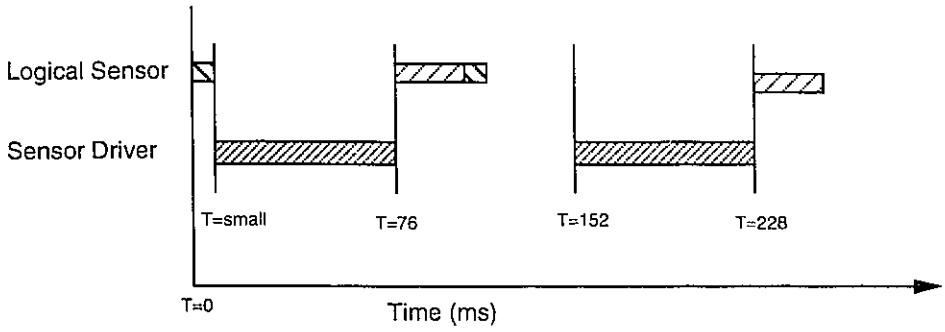

Figure 5.15: The Activity Timing Chart for a logical sensor without a courier.

As soon as the logical sensor has the range data it begins the sensor data interpretation. The courier requests more data from the sensor driver so that the data can be collected in parallel with the interpretation. The courier allows the logical sensor to interpret every range profile. This can be seen more precisely using an Activity Timing Chart. The range data collection requires 76 ms/range profile. There is a short delay between the end of one profile and the beginning of the next. The interpretation of the range data may take 10-15 ms. The design of the data collection should be appropriate for any logical sensor that may be developed in the future, so the exact length of time required for the interpretation of the range data is not important, except to say that it cannot be completed between the collection of profiles. Also, the length of time for the communication is minimal. In Figure 5.15 an ATC is shown for a logical sensor and range data server without a courier. It is clear that only every second range profile can be used since the logical sensor is busy interpreting the previous profile when the next profile must be collected. Figure 5.16 shows the result of using a courier as described and successfully processing all the range data. The code for the range courier can be found in Appendix C.2.

The interpretation of the range data is quite simple. A differential operator is used to locate a step edge in the data. The data around the step edge are then examined

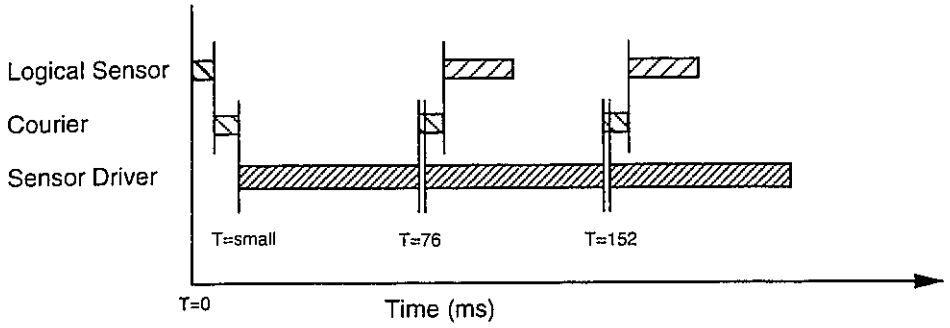

Figure 5.16: The Activity Timing chart with a courier.

to determine the slope of the surface containing the edge. These data-interpretation subroutines are specific to the expected data for edge following experiments. The data interpretation subroutines can be found in Appendix C.1.

5.3.4 Sensor Drivers and Machine Drivers Implementation

Sensor drivers and machine drivers have similar attributes in the SKORP RTS implementation. Each is written specifically for the input/output requirements of a particular device. They contain the following components:

- data structure definitions
- server
- command subroutines

The data definitions describe the formats of the messages that are passed to and from the server. The server is a Harmony task that is available to all other tasks and provides the service of interfacing with the external device. The server receives the request from client tasks, interprets the request, and calls on the command subroutines to carry out the request. The server will then reply to the client with the requested data (if

any) indicating that the request has been completed. The sensor driver code for the force-torque sensor and the laser range finder are included in Appendix D.

Two machine drivers were implemented for the PUMA robot. One of them uses the terminal port of the PUMA to send VAL II commands. The code for the VAL II server can be found in Appendix E.1. The second PUMA driver uses the ALTER port. This is a high-speed serial port that allows an external computer to have access to the control loop. This machine driver receives interrupts from the PUMA controller every 28 ms and must respond with a relative motion even if the response is not to move at all. This driver is lengthy and the complexities are specific to this particular robot controller. The code for this driver is not included in this thesis but is explained in the author's previous publications [34, 82].

5.4 Skills Implementation

In the course of this research, skills were developed that are representative of the types of skills that are interesting to the robotics industry. Implemented skills that do not use sensors include

- Move to Location
- Move Cartesian
- Relative Move - Cartesian
- Pick at Location
- Place at Location

- Palletize

Sensor-based skills that have been implemented include

- Align
- Rub
- Touch
- Press
- Follow Edge
- Standoff

Besides the skill icons three other types of icons were developed for the application programmer's environment. These icons, required for the structure of the operations, include

- Case and Merge
- Start and Restart
- Test the Range Finder or the Force-Torque sensors

One other icon is provided that represents the template for a skill that has yet to be created. This is provided for the system programmer to modify when creating new skill icons. All of the icons (except case and merge) are shown in Figure 5.17. Even with these few skills, operations can be developed in diverse application areas.

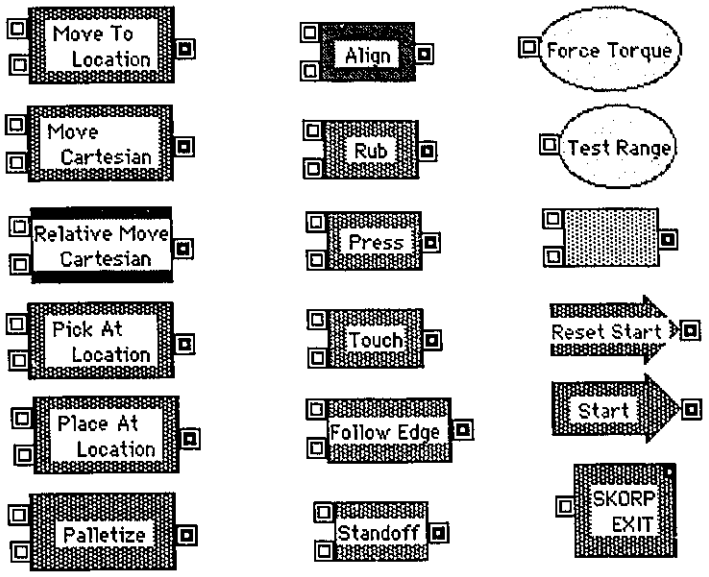

Figure 5.17: The icons that represent skills, testing, and miscellaneous functions.

5.4.1 Sensors

Two sensors are used in the experimental implementation. A wrist-mounted force-torque sensor and a wrist-mounted laser range finder. A photograph of these sensor is shown in Figure 5.18.

5.4.1.1 Six Axis Force-torque Sensor

There are many different types of force and tactile sensors used in robotics. Many of these devices are commercially available and have been used in research for many years [83]. These sensors, such as the one used in these experiments, are very well engineered and produce accurate and reliable data. The introduction of these sensors into industry is therefore not impeded by the sensors themselves; however, the use of

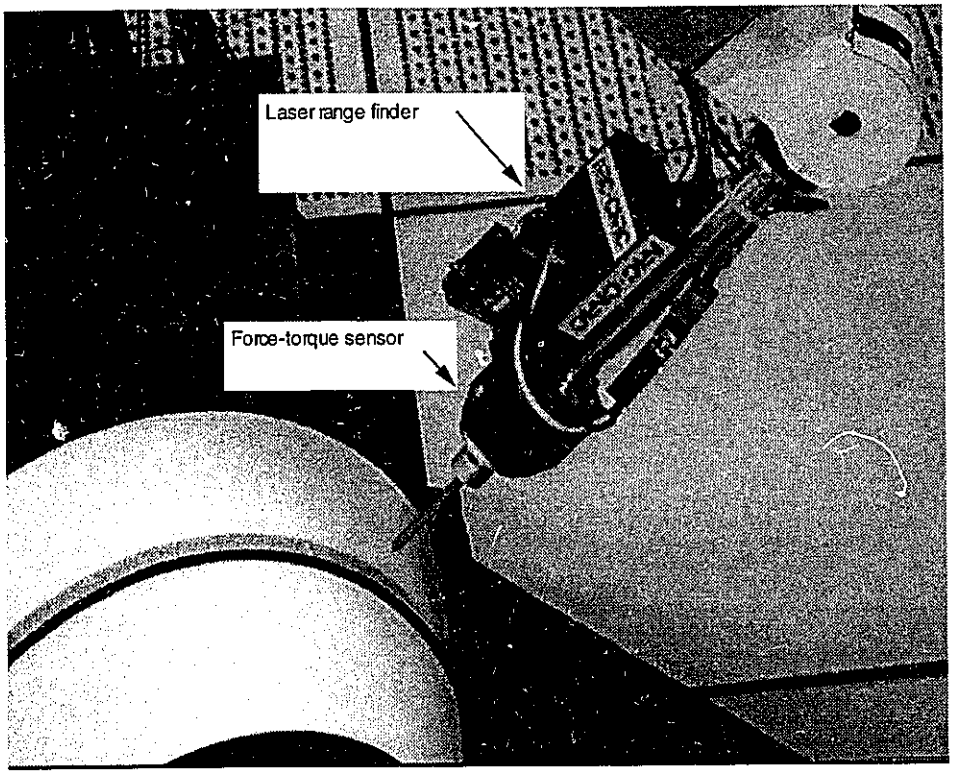

Figure 5.18: The sensors used in the experimental implementation.

force sensors has been limited by the difficulty in implementing reliable applications with the existing commercial robot controller hardware.

The force-torque sensor used in these experiments is available commercially [84] and will be described briefly. The ATI F/T sensor measures three forces and three torques in an orthogonal coordinate system. The force-torque vector is represented as $\{F_x, F_y, F_z, M_x, M_y, M_z\}$.

Physically this sensor is a small disk, 74 mm in diameter, 35 mm in thickness, and weighing 475 g. It is mounted on the wrist of the robot between the end of the wrist and the robot tool. The transducer is a single physical structure that transfers force and torque into analog strain gauge data that are digitized by the F/T controller. The sensor used in the current implementation is calibrated to sense forces in the range of ± 65 N and torques in the range of ± 5 N m. Resolutions in force are specified at 0.05 N in the X and Y directions and 0.10 N in the Z direction. Resolutions in torque are specified as 0.003 N m.

The force and torque data can be read from the controller through either a parallel or a serial port. In this implementation the serial port is used. The sensor driver found in Appendix D illustrates some of the commands available for interaction with the commercially provided controller. The most interesting command is to fetch the current force and torque readings from the sensor. This was timed to require 6 ms from a client task, i.e., a skill module. The data from this sensor have been found to be very reliable and no data averaging was used. A speed of 6 ms using the serial port was found to be fast enough to implement some interesting force control skills.

5.4.1.2 A Wrist-mounted Laser Range Finder

There are several advantages to using active range finding devices as opposed to passive optical sensors that use ambient or external light sources. Most important, these range finders are capable of providing accurate and explicit 3-D information relative to the end effector of a robot. Because the range data are compact, interpretation can be accomplished in real time without special-purpose image processing hardware. The range finder developed at the NRC [85] is specifically designed to be attached to end of arm tooling (EOAT) and has a depth of view from approximately 10 cm to 100 cm, with 40°of scanning in the Y direction.

The use of a double facetted mirror on both the path of the projected beam and on the return path to the CCD is referred to as *synchronized scanning* [86]. Because the reflected light is collected on the back side of the oscillating mirror, the sensor is always looking in the right direction to determine the range for that particular location. This scanning geometry has the advantage of being remarkably immune to ambient light, because the viewing direction is always aligned to the direction of the scan.

Physically the sensor must be very compact to be used on a light payload industrial robot. The prototype sensor has the following physical characteristics: 90 mm width, 140 mm length, 20 mm depth (this is 80 mm where the galvanometer protrudes), and 500 g in weight including the case. The sensor collects one range profile every 76 ms. The range data collected from the sensor are relative to the distances from the sensor to the surface, but must be calibrated for use in Cartesian coordinate space. Each device must be calibrated individually. Range data collected from a known target on a calibration bench are used to create a lookup table that contains the Cartesian coordinates corresponding to the raw range data. This method is described in detail in

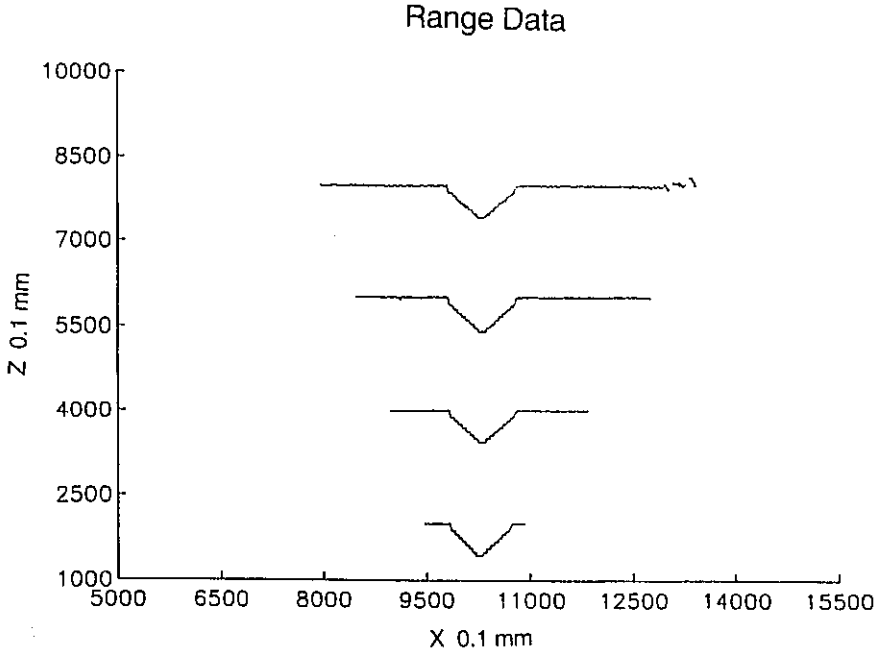

Figure 5.19: Sample range profiles collected at 20 cm, 40 cm, 60 cm, and 80 cm standoffs.

[87]. Figure 5.19 shows four range profiles of a small step pyramid collected at various standoffs to help the reader visualize the field of view of the range finder.

Determining and expressing the accuracy of the range data is quite difficult. The sampled points are not equally spaced in Cartesian space. The accuracy is affected by the conditions of the application, including specular surfaces. Stating the resolution is not usually interesting to the application specialist, since the resolution also varies nonlinearly and the accuracy is not as good as the resolution. The accuracy has been determined in the Z dimension by collecting range data in ideal conditions on a calibration bench. The errors are consistently measured at ± 0.1 mm at standoffs less than 15 cm, ± 0.2 mm at a standoff of 25 cm, and ± 2.0 mm at a standoff of 85 cm

deteriorating rapidly to about ±20 mm at a standoff of 100 cm.

5.4.2 *Rub* Skill

The *Rub* skill was selected to be described here because it is of considerable interest to industry for many applications. The generic skill *Rub* can be applied in the areas of polishing, deburring, cleaning, and any other application where it is required to maintain a constant pressure on a surface while moving the end effector. All skills where a robot is in contact with a solid object are difficult. Very small motions of the manipulator result in large changes in the forces applied, which can result in overcorrection of the position of the tool leading to unstable behaviour. There have been many strategies proposed to achieve stable force control and hybrid position/force control. This was a popular area of research as long ago as 1981 [88, 89].

The *Rub* skill presented here is an attempt to implement the simplest possible feedback algorithm using force feedback. The stability of the behaviour is then left up to the application programmer, who can vary the parameters to the skill and find the correct combination of correction speeds and forces to achieve the desired behaviour for the particular application.

The algorithm for the *Rub* skill can be summarized as follows: Move the end effector along a plane at a constant speed. Within the control loop read the current force in Z from the force torque sensor. If the current force is within the range specified in the dialog box, do not correct the position of the end effector in Z. If too much force is being applied, correct the position of the end effector in the negative Z direction. If too little force is being applied, correct the position of the end effector in the positive Z direction.

The interesting part of the algorithm is that the application programmer determines all the speeds and forces. As was shown in Figure 4.7, the parameters for the horizontal speed of the end effector, the correction speed, and the maximum and minimum forces can all be varied.

Experimentation with this algorithm, with a sponge attached to the end effector, found that a stable set of parameters can be determined experimentally for various conditions within 0.5 h. Although no serious attempt to make advances in force control algorithms was intended, it was found that this simplest of algorithms can perform at least as well as other algorithms that had taken several months to develop.

The code for the *Rub* skill can be found in Appendix F.1.

5.4.3 *Follow an Edge* Skill

A difficult realtime edge-following skill was selected to exemplify the development of skills using the wrist-mounted laser range finder. The objective of the skill is to:

1. position the end effector perpendicular to a surface,
2. follow an edge while moving in the X direction of the tool coordinate system at a constant speed,
3. maintain the position and orientation of the tool constant with respect to the edge in 6 degrees of freedom.

The target position of the tool (relative to the edge) is expressed as:

$$T = \{T_X, T_Y, T_Z, T_{R_X}, T_{R_Y}, T_{R_Z}\} \tag{5.1}$$

The software objects used in this skill corresponding to the lower three layers of Figure 3.1 are shown in Figure 5.21.

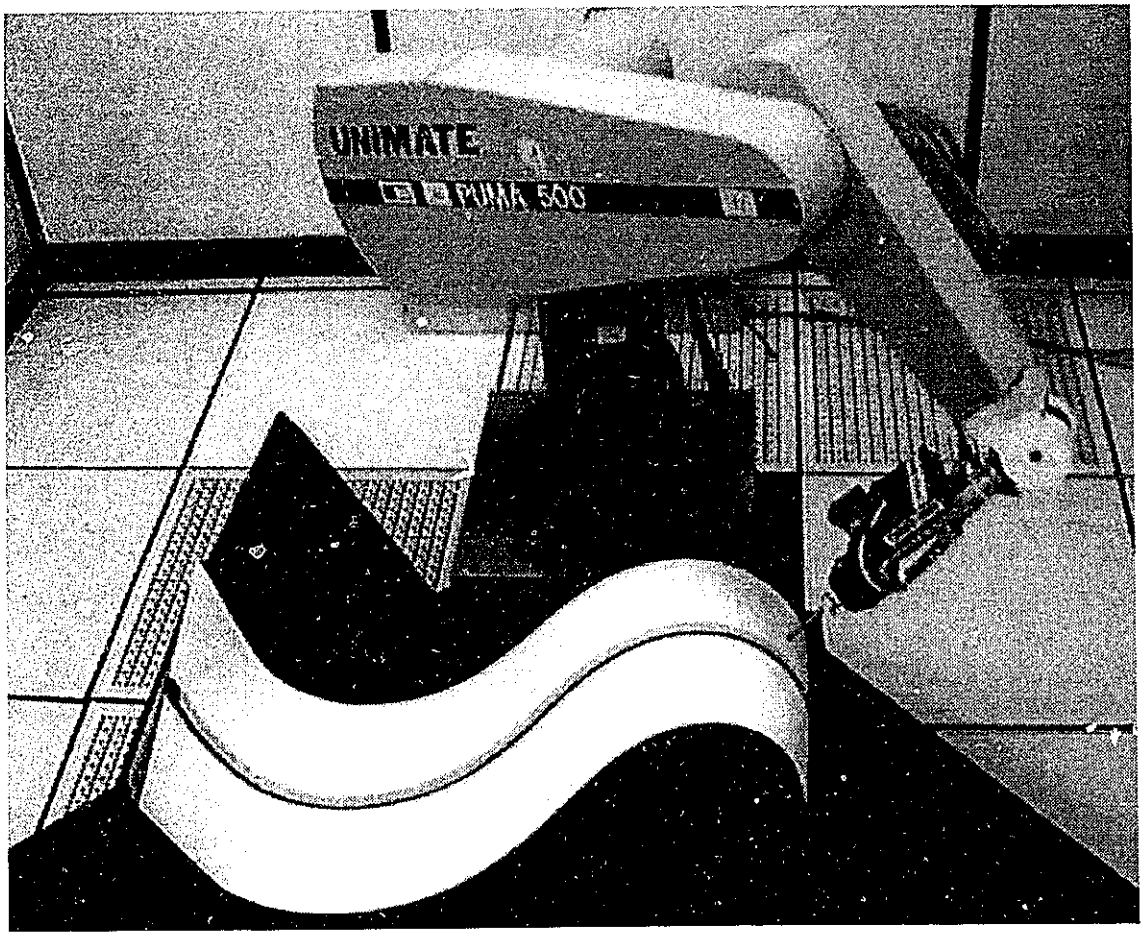

Figure 5.20: Photograph showing the PUMA robot executing the *follow edge* skill.

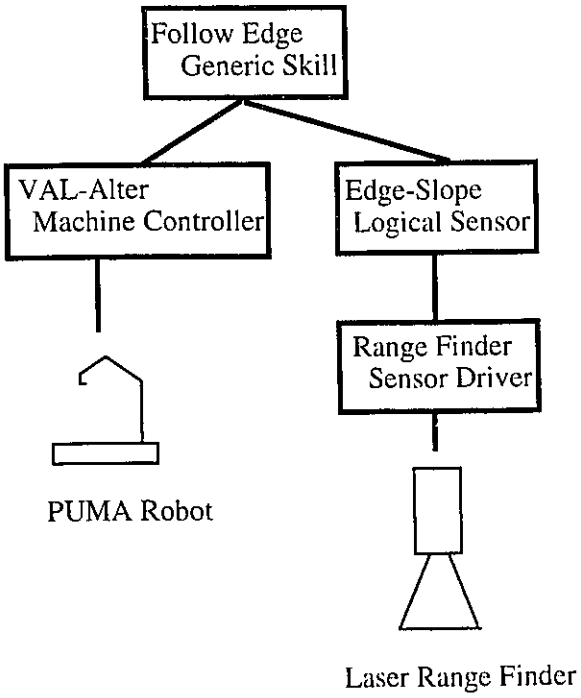

Figure 5.21: The software objects used in the *follow edge* skill.

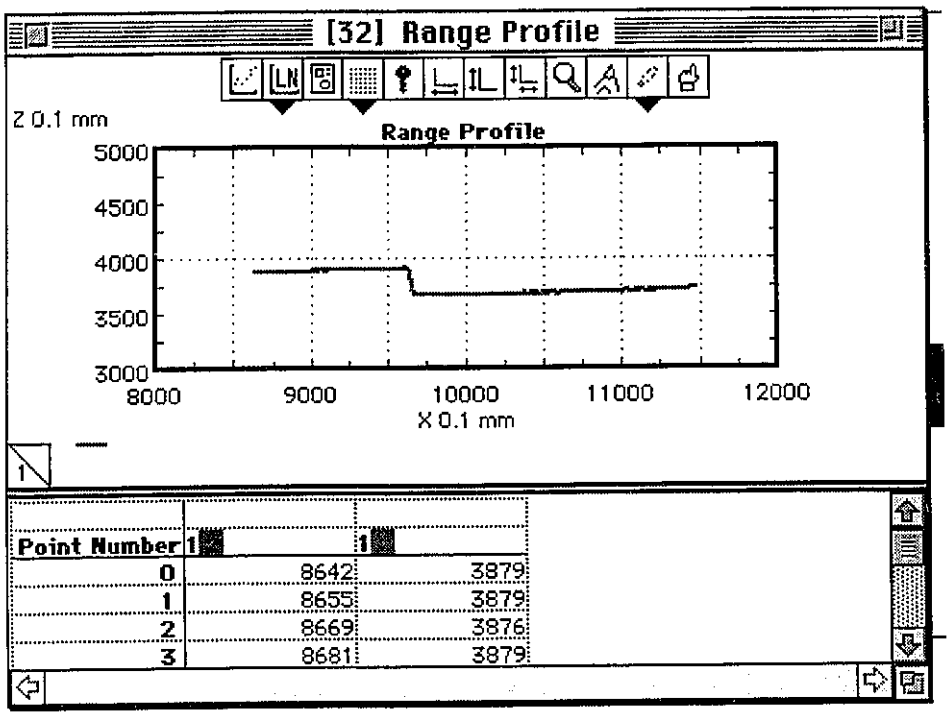

Figure 5.22: A sample range profile as presented to the programmer.

The *Range Finder* sensor driver collects the range data from a parallel port interface when it is requested by the *Edge-Slope* logical sensor. A sample of the range data is shown in Figure 5.22. The logical sensor determines the location of the edge by applying a simple one-dimensional differential operator.

The *Follow Edge* generic skill receives the location of the edge and calculates the amount of correction required in each dimension in the tool coordinate system. Since the range data are two dimensional, i.e., Y and Z, the corrections in these degrees of freedom are immediately available as:

$$C_Y = T_Y - E_Y \tag{5.2}$$

$$C_Z = T_Z - E_Z \tag{5.3}$$

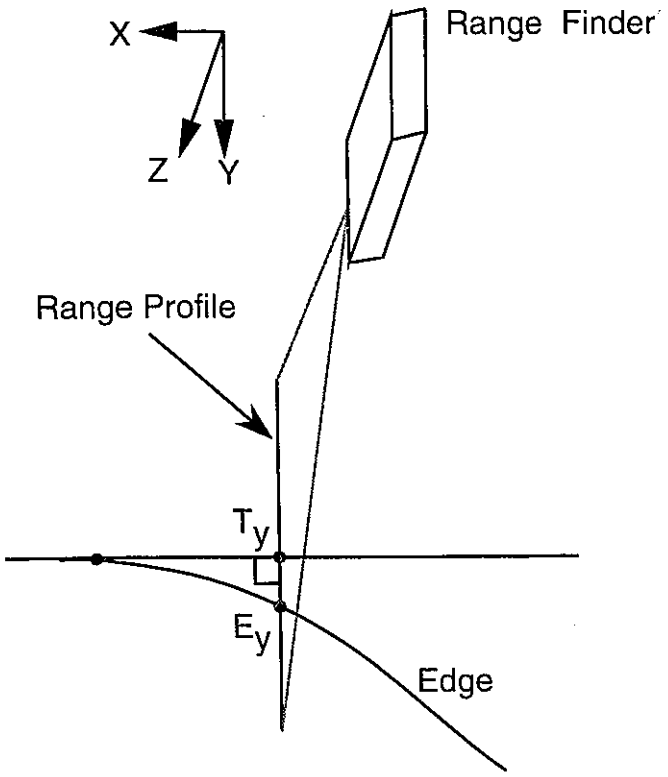

Figure 5.23: Correction in translation along the tool's Y axis.

where C_Y and C_Z are the required corrections in the Y and Z directions and E_Y and E_Z are the detected locations of the edge in Y and Z with respect to the tool. These corrections are illustrated in Figs. 5.23 and 5.24.

Calculating the slope of the surface along the range profile makes the correction around the X axis immediately available. This slope is calculated as:

$$E_{R_X} = \arctan\left(\frac{Z_2 - Z_1}{Y_2 - Y_1}\right) \tag{5.4}$$

where (Y_1, Z_1) and (Y_2, Z_2) are two YZ pairs selected from the range profile such that they are both on the same side of the edge. There is an assumption here that this choice will be the most suitable for the application at hand. The correction around the

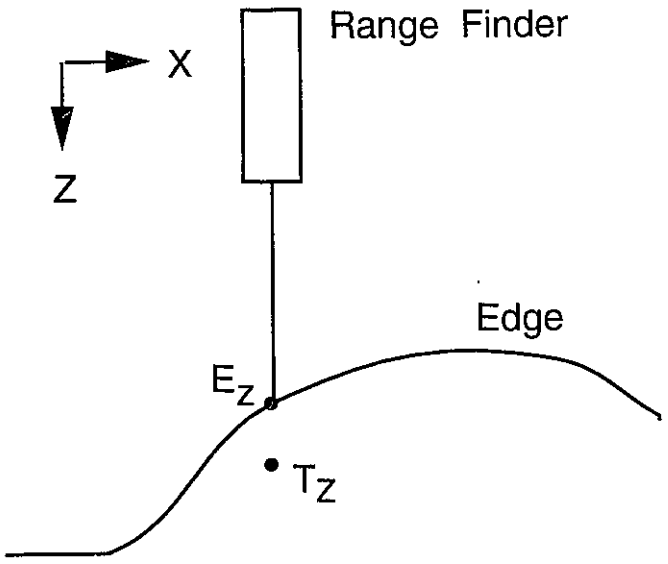

Figure 5.24: Correction in translation along the tool's Z axis.

X axis is calculated as:

$$C_{R_X} = T_{R_X} - E_{R_X} \tag{5.5}$$

This correction is illustrated in Figure 5.25.

The correction around the Y axis, C_{R_Y}, and the Z axis, C_{R_Z}, are calculated based on the recent corrections made in the Y and Z directions, which are stored in a history buffer. The C_Y and C_Z corrections are retained for the last h mm of motion in the X direction, i.e., along the edge. C_{R_Y} and C_{R_Z} are calculated as follows:

$$C_{R_Y} = \arctan\left(\sum C_Z/h\right) \tag{5.6}$$

$$C_{R_Z} = \arctan\left(\sum C_Y/h\right) \tag{5.7}$$

where $\sum C_Y$ and $\sum C_Z$ are the total corrections made in the Y and Z directions during the corresponding distance travelled in the X direction, h. These corrections are illustrated in Figs. 5.26 and 5.27.

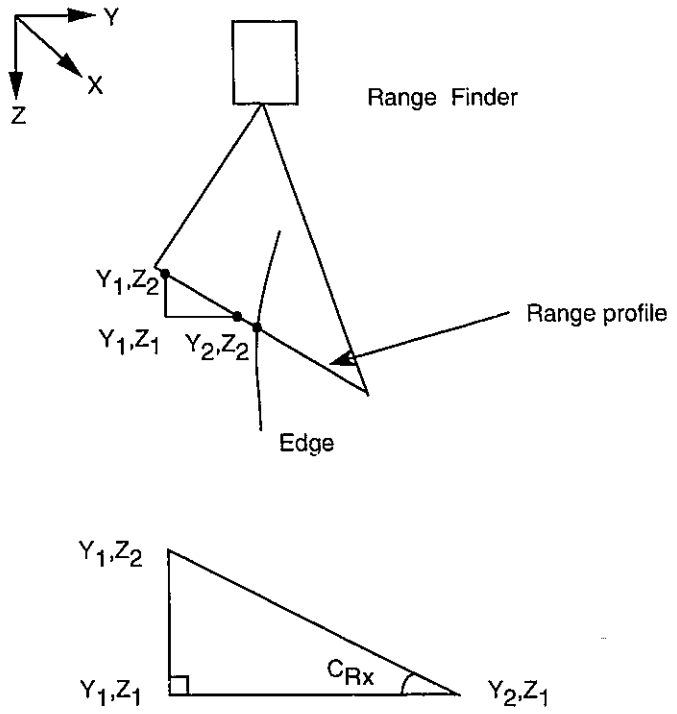

Figure 5.25: Correction in rotation around the tool's X axis.

Once the corrections are calculated for each degree of freedom, the actual motions sent to the machine controller must be limited to ensure safety and stability. Stability is difficult because it depends on three interrelated criteria: the speed of the tool along the edge, the standoff target distance T_Z, and the relative maximum allowable speed of the corrections in translation and rotation. For example, if the standoff is increased, the range data are less accurate, which may create a situation where the system overcorrects, and it is therefore necessary to reduce the maximum allowable speed for the translational and rotational corrections. If the correction in rotation is too large with respect to the corrections in translation the rotations will overcorrect, introduce even more errors in translation, and make the motion unstable. With an approximate mathematical model of the sensor accuracy and the relationships among the

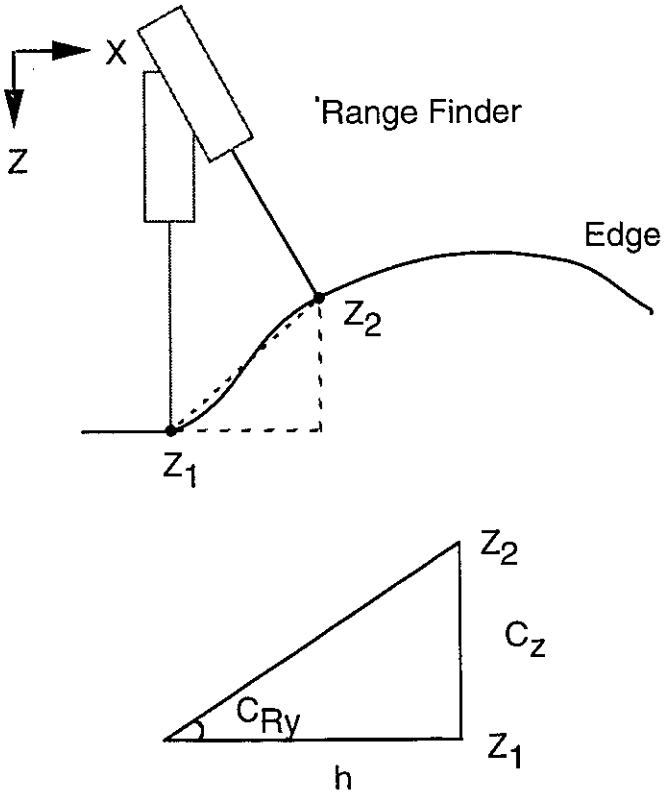

Figure 5.26: Correction in rotation around the tool's Y axis.

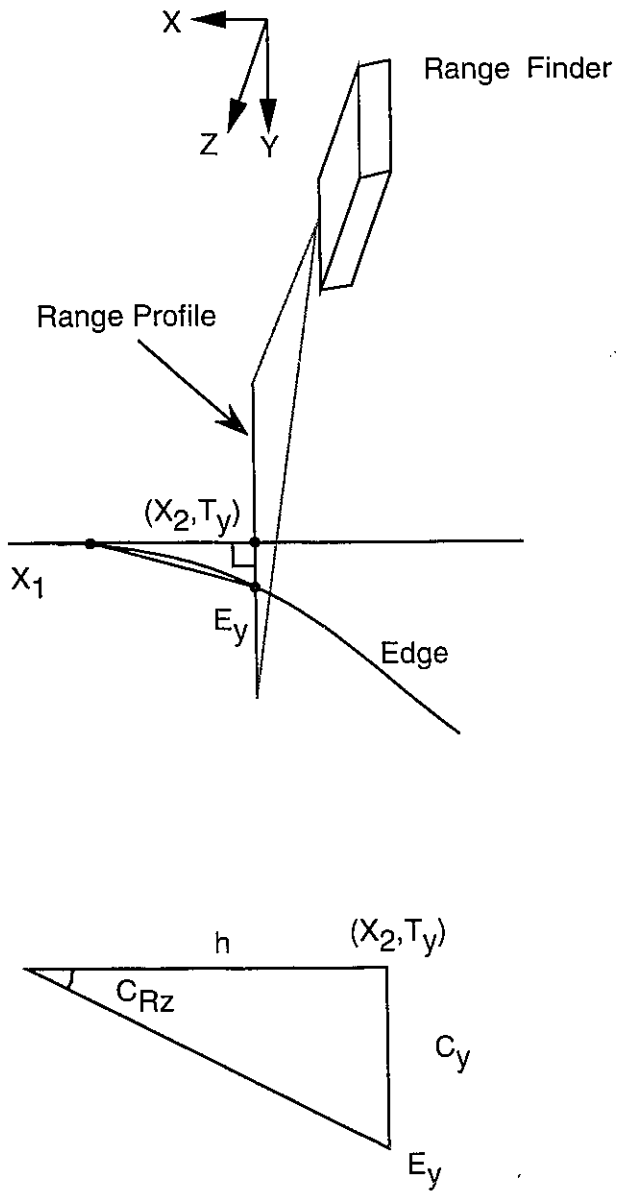

Figure 5.27: Correction in rotation around the tool's Z axis.

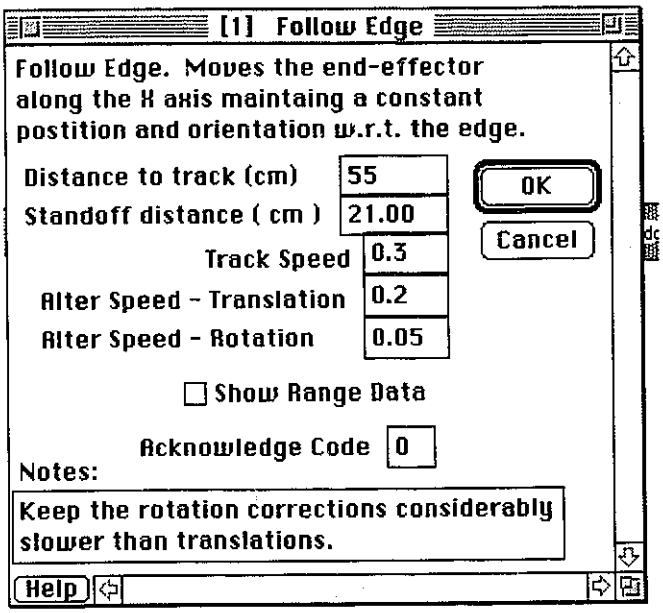

Figure 5.28: The dialog box used to modify the parameters to the *follow edge* skill.

control parameters, it may be possible to determine parameters for stable behaviour. This is difficult and often unexpected variables, such as position and speed dependent robot dynamics, will invalidate the formal solution. The SKORP model was used to create a generic *follow edge* skill with a dialog box that allows the rapid prototyping of a specific edge following operation. Figure 5.28 shows the dialog box that was created for this skill.

This is an example of how the iconic interface can be used to assist the *systems* programmer to determine the allowable parameters for the skill and to physically debug at the generic skill level.

The operation created to test the *Follow edge* skill is shown in Figure 5.29. This is a very simple operation that moves the end effector safely to the approximate location of the edge to be followed and then executes the *Follow edge* skill. Note that any unconnected icons left lying on the canvas will not be executed. The icon in Figure 5.29

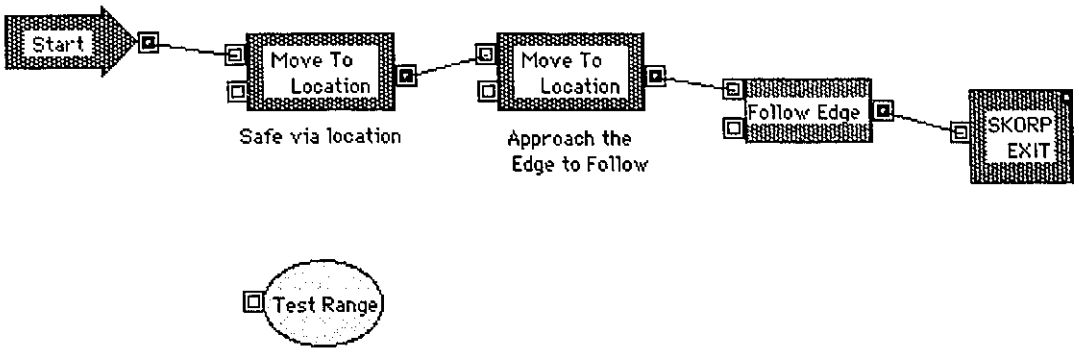

Figure 5.29: The simple iconic operation created to repeatedly test the *follow edge* skill.

labeled *Test Range* is used to test the range finder to ensure that it is functioning properly before the operation invoking the *Follow edge* edge skill is executed.

Chapter 6

Implications and Impact

6.1 Implications

6.1.1 Standardization

The ideas in this thesis reflect the belief that simple robot actions can be defined as stand-alone functional modules (skills) that, when combined correctly, produce useful behaviour for industrial robots. If these concept proves to be commercially useful, there are several practical implications for the field of robotics. Consider, for example, the ongoing struggle in the robotics community to standardize [90]. This has been difficult because of the interdependence of the component technologies. Any standard that defines a language or communication protocol either will be too general to be useful as a standard or will eliminate too many important manufacturers and hence be ignored. The only ISO recognized standard robot language, SLIM [91], is a BASIC-like language with some motion commands included. It is expected to be used for introductory robot educational purposes.

The attempts to create robot-independent languages have also received very little positive response from industry. Each robotic device has very different capabilities and characteristics. It is reasonable to expect that robots yet to be created will be even more varied. For example, redundant manipulators with seven or more axes are beginning to appear. These devices cannot possibly share a common control language with six-axis robots. Motion control that currently exists will be inadequate for the robot of the future.

It is believed that a skill-based robot programming system will create the possibility to standardize at the skill level. For example, many different robots will be able to execute the *Rub* skill, although with widely varying control mechanisms and ranges of parameter values. The skill templates can be used to compare robot capabilities in a standard way by comparing the available standard skills and their parameter ranges. Robotic systems may be purchased for particular applications, depending on the skill parameters, e.g., for grinding applications it is interesting to know how much pressure a robot can apply in a *Rub* skill. Current commercially available robots are specified in terms of payload, accuracy, repeatability, work envelope, and cycle time. The *cycle time* of a robot is a measure of how fast a robot can pick an object and place it at another location. This is a desperate attempt to find a common measure of performance for a variety of commercially available robots. The pick and place cycle time is almost completely useless as a means to determine if a particular robot is appropriate for an application but it is widely used because *any* comparative measure of commercially available devices is required. It would be more reasonable to specify the capabilities of a robot in terms of a range of parameters for a standard skill set.

6.1.2 Use in Research Laboratories

Some of the skills developed at NRC were mentioned briefly in the introduction. These skills took an average of approximately 1.5 person-years *each* to create. Almost all required new hardware and software for the interfaces to the machines and sensors. The interfacing and communication methods for each skill were different. A platform is required where the skills that are developed can be used in conjunction with one another, allowing more complex operations and preserving the skills in a functional state on upgradeable hardware. Skills implemented in SKORP required approximately two weeks each to code, once the other reusable modules were completed. The implication is that the implementation of sensor drivers, machine drivers, and logical sensor modules are more time consuming than the creation of the skills. Creating new skills that reuse the lower-level modules is dramatically simplified when compared to the tools and methods that were used before the SKORP computational model was created.

6.1.3 Telerobotics Research

The implementation described in this thesis provides an environment for rapid prototyping of skills, which may be applied to research for robots that are partially autonomous. For example, if a telerobotic system is to be programmed using supervisory control, the skills must be predictable and repeatable. The combination of skills to yield more complex operations is also facilitated, i.e., as the telerobotic system becomes more autonomous the primitives used by the operator become more powerful. The use of multiple sensors will be facilitated and the experimental programming required to test the development of the various skills using multiple sensors will be much faster. The implication here is that SKORP is an appropriate computational model

for research and development in areas of robotics other than industrial automation.

6.2 Other Controllable Machines

The relevance of the SKORP computational model for programming other devices that use sensor data to interact with the physical world, specifically, numerically controlled machines such as milling machines and lathes, is being explored. Programs controlling these devices are currently more inherently sequential than programs for robotic applications. This is largely because there are inherently fewer conditional operations to be performed and to date very few sensors are used to ascertain what has been done so far and what needs to be done. It is believed that this is currently changing, that sensors are being introduced to determine such things as the tooling required, exact position of the workpiece, tool wear, availability of coolants, and success of part cleaning operations.

In collaboration with the Institute for Advance Manufacturing Technology at the NRC, a mock-up was created of an operation in a system called SKOMP, SKills-Oriented Machine Programming.

Figure 6.1 shows an example of an operation to control a numerically controlled, sensor equipped milling machine. Figures 6.2, 6.3, 6.4, and 6.5 depict a few of the dialog boxes that were created to demonstrate this shop floor programming method. The community surrounding NC machine tools is more established than the robotics community. It was interesting to see the reactions of experienced NC machine programmers to concepts that are completely different from what has existed for many years. The inertia of the established methods makes the the introduction of radically different programming technologies a daunting task, but it is believed that the intro-

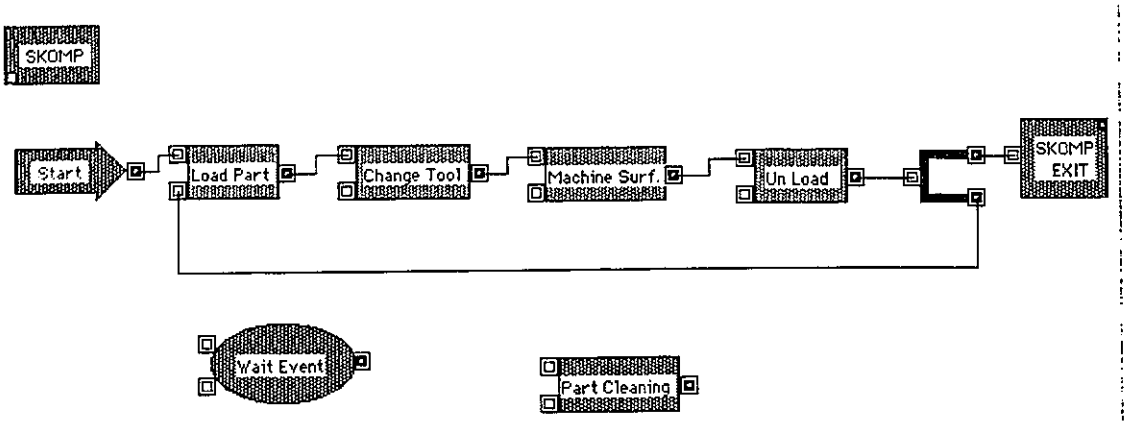

Figure 6.1: An example operation in SKOMP to control an NC milling machine.

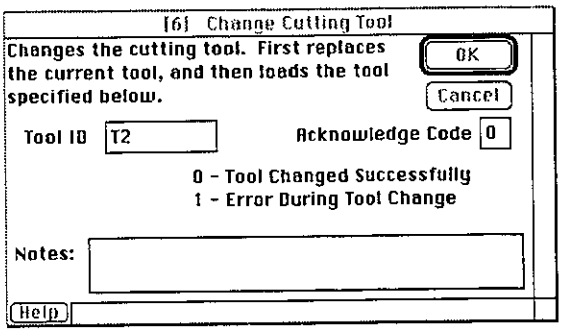

Figure 6.2: A dialog box for the *Change Tool* skill.

duction of sensors to these NC machines will make a high-level view of the machine operation a necessity.

6.3 Results

To date, we have published nine papers that are directly related to this research [52, 92, 93, 94, 95, 96, 97, 98, 99]. Most of the material in these papers has been presented in this thesis. This work has also resulted in patent applications in the United States,

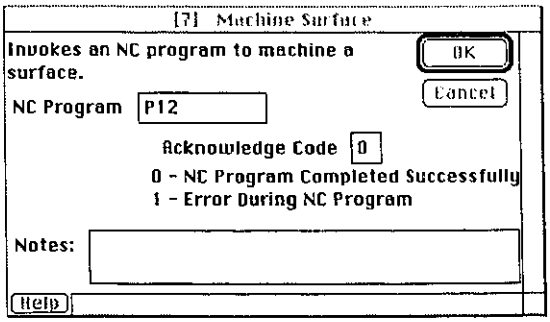

Figure 6.3: A dialog box for the *Machine Surface* skill.

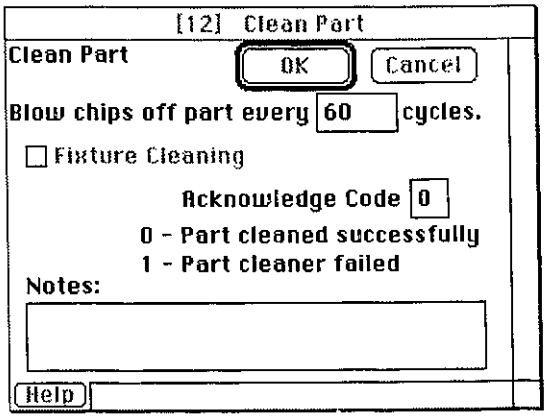

Figure 6.4: A dialog box for the *Clean Part* skill.

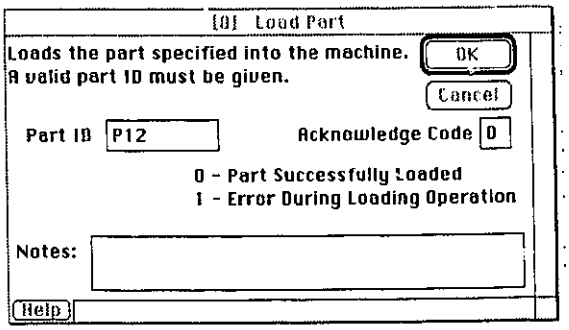

Figure 6.5: A dialog box for the *Load Part* skill.

Japan, and Canada.

It was mentioned that some of the skills themselves became interesting projects. The most complex skill presented in this thesis is the *Follow Edge* skill. It is also one of the most interesting to industry. Currently a Canadian company, Servo Robot, Inc., is implementing the methods that were developed during the creation of this skill. Although the industrial version of this skill does not use iconic programming, it was found that the high-level application programming environment and SKORP's RTS allowed us to develop and demonstrate the skill in a short period of time. It is easy to conclude that, given the available resources, this skill could not have been made of interest to industry without the tools provided in SKORP.

6.3.1 Commercialization of Iconic Robot Programming

A Canadian robot manufacturer, CRS Robotics, Inc., of Burlington, Ontario has signed a licence agreement with the NRC to develop a commercial product based on the SKORP model. The objective is to move from the current textual programming language, RAPL (Robot Application Programming Language), to iconic shop floor programming. A further objective is to create a programming environment that is identical for many different types of robots. CRS Robotics, Inc. manufactures five- and six-axis revolute joint robots, SCARA robots, and gantry robots. All of these types of robot are controlled by a single robot controller architecture based on transputers. The RAPL language is well known and for many years has been successfully used to program robots that do not use sensors.

Writing application code in a language that was specifically designed for ease of use conflicts with the objective of introducing sensors into applications. Sensor data

collection and interpretation require data structures, control constructs, and realtime tools that are not available in BASIC-like languages such as RAPL. The advantages of having a single shop floor programming environment are obvious, and this environment is curently being implemented using the SKORP model.

An example of how an application created in RAPL might be represented using SKORP is presented here. Figure 6.6 shows an incomplete sample of RAPL code.

6.3.2 Implementation in SKORP

The purpose of this example is to demonstrate how existing applications can be moved to an iconic programming environment. The real usefulness of the SKORP model is not demonstrated here since the SKORP model is most valuable for the creation of robot programs that are not currently feasible. However, it is interesting to look at how an application implemented using SKORP compares to textual code.

In Figures 6.7–6.10 the icons that have shadows encapsulate other icons. The icons that appear flat are primitive icons, which are popped open to reveal dialog boxes. Figure 6.7 contains a high-level representation of the entire operation. Six shapes of icons are used in the entire example. Start and Stop have the obvious meanings.

The icon labelled RSP is a specific icon used to set the Robot System Parameters such as the tool transform and software limits for the axes. Note that although the applications programmer is not expected to be a skilled computer programmer, an in-depth knowledge of the robot and its capabilities is required to set these parameters.

The icon containing an exclamation point (!) is an interrupt service routine (ISR). This icon is hanging off the execution path to indicate that it is not part of the functionality of the operation. ISRs are used to direct the program to a block of code that

```
AUTO_ST

        GOSUB HOMESEQ
        GOSUB INIT_SYS
        GOSUB MAIN
        GOSUB GOODNIGHT
        STOP

HOMESEQ

        ONPOWER
        HOMESEQ 1,2
        SPEED SLOW
        JOINT 1,90
        HOMESEQ 3,4,5,6
        READY
        RETURN

ON ERROR

        OUTPUT -ROUTER
        DELAY .5
        ABORT

MAIN

110     SELECT = BUSINPUT(3,3)
        GOTO ((BUSINPUT(3,3)*10+200)
190     ; NO RECOGNIZED PART
200     RETURN
210     GOSUB PART1
        GOTO 110
220     GOSUB PART2
        GOTO 110
230     GOSUB PART3
        GOTO 110
250     GOSUB PART5
        GOTO 110
260     GOSUB PART276
        GOSUB 210
270     GOSUB PART 842
        GOTO 110
```

```
INIT_SYS

        PASSWORD 255
        @REAL[44]=2.0943
        @REAL[52]=-2.0943
        @REAL[43]=1.6580
        @REAL[51]=-1.6580
        @REAL[114]=0.15
        GAIN 2, PID,6,.05,50
        GAIN 3, PID,6,.05,50
        GAIN 5, PID,25,0.03,100
        @REAL[116]=MAXLINS
        @REAL[117]=MAXLINA
        @REAL[118]=MAXROTS
        @REAL[119]=MAXROTA
        @ACCEL 0,0.25
        TOOL ROUTER
        PASE FDN
        ENABLE WRISTFLIP
        ONERR KILL
        RETURN
```

Figure 6.6: An sample of the BASIC-like RAPL language.

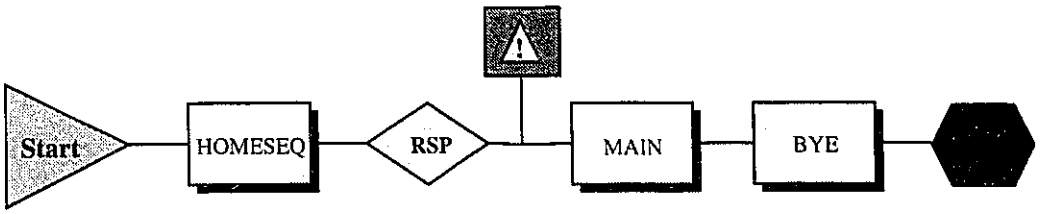

Figure 6.7: The SKORP equivalent representation of the AUTO_ST routine.

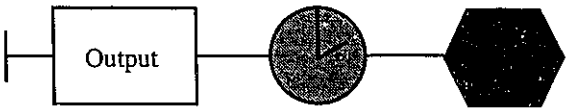

Figure 6.8: The iconic Interrupt Service Routine ON_ERROR.

will be executed in the event an error occurs. For example, if the robot arm strikes a solid object, overloading the joint motors, an interrupt will be generated and the flow of control will switch to the ISR regardless of the current skill being executed. The ISR's can be turned on and off by hanging these special icons off the path of execution. The ISR itself is represented by icons shown in Figure 6.8. The ISR is a control construct that was not presented in previous discussions of iconic programming. It was added here in an attempt to emulate what was done in the textual version of this program. However, it appears that adding this construct to the iconic programming introduced more complexity than desired, and since the ISR's are quite simple operations, they will be added to the individual skills. Thus, the iconic ISR will not be included in the further commercial development of SKORP.

The remaining icons are self-explanatory with two possible exceptions. The icon labelled WAIT in Figure 6.9 stops the flow of execution until a specific event occurs. In

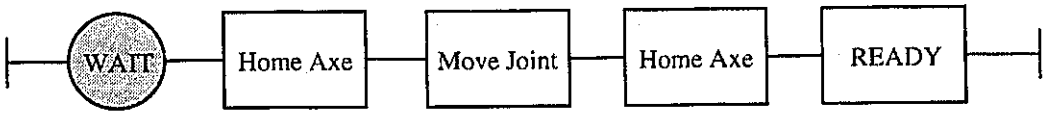

Figure 6.9: The iconic HOME_SEQ.

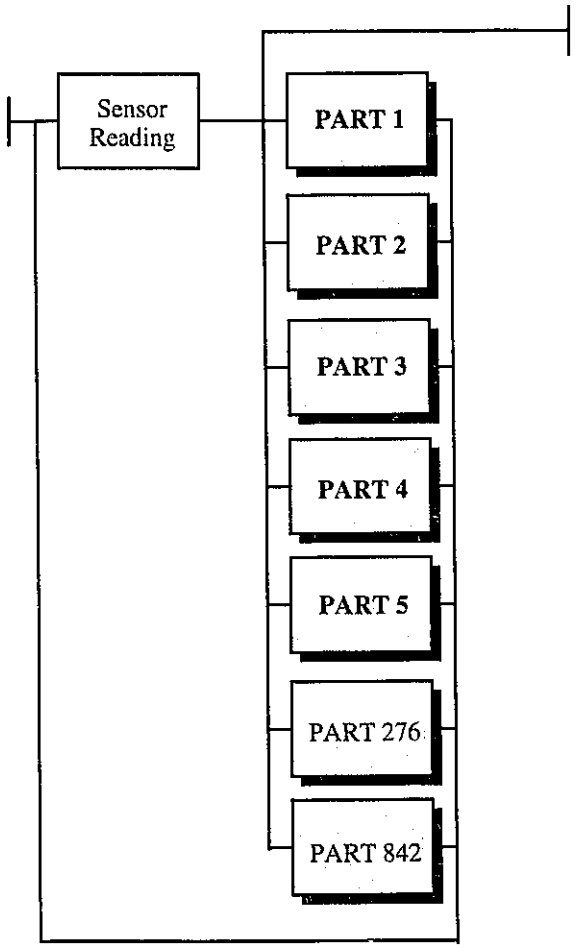

Figure 6.10: The iconic MAIN routine.

this case the WAIT indicates that the program will not continue until the arm power is turned on. If the power is already on, the execution continues immediately. The clock-like icon in Figure 6.8 represents a time delay of a duration specified in the dialog box for this icon. In Figure 6.10 the branching has been made implicit in the construction of the graph. The case icon appears to have been removed but in fact it has been incorporated into the connecting lines that can be popped open as if there were a case icon present. This change was made to remove everything from the network that does not represent a specific sequential step in the description of the operation.

A few examples of dialog boxes for icons in the preceding figures are shown in Figure 6.11. These dialog boxes are not very complicated and it is recognized that considerable effort will be required to design dialog boxes that are effective for long term use.

6.3.3 Conclusions on Using SKORP in this Case Study

One of the immediate observations of implementing a previously existing application in SKORP is that the separation of the application programmer and the system programmer requires an approach different from traditional robot systems development. In the example shown, the subroutines would be implemented as skills, and the skills would contain the interrupt service routines that are the responsibility of a professional computer programmer. In the textual code, the logic to accomplish the skill is mixed with the logic to deal with unexpected events. This resulted in the introduction of the ISR control construct into the example in an attempt to produce a more exact translation of what had been implemented in RAPL. An actual implementation of this application in SKORP would have no ISR construct in the application programmer

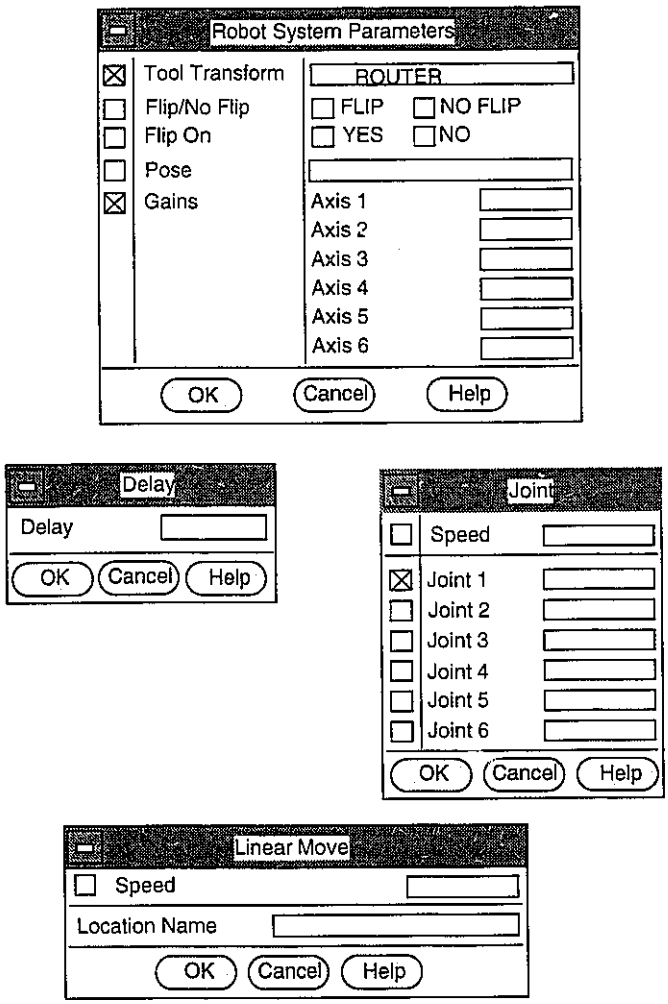

Figure 6.11: Some examples of dialog boxes which are presented to the programmer when the primitive icons are popped open.

environment.

The entire approach to creating robot applications where the application programmer is distinct from the system programmer will be different from the current single level of application development. The fact that problems will be solved differently leads to the conclusion that completely different problems will be solved. The only way to determine how this model will be most effectively used is to place it in the intended workplace and observe the results. These experiments will begin within the next year.

Chapter 7

Conclusions and Future Research Directions

7.1 Conclusions

In order to have a real impact on the competitive world of industrial automation, robot programming methods must be capable of taking advantage of advances in component technology as they are made. This can only be achieved by employing software development methods that are designed in a modular way from their conception.

The use of sensors in robotic systems presents a complexity barrier that has adequately been addressed in the literature. Although the issue of resource management has not been specifically addressed in this thesis, many of the issues are the same. Software design of operating systems, where the objective is to manage the physical and computational resources, shares many of the same problems as the resource management issues in robot workcell programming. Resource management is further discussed in [100]. The use of a modular computational model allows the complexity

to be isolated into solvable units. Other side effects of modular design include

- reusability
- accretion
- extendibility
- manageability

The SKORP computational model has been designed to create robotic systems with these charateristics.

Advanced robot programming involves multiple disciplines. Common use of robots that use sensors should not require engineers who are capable in all of these disciplines. A robot programming system that takes advantage of specialists or teams of specialists, is required to produce robotic applications in a manageable way. This is recognized in SKORP by separation of responsibilities, especially between the system specialists and application specialists. This separation is intrinsic to SKORP. The computing architecture for the application specialist, i.e., the commercially available workstation, provides the tools necessary to create a programming environment that permits attention to be focused on the issues in the physical robot habitat. The hardware used by the system specialist must be dedicated to providing tools for the creation of real-time software. Different computational paradigms are appropriate for these different computing architectures.

The iconic application programmer environment uses a context-free control flow programming method. A study of programming mechanisms lead to the conclusion that no robot programming tools exist that consider the attributes of the shop floor

robot programmer. The visual programming method presented in the SKORP model address the complexities of programming on the shop floor.

The context-free control flow model also introduces limitations on what can be expressed. The absence of data passing through the icons not only reduces the complexity, but of course reduces the possibility of passing information from one icon to the next. It will be frustrating for those trained in traditional programming techniques to devise skills that function independently, but can be connected to create useful operations.

Although robot simulations can solve a class of potential problems in robot programs, creating these simulations often requires more effort than the development of the program, and there are inevitably unexpected problems that will become apparent only when the robot is placed on line.

Debugging facilities at the application programming level are inherent in the design. Trial and error debugging is encouraged. It is inevitable that trial and error debugging will be used to fine tune the algorithms online during initial installation and when changes are made to the robot habitat. Providing tools for fine tuning algorithms will reduce the cost of modifying the production processes.

7.2 Future Research Directions

There are many possible directions for further developments and improvements to this work.

It has been suggested that the textual information that describes each skill can be combined to form a storyboard for an operation. This process has not been automated, but it is still being considered as a future research project. It is unclear if the storyboard would appear continuous and how conditional execution of skills can be incorporated

into such a textual description of a robot operation.

It has been suggested that multiple execution icons can be used in large workcells with each execution token representing the work of a different machine. The risk of introducing this to the application programming environment is that a layer of complexity is added to the shop floor. Another alternative to multiple execution tokens is to allow multiple operation windows to execute in parallel. This appears to be a more appropriate solution to the multiple machine workcell, but will require investigation.

Only one robot has been used in these experiments. The clear implication of the previous discussion on standardization of skills is that many different robots will have the capability to execute the same generic skills with variation in the valid range of parameter values. Experiments to implement skills that function on multiple robots should be conducted. These experiments are required to make definitive conclusions on the possibility of replacing machine driver modules while maintaining the remaining code for skills. The compromise to this is that the icon will remain the same for many types of robots, but all of the code on the RTS must be modified. Although this would be a far less satisfactory conclusion, it would still mean that a single application programming environment could be ported to a variety of robots, as opposed to the current situation where every robot manufacturer uses a different application programming method.

In addition to coordination of the software modules on the RTS, we are considering the use of coordination languages for higher-level robot programming. The coordination of the skills, for example, may be possible through the use of tools similar to those described for coordinating the software modules that make up the skills.

Improvements to individual skills are being requested. These skills were created

largely for demonstration of the computational model. Of particular interest is the *Follow Edge* skill, which is being implemented commercially. Although there is currently no measure of accuracy for this skill, accuracy specification is a requirement for marketing robot welding systems. A project is underway to use a milling machine to create an undulating surface NC machined in aluminum with a simulated weld seam that travels over this surface. The objective is to compare an accurate model of the surface, which can be extracted from the milling machine tools paths, to the actual path that the robot follows in six degrees of freedom.

New sensor-based skills have been proposed by several academic and industrial researchers. Using the prototype system that has been created to research new skills will begin immediately.

A commercial version of the iconic programming interface is currently being created with patents pending in Canada, the United States, and Japan.

It will be interesting to study the responses of the users when this tool is actually being used in production. The application programmer's interface has already evolved on the basis of comments from potential users and this is certain to continue.

Appendix A

ModL Code

A.1 ModL Code for the Rub Skill Icon

```
Real FTdata[];
String IntToString(integer thisint);  // forward declarations
Procedure Get_Response();
Procedure GetAckCode();
Procedure GetAndPlotFT();

procedure DoSkill()
{
    if (animationOn)
    {
        animationShow(1);
        animationShow(2);
        waitNTicks(5);
    }

    if(!GlobalInt0)  // If not testing then go do it
    {
        SerialWrite(FALSE,"rub "+" "+direction+" "+distance+" "
        +HSpeed+" "+VSpeed+" "+Maxforce+" "+Minforce+" "+Stopforce+" "
        +1000+" "+ ShowData+StrPutAscii(13));

        // either present an error to programmer or continue
        Get_Response();
    }

    if (animationOn)
    {
        animationHide(1, FALSE);
        animationHide(2, FALSE);
    }

    SendMsgToInputs(itemOut);  // pass control to the next skill block

} // end of do rub skill procedure


Procedure Get_Response()
{
    integer count,i;
```

```
    integer AckOrNak, Ack, Nak, Data;  // One of these will be returned
    integer response[100];
    string onechar,errorstring;

    AckOrNak = FALSE;
    response[0] = 0;
    Ack = 35;          //#
    Nak = 33;          //!
    Data = 62;          //>


    While(!AckOrNak)
    {
        count = 1;
        FSRead(-6, count,response);

        onechar = IntToString(response[0]);

        if(StrGetAscii(onechar) == Nak)
        {
            count = 50;          // error messages are 50 chars long.
            FSread(-6,count,response);

            errorstring = "";
            for(i=0; i<count; i++)
            {
                errorstring = errorstring+IntToString(response[i]);
            }

            UserError(StrPart(errorstring,0,count));
            AbortAllSims();
        }
        if(StrGetAscii(onechar) == Ack)
        {
            GetAckCode();
            AckOrNak = TRUE;
        }
        if(StrGetAscii(onechar) == Data)
        {
            GetAndPlotFT();
        }
    }
}

Procedure GetAckCode()
{
    String AckString;

    AckString = SerialRead(FALSE);
    // Put the AckCode in the dialog Window.
    AckCode = StrToReal(AckString);

    // Set the global variable to the Ack code 0-3.
    GlobalInt1 = Ackcode;
}

Procedure GetandPlotFT()
{
    Integer i;
    Integer count, allbytes[2];
    string tempstring;
```

```
    MakeArray(FTdata,6);

    count = 8;

    // get the data out of the integer array into a float array

    for(i=0; i<6; i++)
    {
        FSRead(-6,count,allbytes);
        tempstring = IntToString(allbytes[0]) + InttoString(allbytes[1]);
        //usererror(tempstring);
        FTdata[i] = StrToReal(tempstring);
    }


    // Now make a plot.

    InstallAxis(0, "Force Torque Data", "X     Y     Z     Xt    Yt    Zt",
                   False,1,6,
                   "",False,-100,100,
                   "",False,0.0,0.0,blackpattern,cyanColor,200);

    ChangePlotType(0,6);  // Bar plot

    InstallArray(0, 0, "", FTdata, 0.0, 1.0,
                   6, 0, blackPattern, cyanColor);

    ChangeSignalWidth(0,0,10);

    ShowPlot(0,"Force Torque Data");

}

// The on itemIn message handler is received when a message is sent
// to the input connector.
// This message means that there is an item available to
// be taken.

on item1In
{
    DoSkill();
}

on item2In
{
    DoSkill();
}

on Simulate
{
}

on checkData
{

    getSimulateMsgs(FALSE);

    animationRectangle(1);
    animationRectangle(2);
    animationColor(1, 60000, 60000, 60000, 1);
    animationColor(2, 60000, 60000, 60000, 1);
    animationHide(1, FALSE);
```

```
    animationHide(2, FALSE);
}


on initSim
{
}


** Set the default values for the items in the dialog box.

on createBlock
{
    Direction = 0;
    StopForce = 700;
    Distance = 10;
    minforce = -30;
    maxforce = -170;
    HSpeed = .2;
    Vspeed = .8;
    ShowData = 0;
}

on endSim
{
}

** This procedure converts a 4 byte integer into a 4 character string,
**   and returns the string.

String IntToString(Integer thisint)
{
    integer temp;
    string result;

    // isolate the first 8 bits
    temp = BitOr(thisint,4278190080);        // 0xff000000
    temp = BitShift(thisint,-24);
    result = StrPutAscii(temp);              // first character
    temp = BitAnd(thisint,16711680);         // 0x0ff0000
    temp = BitShift(temp,-16);
    result = result + StrPutAscii(temp);     // second character
    temp = BitAnd(thisint,65280);            // 0x00ff00
    temp = BitShift(temp, -8);
    result = result + StrPutAscii(temp);     // third char
    temp = BitAnd(thisint,255);
    result = result +StrPutAscii(temp);
    return(result);
}
```

A.2 ModL Code for the CASE Icon

```
// The CASE 4 icon receives a signal from the previous skill
// and branches based on the value of the acknowledge code
// that is stored in the global variable globalInt1.

on itemIn
{

    // check the value of the global variable
    // which represent the acknowledgement
    // codes and exit based on this value.

    if(GlobalInt1 == 0)
    {
        if (animationOn)
        {
            animationShow(1);
            waitNTicks(20);
            animationHide(1, FALSE);
        }

    // pass control through top connector
        SendMsgToInputs(aOut);
    }
    else if(globalInt1 == 1)
    {
        if (animationOn)
        {
            animationShow(2);
            waitNTicks(20);
            animationHide(2, FALSE);
        }

    // pass control through second connector
        SendMsgToInputs(bOut);
    }
    else if(globalInt1 == 2)
    {
        if (animationOn)
        {
            animationShow(3);
            waitNTicks(20);
            animationHide(3, FALSE);
        }

    // pass control through third connector
        SendMsgToInputs(cOut);
    }
    else if(globalInt1 == 3)
    {
        if (animationOn)
        {
            animationShow(4);
            waitNTicks(20);
            animationHide(4, FALSE);
        }

        // pass control through fourth connector
        SendMsgToInputs(dOut);
    }
```

```
    else
    {
        UserError("The result code from the previous Skill is
          not within range for the CASE 4 icon.");
        AbortAllSims();
    }
}


on Simulate
{
    Beep();
}

on checkData
{
    animationRectangle(1);
    animationColor(1, 37000, 39000, 65500, 3);
    animationRectangle(2);
    animationColor(2, 37000, 39000, 65500, 3);
    if (animationOn)
    {
        animationHide(1, FALSE);
        animationHide(2, FALSE);
    }
}


** Initialize any simulation variables.
on initSim
{
    getSimulateMsgs(FALSE);
}

on endSim
{

}

on createBlock
{
}
```

Appendix B

Harmony Task Templates

```
/* skorptemp0.c
*    Processor 0
*    Defines the task templates resident on this processor.
*/

extern task main();
extern task _Directory();
extern task _Gossip();
extern task _Tty_server();
extern task _SPi_mc68901();
extern task _SPo_mc68901();
extern task _Dbg_control();
extern task _Dbg_shadow();
extern task _Dbg_agent();
extern task _Dbg_interpreter();
extern task _Dbg_bp_ui();
extern task _Clock_server();
extern task _Clock_notifier();
extern Logical_Sensor Nearest_Point();
extern Logical_Sensor Slope();
extern Logical_Sensor Edge_point();

extern task Debug();

extern void _Mc68901_int();

uint_32 _Pnumber = 0;

struct TASK_TEMPLATE _Template_list[] =
  {
    { MAIN,             main,            6000,       7},
    { DIRECTORY,        _Directory,      1000,       7},
    { GOSSIP,           _Gossip,         2000,       5},
    { TTY,              _Tty_server,     2000,       5},
    { TTI,              _SPi_mc68901,    1000,       0},
    { TTO,              _SPo_mc68901,    1000,       0},
    { DBG_CONTROL,      _Dbg_control,    2000,       7},
    { DBG_SHADOW,       _Dbg_shadow,     1000,       7},
    { DBG_OAGENT,       _Dbg_agent,      1000,       5},
    { DBG_INTERPRETER,  _Dbg_interpreter,5000,       7},
```

```
    { DBG_BP_UI,          _Dbg_bp_ui,       2000,        7},
    { CLOCK,              _Clock_server,    1000,        6},
    { CLOCK_NOTIFIER,     _Clock_notifier,   700,        0},
    { DEBUG_TASK,         Debug,             500,        6},
    { NEAREST_POINT,      Nearest_Point,    4000,        6},
    { SLOPE,              Slope,            5500,        6},
    { EDGE_POINT,         Edge_point,       8000,        6},
    { 0, 0, 0, 0}
  };

struct INT_PAIR _Interrupt_list[] =
{
    { 5,  _Mc68901_int },
    { 0,  0 }
};



/* skorptemp1.c
*    Processor 1
*    Defines the task templates resident on this processor.
*
*/

extern task _Ehvt_server();
extern task _Ehvtip_mc68901();
extern task _Ehvtic();
extern task _Sspo_mc68901();
extern task _Dbg_agent();
extern task range_server();
extern task range_courier();
extern task valserver();

extern void _SMc68901_int();

uint_32 _Pnumber = 1;

struct TASK_TEMPLATE _Template_list[] =
  {
      {EHVT,            _Ehvt_server,        1500,    6},
      {EHVTIP,          _Ehvtip_mc68901,      850,    0},
      {EHVTIC,          _Ehvtic,             1000,    6},
      {EHVTO,           _Sspo_mc68901,        750,    0},
      {RANGE_SERVER,    range_server,        4000,    5},
      {RANGE_COURRIER,  range_courier,      4000,    6},
      {VAL_SERVER,      valserver,           2000,    6},
      {DBG_1AGENT,      _Dbg_agent,          1000,    7},
      {0,               0,                      0,    0}
  };

struct INT_PAIR _Interrupt_list[] =
  {
    { 5, _SMc68901_int },
    { 0, 0 }
  };



/* skorptemp2.c
*    Processor 2
*    Defines the task templates resident on this processor.
*/
```

```
extern task _Out_packet();
extern task _In_packet();
extern task _Dbg_agent();

extern task Alt_handler();
extern task Alt_courier();
extern task Alt_server();

extern void _VAVP_int();  /* interupt handler for ALTER */

uint_32 _Pnumber = 2;

struct TASK_TEMPLATE _Template_list[] =
  {
      { _OUT_PACKET,   _Out_packet,   800,  4},
      { _IN_PACKET,    _In_packet,    800,  3},
      { ALTSERV,       Alt_server,   1100,  5},
      { ALTCOUR,       Alt_courier,   800,  6},
      { ALTER,         Alt_handler,  1300,  7},
      { DBG_2AGENT,    _Dbg_agent,   1000,  8},
      { 0, 0, 0, 0}
  };

struct INT_PAIR _Interrupt_list[] =
  {
    { 7, _VAVP_int},
    { 0, 0 }
  };


/* skorptemp2.c
*    Processor 3
*    Defines the task templates resident on this processor.
*/

extern task _Sercom_server();
extern task _Sercomip_mc68901();
extern task _Sercomic();
extern task _Sspo_mc68901();

extern task _Dbg_agent();

extern task ft_server();

extern  _SMc68901_int();    /* Serial communication interrupt handler. */

uint_32 _Pnumber = 3;

struct TASK_TEMPLATE _Template_list[] =
  {
      { SERCOMM,       _Sercom_server,      1500,    5},
      { SERCOMIP,      _Sercomip_mc68901,    850,    0},
      { SERCOMIC,      _Sercomic,            750,    6},
      { SERCOMO,       _Sspo_mc68901,        750,    0},
      { DBG_1AGENT,    _Dbg_agent,          1000,    5},
      { FT_SERVER,     ft_server,           3000,    6},
      { 0,             0,                      0,    0}
  };

struct INT_PAIR _Interrupt_list[] =
  {
```

```
  { 5, _SMc68901_int },
  { 0, 0 }
};
```

Appendix C

Logical Sensors Code

C.1 Edge-slope Logical Sensor Code

```
/* edge_slope.c
*  This is a logical sensor that  determines the
*  point in the range profile that represents a step edge in
*  the profile. Currently it assumes that there is only one.
*  It is used in realtime edge tracking experiments.
*
*/

/** Logical Sensor Edge_slope **/

#define MASK_SIZE    15  /* surfels */
#define MIN_JUMP     5    /* mm */
#define EDGE_THRESH  200

typedef struct EDGE_RQST
{
  struct  STD_RQST  STD_REQUEST;
  int_16            plot_data;

};

typedef struct EDGE_RPLY
{
  struct STD_RPLY  STD_REPLY;
  int_32           x,z;
  float            slope;
};


Logical_Sensor Edge_slope()
{
  extern void step_edge();
  extern void calc_edge_slope();

  uint_32  requestor, range_courier_id;

  struct  RANGE_RQST range_rqst;
  struct  RANGE_RPLY range_rply;
```

```
 struct  EDGE_RQST edge_rqst;
 struct  EDGE_RPLY edge_rply;

 /* The logical sensors do nothing when they are created,
 except maybe create a courier for the device driver and
 wait for a request from a client */

 range_courier_id = _Create(RANGE_COURRIER);

  /* only the skill that created this logical sensor can make requests */
 requestor = _Father_id();

 for(;;)
 {
   edge_rqst.STD_REQUEST.MSG_SIZE = sizeof(struct EDGE_RQST);
   edge_rply.STD_REPLY.MSG_SIZE = sizeof(struct EDGE_RPLY);

   /* Get a request from the skill  ....
       logical sensors spend most time blocked here */

   _Receive( (char *) &edge_rqst, requestor );

   range_rqst.STD_REQUEST.MSG_TYPE = GET_PROFILE;

   /* Get the range data from the range courier */
   range_rqst.STD_REQUEST.MSG_SIZE = sizeof(struct RANGE_RQST);
   range_rply.STD_REPLY.MSG_SIZE = sizeof(struct RANGE_RPLY);

   _Receive( (char *) &range_rply, range_courier_id);
   range_rqst.STD_REQUEST.MSG_TYPE = GET_PROFILE;

   /* Unblock the courier */
   _Reply( (char *) &range_rqst, range_courier_id);

   /*  find the edge point */

   step_edge(&(range_rply.profile), &edge_rply.x, &edge_rply.z,
               edge_rqst.plot_data);

   /* find the slope */

   calc_edge_slope(&(range_rply.profile), &edge_rply.slope);

   /* reply to the requestor with the edge information */
   _Reply( (char *) &edge_rply, requestor );

  }  /* for(;;) */
};

void calc_edge_slope(profile, slope)
struct PROFILE *profile;
float *slope;
{

  int_32 start_x, stop_x, start_z, stop_z, i;
  float rise, run;


  /*   find the slope of the surface  */

  *slope = 0.0;
```

```
  /* starting at the 25th surfel take the first one
       that is valid as the start point */

  i=25;
  while(!profile->invalid[i]  && i < 50 ) i++;

  start_x = profile->x[i];
  start_z = profile->z[i];

  /* now start at the 75th surfel slooking for a valid end point */

  i=75;
  while(!profile->invalid[i]  && i < 100 ) i++;

  stop_x = profile->x[i];
  stop_z = profile->z[i];

  rise = (float) (stop_z - start_z);
  run = (float) (stop_x - start_x);

  *slope = RAD_TO_DEG * (float) atan((extended)(rise/run));
}

/* step_edge.c
*  This is a subroutine that finds the step edge in the range profile.
*  The assumption is that there is only one, else it will return the
* first one that is found.
*
*/

void step_edge(profile, x,z,plot_data)
struct PROFILE *profile;
int_32 *x, *z;
int_16 plot_data;
{
  extern int_32 findmaxpoint();
  extern void derivative();

  int_32 i, besti, bestz;

  struct PROFILE first_deriv;


  /* take the first derivative of the profile */
  derivative(profile, &first_deriv);
  besti = findmaxpoint(&first_deriv);
  if(plot_data == 1)
      putprofile(&first_deriv);


  /* get the Z value by looking at the nearby surfels....
    want the top of the edge not bottom, nor along the side. */

  bestz = 99999;
  if(besti > 12 && besti < SURFELS - 12)
    for(i = besti-13; i< besti+13; i++)
      if((profile->z[i] < bestz)  && (!profile->invalid[i]) )
      {
        bestz = profile->z[i];
        *z = profile->z[i];
        *x = profile->x[i];
      }
```

```
  if(plot_data == 1)  /* sends back data to Extend for the Mac plot */
  {
    putprofile(profile);
    overlay(1, x,z);        /* overlay 1 point on the open plot */
  }
}

/* find the index of the  first large derivative value in the profile */

int_32 findmaxpoint(profilein)
struct PROFILE *profilein;
{
  int_32 i;

  for( i=0; i< SURFELS; i++)
    if( ABS(profilein->z[i]) > EDGE_THRESH)
      return(i);    /* take the first point that is over the threshold */
}


void derivative(profilein, profileout)
struct PROFILE *profilein, *profileout;
{

  /* take the derivative of the z values in the first profile
     and return the result in the second */

  int_16 i,j;
  int_32 difference;

  /* let the X values be the same - for plotting only */

  for(i=0; i<SURFELS; i++)
    profileout->x[i] = profilein->x[i];

  for(i = MASK_SIZE/2; i< SURFELS - MASK_SIZE/2; i++)
  {
    difference = 0;
    for(j=1; j<MASK_SIZE/2; j++)
      if(!(profilein->invalid[i-j] || profilein->invalid[i+j]))
        /* find the derivative at this point */
        difference += (profilein->z[i-j] - profilein->z[i+j]);

    profileout->z[i] = difference;
  }

  /* fill in the ends of the profile out so it doesn't
      find edges at the ends */

  for(i = 0; i< MASK_SIZE/2; i++)
    profileout->z[i] = profileout->z[MASK_SIZE/2];

  for(i = SURFELS - MASK_SIZE/2;  i<SURFELS; i++)
    profileout->z[i] = profileout->z[SURFELS - MASK_SIZE/2 -1];

}

\section{Nearest Point Logical Sensor Code}
\label{nearest}

/* nearestpoint.c
```

```
*  This is a logical sensor that  determines the
*  point in the range profile that represents the nearest
*  point to the sensor.
*
*  Uses the range courier and indirectly the range server.
*
*/

/** Logical Sensor Nearest_point **/

typedef struct NRST_PT_RQST
{
  struct  STD_RQST  STD_REQUEST;
  int_16            plot_data;

};

typedef struct NRST_PT_RPLY
{
  struct STD_RPLY  STD_REPLY;
  int_32          x,z;
};

Logical_Sensor Nearest_Point()
{

    int_32 i;
    uint_32  requestor, range_courier_id;

    struct  RANGE_RQST range_rqst;
    struct  RANGE_RPLY range_rply;

    struct  NRST_PT_RQST nearest_rqst;
    struct  NRST_PT_RPLY nearest_rply;

    /* The logical sensors do nothing when they are created,
    except create a courier for the device driver and wait for a request */

    range_courier_id = _Create(RANGE_COURRIER);

    /* only the skill that created this logical sensor can make requests */
    requestor = _Father_id();

    for(;;)
    {
      nearest_rqst.STD_REQUEST.MSG_SIZE = sizeof(struct NRST_PT_RQST);
      nearest_rply.STD_REPLY.MSG_SIZE = sizeof(struct NRST_PT_RPLY);

      /* Get a request from the skill
          .... logical sensors spend most time blocked here */

      _Receive( (char *) &nearest_rqst, requestor );

      range_rqst.STD_REQUEST.MSG_TYPE = GET_PROFILE;

      /* Get the range data from the range courier */

      _Receive( (char *) &range_rply, range_courier_id);
      range_rqst.STD_REQUEST.MSG_TYPE = GET_PROFILE;

      /* Unblock the courier */
      _Reply( (char *) &range_rqst, range_courier_id);
```

```
      /*  find the nearest point to the sensor */

      nearest_rply.z = 999999;
      nearest_rply.x = 0;

      for(i=1; i<SURFELS-5; i++)
        if((range_rply.profile.z[i] < nearest_rply.z) &&
            !range_rply.profile.invalid[i])
        {
          nearest_rply.z = (int_32) range_rply.profile.z[i];
          nearest_rply.x = (int_32) range_rply.profile.x[i];
        }

      /* sends back data to Extend for the Mac plot */
      if(nearest_rqst.plot_data == 1)
      {
        putprofile(&range_rply.profile);
      }

      /* reply to the requestor with the nearest point information */

      _Reply( (char *) &nearest_rply, requestor );

    }  /* for(;;) */
};
```

C.2 Range Finder Courier

```
/* range_courier.c
*  This is an intermediary between the logical sensors and the range server.
*  Its only function is to prevent blocking of the logical sensor while the
*  range data is being collected.  The range courier is not used when the
*  logical sensor is not time critical.  In those cases the logical sensor
*  interacts directly with the range server.
*/

task range_courier()
{
  struct UCB *range_server_ucb;

  struct  RANGE_RQST range_rqst;
  struct  RANGE_RPLY range_rply;


  /* open a connection to the range server  and intialize the scanner */

  range_server_ucb = _Open("RANGE_SERVER:",0);

  range_rqst.STD_REQUEST.MSG_SIZE = sizeof(struct RANGE_RQST);
  range_rply.STD_REPLY.MSG_SIZE = sizeof(struct RANGE_RPLY);
  range_rqst.STD_REQUEST.MSG_TYPE = INIT_SCANNER;

  _Send((char *)&range_rqst, (char *)&range_rply,
           range_server_ucb->UCB_MAIN.UCB_SERVER);

  /* assume first request is for a profile */

  range_rqst.STD_REQUEST.MSG_TYPE = GET_PROFILE;

  /* throw away the first one */

  _Send((char *)&range_rqst, (char *)&range_rply,
           range_server_ucb->UCB_MAIN.UCB_SERVER);


  for(;;)
  {
    /* send the request to the range server */

    range_rqst.STD_REQUEST.MSG_SIZE = sizeof(struct RANGE_RQST);
    range_rply.STD_REPLY.MSG_SIZE = sizeof(struct RANGE_RPLY);

    _Send((char *)&range_rqst, (char *)&range_rply,
              range_server_ucb->UCB_MAIN.UCB_SERVER);

    /* send the reply from the range server back to the logical sensor */

    range_rqst.STD_REQUEST.MSG_SIZE = sizeof(struct RANGE_RQST);
    range_rply.STD_REPLY.MSG_SIZE = sizeof(struct RANGE_RPLY);

    _Send((char *)&range_rply,  (char *)&range_rqst, _Father_id());
  }
}
```

Appendix D

Sensor Driver Code

D.1 Force-torque Sensor Driver

```
/*
*  force_torque.h
*  Structures for the collection and transmission of
*  force-torque information.
*
*/

struct FT
{
    char instr[80];
    int_16 overload, fx,fy,fz,mx,my,mz;
};


typedef struct FT_RQST
{
  struct STD_RQST    STD_REQUEST;
  int_16             FT_OPERATION;          /* The operation  */
  uint_32            FT_OP_PARAMETERS[10];  /* Op Parameters  */
};

typedef struct FT_RPLY
{
  struct STD_RPLY    STD_REPLY;
  struct FT          FT_DATA;
};

/* The possible requests that can be sent to the FT Server.
*     1,2,3 are defined in connect.h. */

/* #define OPEN_REQUEST          1  */
/* #define CLOSE_REQUEST         2  */
/* #define CLIENT_DIED           3  */

#define  CHANGE_COMM_VECTOR  4
#define  GET_FT_RECORD       5
#define  RESET_FT_BIAS       6
#define  UNBIAS              7
#define  ZERO_BIAS           8
#define  ONE_FT_RECORD       9
```

```
#define  ENABLE_FT_OUTPUT   10
#define  INHIBIT_FT_OUTPUT  11
#define  INITIALIZE_FT      12


/*
*  ft_server.c
*  Processor 3  (Test on P1)
*  This task receives ft_requests for control of the force-torque sensor,
*  or for force-torque data.  The data is collected and replied to the
*  client.  There is no queue, and no courier, the server simply calls
*  the appropriate routines to carry out the ft_request.
*
*/

task ft_server()
{

  struct  STD_RQST      std_rqst;
  struct  STD_RPLY      std_rply;
  struct  OPEN_RPLY      open_rply;

  struct  FT_RQST  ft_rqst;
  struct  FT_RPLY  ft_rply;

  uint_32 requestor;   /* the client's id */

  extern struct FT_INIT_REC ft_init;
  extern void Sercom_startup();

  /* Create, open and Select the SERCOM server */

  Sercom_startup();

  /*  Initialize the ft_server  */

  std_rqst.MSG_SIZE = sizeof( struct STD_RQST );
  std_rqst.MSG_TYPE = INITIALIZE_SERVER;

  ft_init.FT_HDR.STANDARD.MSG_SIZE = sizeof( struct FT_INIT_REC );

  _Send( (char *)&std_rqst, (char *)&ft_init, _Father_id() );

  if ( ft_init.FT_HDR.STANDARD.MSG_TYPE != FT_SERVER_INIT_REC )
      _Abort( "ft_server: bad init type. \n" );

  /* set up for all replies */
  ft_rply.STD_REPLY.MSG_SIZE = sizeof( struct FT_RPLY );
  std_rply.MSG_SIZE = sizeof( struct STD_RPLY );
  std_rply.RESULT = OK;

  _Report_for_service( ft_init.FT_NAME, REPORT_FOR_SERVICE );

  /* get requests from Clients */

  for(;;)
  {
    ft_rqst.STD_REQUEST.MSG_SIZE = sizeof( struct FT_RQST );
    requestor = _Receive( (char *)&ft_rqst, 0 );  /* from a client */
```

```
switch( ft_rqst.STD_REQUEST.MSG_TYPE )
{
  case CHANGE_COMM_VECTOR:
    ChangeCommVector(ft_rqst.FT_OP_PARAMETERS[0]);
    _Reply( (char *)&ft_rply, requestor);
  break;

   case GET_FT_RECORD:
      Get_FT_Record(&(ft_rply.FT_DATA));
      _Reply( (char *)&ft_rply, requestor);
  break;

  case RESET_FT_BIAS:
    Reset_FT_Bias();
    _Reply( (char *)&std_rply, requestor);
  break;

  case UNBIAS:
    UnbiasFT();
    _Reply( (char *)&std_rply, requestor);
  break;

  case ZERO_BIAS:
    ZeroBiasFT();
    _Reply( (char *)&std_rply, requestor);
  break;

   case ONE_FT_RECORD:
      OneFTrecord(&(ft_rply.FT_DATA));
      _Reply( (char *)&ft_rply, requestor);
  break;

  case ENABLE_FT_OUTPUT:
    Enable_FT_Output();
    _Reply( (char *)&std_rply, requestor);
  break;

  case INHIBIT_FT_OUTPUT:
    Inhibit_FT_Output();
    _Reply( (char *)&std_rply, requestor);
  break;

  case INITIALIZE_FT:
    Initialize_FT();
    _Reply( (char *)&std_rply, requestor);
  break;

  case OPEN_REQUEST:
  {

    open_rply.STD_REPLY.MSG_SIZE = sizeof( struct OPEN_RPLY );
    open_rply.STD_REPLY.RESULT = OK;
    open_rply.UCB_SERVER = _My_id();
    open_rply.UCB_CONNECTION = 1;  /* arbitrary */
    open_rply.UCB_FLAGS = 0;
    open_rply.SIZE_UCB_XTRA = 0;

    _Reply( (char *)&open_rply, requestor );

    /* since connection not recorded, don't care if client died */
    break;
  }
```

```
      case CLOSE_REQUEST:
      {
        /* for now, as connection not recorded,just reply */

        _Reply( (char *)&std_rply, requestor );

        break;
      }

      default:
      {
        _Abort( "FT_SERVER: BAD MSG_TYPE. \n" );
        break;
      }
    }  /* end switch */
    }  /* end forever */
};


/* force_commands.c
* Routines used to comunicate with the Force-Torque sensor
* These routines are specific to the ATI sensor.
*/

/****** ATI FT Communication ******/

/* Transmits a character string to the ForceTorque sensor */

Send_Force_Command (string)
char string[];
{
  int i;
  char ch;

  for (i = 0; string[i]; i++)
  {
    ch = string[i];
    _Put(ch);
    _Flush ();
    _Get();      /* get the echo for this char */
  }
  _Put('\n');    /* Carriage return */
  _Flush ();
  _Get();        /*    and get the echo */
}


/* Vector is a single hex value treated as a vector of bits */

ChangeCommVector( Vector )
uint_16 Vector;
{
  extern uint_16 CommVector;

  char OutString[10];

  _Sprintf(OutString,"CV %x",  Vector);

  CommVector  = Vector;  /* Save as a global to this server */
  Send_Force_Command(OutString);
```

```
  wait_for('>');
}

wait_for (ch)    /* waits until ch is found  */
char ch;
{
  int i;
  char waiting;

  for (i = 0; ( waiting = _Get() ) != ch ; );
}

/* Inputs one binary record acording to the comm vector */

Get_FT_Record(fm)
struct FT *fm;
{
  extern uint_16 CommVector;

  union {
    char mybytes[2];
    int_16 theword;
  }highlow;


  fm->overload =  _Get();

  if( CommVector & 0x01){
    highlow.mybytes[0] =  _Get();
    highlow.mybytes[1] =  _Get();
    fm->fx = highlow.theword;
  }
  if( CommVector & 0x02){
    highlow.mybytes[0] =  _Get();
    highlow.mybytes[1] =  _Get();
    fm->fy = highlow.theword;
  }

  if( CommVector & 0x04){
    highlow.mybytes[0] =  _Get();
    highlow.mybytes[1] =  _Get();
    fm->fz = highlow.theword;
  }

  if( CommVector & 0x08){
    highlow.mybytes[0] = _Get();
    highlow.mybytes[1] =  _Get();
    fm->mx = highlow.theword;
  }

  if( CommVector & 0x10){
    highlow.mybytes[0] =  _Get();
    highlow.mybytes[1] =  _Get();
    fm->my= highlow.theword;
  }

  if( CommVector & 0x20){
    highlow.mybytes[0] =  _Get();
    highlow.mybytes[1] =  _Get();
    fm->mz = highlow.theword;
  }
}
```

```
/******  Bias control ******/

Reset_FT_Bias()
{
  Send_Force_Command("SB");
  wait_for('>');
}

UnbiasFT()
{
  Send_Force_Command("SU");
  wait_for('>');
}

ZeroBiasFT()
{
  Send_Force_Command("SZ");
  wait_for('>');
}

/******       Query data Requests      *****/

OneFTrecord(fm)
struct FT *fm;
{
  /* Send_Force_Command("QR"); */
  _Put('\x14');  /* Control T */
  _Flush();
  Get_FT_Record(fm);
}

Enable_FT_Output()
{
  Send_Force_Command("QS");
}


Inhibit_FT_Output()
{
  Send_Force_Command(" "); /* just send a <cr> */
  wait_for('>');
}

/******  Initialize the FT sensor resets the Bias   ******/

Initialize_FT()
{
  /* wakeup! */
  _Put('\x17');  /* control w  warm boot*/
  _Flush();
  wait_for('>');
  Send_Force_Command("CD B");
  wait_for('>');

  Reset_FT_Bias();
}
```

D.2 Range Finder Sensor Driver

```
/*
*  range.h
*  Structures and definitions for handling range data.
*
*/

#define  ABS(x)      ((x>0) ? x : -x)
#define  MAX(x,y)    ((x) > (y) ? (x) : (y))

#define  P_ERROR    0.1

#define  SURFELS    255
#define  CONVERT    (180.0 * (7.0/22.0))


typedef struct PROFILE
{
  int_32    x[SURFELS],z[SURFELS];
  char      invalid[SURFELS];
};

typedef struct RANGE_RQST
{
  struct  STD_RQST  STD_REQUEST;
  int_16            plot_data;
};

typedef struct RANGE_RPLY
{
  struct STD_RPLY    STD_REPLY;
  struct  PROFILE    profile;
};

/* The possible requests that can be sent to the
*     Range Server.  1,2,3 are defined in connect.h. */

/* #define OPEN_REQUEST          1  */
/* #define CLOSE_REQUEST         2  */
/* #define CLIENT_DIED           3  */

#define  INIT_SCANNER  4
#define  GET_PROFILE  5

/*
*  range_server.c
*  Processor 1
*  This task receives range_requests for control of the range finder.
*  The data is collected and replied to the client.
*  There is no queue, and no courier, the server simply
*  calls the appropriate routines to carry out the range_request.
*
*/

task range_server()
{

  struct  STD_RQST    std_rqst;
  struct  STD_RPLY    std_rply;
  struct  OPEN_RPLY   open_rply;
```

```
struct  RANGE_RQST  range_rqst;
struct  RANGE_RPLY  range_rply;

uint_32 requestor;   /* the client's id */

extern struct RANGE_INIT_REC range_init;
extern uint_32 init_scanner();
extern void Get_profile();

/*  Initialize the range_server  */

std_rqst.MSG_SIZE = sizeof( struct STD_RQST );
std_rqst.MSG_TYPE = INITIALIZE_SERVER;

range_init.RANGE_HDR.STANDARD.MSG_SIZE = sizeof( struct RANGE_INIT_REC );

_Send( (char *)&std_rqst, (char *)&range_init, _Father_id() );

if ( range_init.RANGE_HDR.STANDARD.MSG_TYPE != RANGE_SERVER_INIT_REC )
    _Abort( "range_server: bad init type. \n" );

/* set up for all replies */
range_rply.STD_REPLY.MSG_SIZE = sizeof( struct RANGE_RPLY );
std_rply.MSG_SIZE = sizeof( struct STD_RPLY );
std_rply.RESULT = OK;

_Report_for_service(range_init.RANGE_NAME, REPORT_FOR_SERVICE );

/* get requests from Clients */

for(;;)
{
  range_rqst.STD_REQUEST.MSG_SIZE = sizeof( struct RANGE_RQST );
  requestor = _Receive( (char *)&range_rqst, 0 );  /* from a client */


  switch( range_rqst.STD_REQUEST.MSG_TYPE )
  {
    case GET_PROFILE:
      get_profile(&range_rply.profile);
      calcx(&range_rply.profile);
      validate(&range_rply.profile);
      _Reply( (char *)&range_rply, requestor);
    break;

    case INIT_SCANNER:
      init_scanner();
      _Reply( (char *)&std_rply, requestor);
    break;

    case OPEN_REQUEST:
    {

      open_rply.STD_REPLY.MSG_SIZE = sizeof( struct OPEN_RPLY );
      open_rply.STD_REPLY.RESULT = OK;
      open_rply.UCB_SERVER = _My_id();
      open_rply.UCB_CONNECTION = 1;  /* arbitrary */
      open_rply.UCB_FLAGS = 0;
      open_rply.SIZE_UCB_XTRA = 0;

      _Reply( (char *)&open_rply, requestor );
```

```
        /* since connection not recorded, don't care if client died */
        break;
      }

      case CLOSE_REQUEST:
      {
        /* for now, as connection not recorded,just reply */

        _Reply( (char *)&std_rply, requestor );

        break;
      }

      default:
      {
        _Abort( "RANGE_SERVER: BAD MSG_TYPE. \n" );
        break;
      }

    }  /* end switch */
  }  /* end forever */
};
```

Appendix E

Machine Drivers Code

E.1 PUMA-Val Machine Driver

```
/* val.h
* The following definitions are for the val2 machine driver
* server.  This must be included with Skills that use the Val2 controller.
*
*/
extern uint_16 get_response();

typedef struct VAL_RQST
{
/* Use Message type to identify the desired op */
  struct STD_RQST   STD_REQUEST;

  /* Up to 3 string parameters */
  char              STR1[20],STR2[20],STR3[20];

  /* Up to 6 float parameters  */
  float             FL_PARM[6];
};

typedef struct VAL_RPLY
{
  struct STD_RPLY  STD_REPLY;
  uint_16          SUCCESS;          /*  TRUE=1, no VAL error, else 0 */
  char             ERROR_MSG[100];   /* Error message from VAL */
  float            RET_PARM[12];     /* Up to 12 floating values returned */
};


/* The VAL operations defined.  */

/* #define OPEN_REQUEST          1  */
/* #define CLOSE_REQUEST         2  */
/* #define CLIENT_DIED           3  */

#define DELAY          4
#define OPENI          5
#define CLOSEI         6
#define SPEED          7
#define SET_TRAN       8
```

```
#define DRIVE          9
#define DEPART        10
#define DEPARTS       11
#define APPROACH      12
#define APPROACHS     13
#define SHIFT         14
#define WHERE         15
#define MOVE_JIM      16
#define RELMOVE       17
#define MOVE_STRAIGHT 18
#define EXEC1         19
#define BASE          20
#define TOOL1         21
#define HERE          22
#define CHANGE_MODE   23
#define ABORT         24
#define PC_ABORT      25
#define CLEAR         26
#define FULLTRAN      27
#define POINT         28


/* valcommands.c
*
*  This is a rewrite of VALPAK 2 written by Ron Kutrz in 1984.
*  It is completely new, and written to be accessed through the
*  Val server.
*

*/

delay (time,success,message)  /* val delay command */
char message [];
uint_16 *success;
int_32 time;
{
  char command_str[80];

  _Sprintf (command_str,"DO DELAY %d", time);
  _Printf("%s\n",command_str);   /* transmit command string */
  _Flush ();

  *success = get_response(message);
}

openi(success,message)
char message [];
uint_16 *success;
{
  _Printf("%s\n","DO OPENI");
  _Flush ();
  *success = get_response(message);
}

closei(success,message)
char message [];
uint_16 *success;
{
  _Printf("%s\n","DO CLOSEI");
  _Flush ();
  *success = get_response(message);
}
```

```
speed(thespeed,success,message)
int_32 thespeed;
char message [];
uint_16 *success;
{
  char command_str[80];

  _Sprintf (command_str,"SPEED %d",thespeed);
  _Printf("%s\n",command_str);
  _Flush ();
  *success = get_response(message);
}

/* VAL create a transformation  */

fulltran(tran, x, y, z, o, a, t,success, message)
char message[];
uint_16 *success;
char tran[];
int_32 x, y, z, o, a, t;
{
  char command_str[80];

    _Sprintf (command_str,"DO SET %s=TRANS(%d,%d,%d,%d,%d,%d)",
                    tran, x, y, z, o, a, t);
  _Printf("%s\n",command_str);
  _Flush ();
  *success = get_response(message);
}

set_tran(loc1,loc2,success,message)
char loc1[], loc2[];
char message [];
uint_16 *success;
{
  char command_str[80];

  _Sprintf (command_str,"DO SET %s = %s",loc1,loc2);
  _Printf("%s\n",command_str);
  _Flush ();
  *success = get_response(message);
}

drive(joint, change, valspeed,success,message)
int_32 joint, change, valspeed;
char message [];
uint_16 *success;
{
  char command_str[80];

  _Sprintf (command_str,  "DO DRI %d, %d, %d", joint,   change, valspeed);
  _Printf("%s\n",command_str);
  _Flush ();
  *success = get_response(message);
}

departs(distance,success,message)
int_32 distance;
char message [];
uint_16 *success;
{
```

```
  char command_str[80];
  _Sprintf (command_str,  "DO DEPARTS %d", distance);
  _Printf("%s\n",command_str);
  _Flush ();
  *success = get_response(message);
}

depart(distance,success,message)
int_32 distance;
char message [];
uint_16 *success;
{
  char command_str[80];
  _Sprintf (command_str,  "DO DEPART %d", distance);
  _Printf("%s\n",command_str);
  _Flush ();
  *success = get_response(message);
}

approach(loc,distance,success,message)
char loc[];
int_32 distance;
char message [];
uint_16 *success;
{
  char command_str[80];
  _Sprintf (command_str,  "DO APPRO %s %d",loc, distance);
  _Printf("%s\n",command_str);
  _Flush ();
  *success = get_response(message);
}

approachs(loc,distance,success,message)
char loc[];
int_32 distance;
char message [];
uint_16 *success;
{
  char command_str[80];
  _Sprintf (command_str,  "DO APPROS %s %d",loc, distance);
  _Printf("%s\n",command_str);
  _Flush ();
  *success = get_response(message);
}

shift(loc,dx,dy,dz,success,message)
char loc[];
int_32 dx,dy,dz;
char message [];
uint_16 *success;
{
  char command_str[80];
  _Sprintf (command_str, "DO SET %s=SHIFT(%s BY %d, %d, %d)",
                                        loc,loc,dx,dy,dz);
  _Printf("%s\n",command_str);
  _Flush ();
  *success = get_response(message);
}

move_jim(loc,success,message)
char loc[];
char message [];
```

```
uint_16 *success;
{
  char command_str[80];
  _Sprintf (command_str, "DO MOVE %s", loc);
  _Printf("%s\n",command_str);
  _Flush ();
  *success = get_response(message);
}

relmove(loc,success,message)
char loc[];
char message [];
uint_16 *success;
{
  char command_str[80];
  _Sprintf (command_str, "DO MOVE HERE:%s", loc);
  _Printf("%s\n",command_str);
  _Flush ();
  *success = get_response(message);
}

move_straight(loc,success,message)
char loc[];
char message [];
uint_16 *success;
{
  char command_str[80];
  _Sprintf (command_str, "DO MOVES %s", loc);
  _Printf("%s\n",command_str);
  _Flush ();
  *success = get_response(message);
}

execute1(prog,count)
int_32 prog,count;
{
  char command_str[80];
  _Sprintf (command_str, "EXECUTE PROG%d,%d", prog,count);
  _Printf("%s\n",command_str);
  _Flush ();
}

clear()
{
  char message[100];
  get_response(message);
}

base(dx,dy,dz,zrot,success,message)
int_32 dx,dy,dz,zrot;
char message [];
uint_16 *success;
{
  char command_str[80];
  _Sprintf (command_str, "BASE %d, %d, %d, %d", dx, dy, dz, zrot);
  _Printf("%s\n",command_str);
  _Flush ();
  *success = get_response(message);
}

tool1(loc,success,message)
char loc[];
```

```
char message [];
uint_16 *success;
{
  char command_str[80];
  _Sprintf (command_str, "TOOL %s", loc);
  _Printf("%s\n", command_str);
  _Flush ();
  *success = get_response(message);
}

here(loc,success,message)
char loc[];
char message[];
uint_16 *success;
{
  char command_str[80];
  _Sprintf (command_str, "DO HERE %s", loc);
  _Printf("%s\n",command_str);
  _Flush ();
  *success = get_response(message);
}

change_mode(mode, modenum,success,message)
int_32 mode, modenum;
char message[];
uint_16 *success;
{
  char command_str[80];
  _Sprintf (command_str, "DO MODE%d = %d", mode, modenum);
  _Printf("%s\n",command_str);
  _Flush ();
  *success = get_response(message);
}

abort(success,message)
char message [];
uint_16 *success;
{
  _Printf("ABORT \n");
  _Flush ();
  *success = get_response(message);
}

pcabort(success,message)
char message [];
uint_16 *success;
{
  _Printf("PCABORT \n");
  _Flush ();
  *success = get_response(message);
}

where(thisposition,message)
float thisposition[];
char message [];
{
  char response[300];
  _Printf("WHERE\n");
  _Flush ();
  get_str(response);

  /*_Log_gossip(response); */
```

```
  _Sscanf(response,"%f %f %f %f %f %f %f %f %f %f %f %f",
                   &thisposition[0],&thisposition[1],
                   &thisposition[2],&thisposition[3],
                   &thisposition[4],&thisposition[5],
                   &thisposition[6],&thisposition[7],
                   &thisposition[8],&thisposition[9],
                   &thisposition[10],&thisposition[11]);
  message[0] = '\0';
}

/* get the coordinates of a named point */

point(loc, thisposition, message)
float thisposition[];
char loc[];
char message [];
{
  char response[300];
  char command_str[80];

  /* read the position information */
  _Sprintf (command_str, "POINT %s", loc);
  _Printf("%s\n",command_str);
  _Flush ();
  get_point_location(response);
  _Printf("\n");  /* a null response to the ? */
  _Flush ();
  get_response(message);  /* clear the . */

  /* _Log_gossip(response); */

  /* get the position info from the string */

  _Sscanf(response,"%f %f %f %f %f %f",
           &thisposition[0],&thisposition[1],
           &thisposition[2],&thisposition[3],
           &thisposition[4],&thisposition[5]);

}

get_point_location(response)
char response[];
{
  int i=0;
  char rch;

  while(rch != '\12')  /* skip the first line */
    rch = _Get();

  rch = _Get();
  while(rch != '\12')  /* skip the second line */
    rch = _Get();

  while(rch != '?')  /* collect chars until a ? appears */
  {
    rch = _Get();
    response[i++] = rch;
  }
  response[i] = '\0';
}
```

```
get_str (char_str)  /* used by the WHERE routine to get the response */
char char_str[];
{
  int i = 0, count = 0, space = 0;
  char rch;

  while (count < 5)  /* after the 5th LF terminate */
  {
    rch = _Get();
    if (rch == '\12')       /* if LF      */
    {
      space = 1;
      count++;              /* count line feed  */
      if (count == 4)
        char_str[i++] = ' ';
    }
    else
    {
      if (count == 2 || count == 4)
        if (rch == ' ' && space == 0)
        {
          char_str[i++] = rch;
          space++;
        }
        else if (rch != ' ')
        {
          space = 0;
          char_str[i++] = rch;
        }
    }
  }
  char_str[i] = '\0';

/*  wait for '.' response  */
  while ( ( rch = _Get() ) != '.' );    /*  do nothing  */
}

/* waits until . is found, all read in characters are put in string  */
uint_16 get_response(message)
char message[];
{
  uint_16 i,success;

  success = 1;  /* start by assuming success */

  message[0] = message[1] = '\0';

  for (i = 2; ; i++ )
  {
    message[i] = _Get();

    if(message[i] == '.')
      return(success);  /* may be an error message has been read */
    if(message[i] == '*')
      success = 0;      /* an error occured */

    /* skip line feeds and carriage returns */
    if((message[i] == '\n') || (message[i] == '\r')) i--;
  }
}
```

```
/*
*  valserver.c
*  Processor 1  (Test on P1)
*  This task receives val_requests for control of the PUMA 560.
*  The robot actions are carried out by using the EHVT server to
*  communicate with VAL2.  The error messages, or robot location
*  information are passed back to the client through the reply
*  messages.
*
*/

task valserver()
{
  /* EHVT  */

  struct  UCB *ucb_iv, *ucb_ov;
  extern  struct EHVT_PORT_INIT_REC  Ehvt1_init;

  /* Val server */
  struct  STD_RQST      std_rqst;
  struct  STD_RPLY      std_rply;
  struct  OPEN_RPLY      open_rply;

  struct  VAL_RQST  val_rqst;
  struct  VAL_RPLY  val_rply;

  uint_32 requestor;   /* the client's id */
  int_16 i;

  extern struct VAL_INIT_REC val_init;

  /* Create, open, and select the EHVT server for communication */
  _Server_create( EHVT, (struct INIT_REC *)&Ehvt1_init);

  ucb_iv = _Open( "EHVT: +r", 0 );
  ucb_ov = _Open( "EHVT: +w", 0);
   _Selectinput ( ucb_iv );
  _Selectoutput ( ucb_ov );

  /*  Initialize the valserver  */

  std_rqst.MSG_SIZE = sizeof( struct STD_RQST );
  std_rqst.MSG_TYPE = INITIALIZE_SERVER;


  val_init.VAL_HDR.STANDARD.MSG_SIZE = sizeof( struct VAL_INIT_REC );

  _Send( (char *)&std_rqst, (char *)&val_init, _Father_id() );

  if ( val_init.VAL_HDR.STANDARD.MSG_TYPE != VAL_SERVER_INIT_REC )
      _Abort( "val_server: bad init type. \n" );

  /* set this up for all replies */
  val_rply.STD_REPLY.MSG_SIZE = sizeof( struct VAL_RPLY );
  std_rply.MSG_SIZE = sizeof( struct STD_RPLY );
  std_rply.RESULT = OK;

  _Report_for_service( val_init.VAL_NAME, REPORT_FOR_SERVICE );

  /* get requests from Clients */

  for(;;)
```

```
{
  val_rqst.STD_REQUEST.MSG_SIZE = sizeof( struct VAL_RQST );
  requestor = _Receive( (char *)&val_rqst, 0 );  /* from a client */

  for(i=0; i<100; i++)
    val_rply.ERROR_MSG[i] = '\0';    /* clear old error messages */

  switch( val_rqst.STD_REQUEST.MSG_TYPE )
  {
    case DELAY:
      delay((int_32) val_rqst.FL_PARM[0],&val_rply.SUCCESS,
                                        val_rply.ERROR_MSG);
      _Reply( (char *)&val_rply, requestor);
    break;

    case OPENI:
      openi(&val_rply.SUCCESS, val_rply.ERROR_MSG);
      _Reply( (char *)&val_rply, requestor);
    break;

    case CLOSEI:
      closei(&val_rply.SUCCESS, val_rply.ERROR_MSG);
      _Reply( (char *)&val_rply, requestor);
    break;

    case SPEED:
      speed((int_32) val_rqst.FL_PARM[0],&val_rply.SUCCESS,
                                        val_rply.ERROR_MSG);
      _Reply( (char *)&val_rply, requestor);
    break;

    case SET_TRAN:
      set_tran(val_rqst.STR1,val_rqst.STR2,&val_rply.SUCCESS,
                                          val_rply.ERROR_MSG);
      _Reply( (char *)&val_rply, requestor);
    break;

    case DRIVE:
      drive((int_32) val_rqst.FL_PARM[0],(int_32) val_rqst.FL_PARM[1],
            (int_32) val_rqst.FL_PARM[2],
            &val_rply.SUCCESS, val_rply.ERROR_MSG);
      _Reply( (char *)&val_rply, requestor);
    break;

    case DEPART:
      depart((int_32) val_rqst.FL_PARM[0], &val_rply.SUCCESS,
                                        val_rply.ERROR_MSG);
      _Reply( (char *)&val_rply, requestor);
    break;

    case DEPARTS:
      departs((int_32) val_rqst.FL_PARM[0], &val_rply.SUCCESS,
                                          val_rply.ERROR_MSG);
      _Reply( (char *)&val_rply, requestor);
    break;

    case APPROACH:
      approach(val_rqst.STR1,(int_32) val_rqst.FL_PARM[0],
               &val_rply.SUCCESS, val_rply.ERROR_MSG);
      _Reply( (char *)&val_rply, requestor);
    break;
```

```
case APPROACHS:
  approachs(val_rqst.STR1,(int_32) val_rqst.FL_PARM[0],
            &val_rply.SUCCESS, val_rply.ERROR_MSG);
  _Reply( (char *)&val_rply, requestor);
break;

case SHIFT:
  shift(val_rqst.STR1,(int_32) val_rqst.FL_PARM[0],
        (int_32) val_rqst.FL_PARM[1], (int_32) val_rqst.FL_PARM[2],
        &val_rply.SUCCESS, val_rply.ERROR_MSG);
  _Reply( (char *)&val_rply, requestor);
break;

case WHERE:
  where(val_rply.RET_PARM,  val_rply.ERROR_MSG);
  val_rply.SUCCESS = 1;
  _Reply( (char *)&val_rply, requestor);
break;

case POINT:
  point(val_rqst.STR1, val_rply.RET_PARM,  val_rply.ERROR_MSG);
  val_rply.SUCCESS = 1;
  _Reply( (char *)&val_rply, requestor);
break;

case MOVE_JIM:
  move_jim(val_rqst.STR1, &val_rply.SUCCESS, val_rply.ERROR_MSG);
  _Reply( (char *)&val_rply, requestor);
break;

case RELMOVE:
  relmove(val_rqst.STR1, &val_rply.SUCCESS, val_rply.ERROR_MSG);
  _Reply( (char *)&val_rply, requestor);
break;

case MOVE_STRAIGHT:
  move_straight(val_rqst.STR1, &val_rply.SUCCESS, val_rply.ERROR_MSG);
  _Reply( (char *)&val_rply, requestor);
break;

case EXEC1:
  execute1((int_32) val_rqst.FL_PARM[0],
           (int_32) val_rqst.FL_PARM[1]);
  val_rply.SUCCESS = 1;
  _Reply( (char *)&val_rply, requestor);
break;

case BASE:
  base((int_32) val_rqst.FL_PARM[0], (int_32) val_rqst.FL_PARM[1],
       (int_32) val_rqst.FL_PARM[2], (int_32) val_rqst.FL_PARM[3],
       &val_rply.SUCCESS, val_rply.ERROR_MSG);
  _Reply( (char *)&val_rply, requestor);
break;

case TOOL1:
  tool1(val_rqst.STR1,&val_rply.SUCCESS, val_rply.ERROR_MSG);
  _Reply( (char *)&val_rply, requestor);
break;

case HERE:
  here(val_rqst.STR1,&val_rply.SUCCESS, val_rply.ERROR_MSG);
  _Reply( (char *)&val_rply, requestor);
```

```
break;

case CHANGE_MODE:
  change_mode((int_32) val_rqst.FL_PARM[0],
              (int_32) val_rqst.FL_PARM[1],
              &val_rply.SUCCESS, val_rply.ERROR_MSG);
  _Reply( (char *)&val_rply, requestor);
break;

case ABORT:
  abort((int_32) val_rqst.FL_PARM[0],(int_32) val_rqst.FL_PARM[1],
              &val_rply.SUCCESS, val_rply.ERROR_MSG);
  _Reply( (char *)&val_rply, requestor);
break;

case PC_ABORT:
  pcabort(&val_rply.SUCCESS, val_rply.ERROR_MSG);
  _Reply( (char *)&val_rply, requestor);
break;

case CLEAR:
  clear();
  _Reply( (char *)&val_rply, requestor);
break;

case FULLTRAN:
  fulltran(val_rqst.STR1,
      (int_32) val_rqst.FL_PARM[0],(int_32) val_rqst.FL_PARM[1],
      (int_32) val_rqst.FL_PARM[2],(int_32) val_rqst.FL_PARM[3],
      (int_32) val_rqst.FL_PARM[4],(int_32) val_rqst.FL_PARM[5],
      &val_rply.SUCCESS, val_rply.ERROR_MSG);
  _Reply( (char *)&val_rply, requestor);
break;


case OPEN_REQUEST:
{
  open_rply.STD_REPLY.MSG_SIZE = sizeof( struct OPEN_RPLY );
  open_rply.STD_REPLY.RESULT = OK;
  open_rply.UCB_SERVER = _My_id();
  open_rply.UCB_CONNECTION = 1;  /* arbitrary */
  open_rply.UCB_FLAGS = 0;
  open_rply.SIZE_UCB_XTRA = 0;

  _Reply( (char *)&open_rply, requestor );

  /* since connection not recorded, don't care if client died */
  break;
}

case CLOSE_REQUEST:
{
  /* for now, as connection not recorded,just reply */
  _Reply( (char *)&std_rply, requestor );
  break;
}

default:
{
  _Abort( "VAL_SERVER: BAD MSG_TYPE. \n" );
  break;
}
```

```
    }  /* end switch */

    }  /* end forever */
} /* end VAL server */
```

Appendix F

Skills Code

F.1 Rub Skill

```
/* rub.c
*    The rub skill pushes the end-effector in a direction specified in the
*    tool coordinate sys. The end-effector is in contact with the surface
*    using the min-force and max-force vectors,as in the press skill.
*    The rubbing will stop in the event of a stop force is encountered, a
*    specified distance has been traversed, or in the failure case, and
*    emergency force has beenencountered.
*
*    Parameters:
*      Dirtection to rub  int      degrees in the XY tool coordinate sys.
*      Distance to rub    float    cm to rub
*      Unit Speed         float    distance to move each set point
*      Max Z Force        int      upper bound of the pressure
*      Min Z Force        int      lower bound of the pressure
*      Stop Forces        int      abs value of forces to stop at when
*                                      encountered
*      Error Stop Forces  int      abs value of forces that mean something
*                                      has gone wrong
*
*  Acknowledgements:
*    0 - Distance completely moved
*    1 - Stop due to normal forces
*    2 - Stop due to error forces
*
*/

#include <Math.h>

Skill Rub(argc,argv)
int argc;
STR argv[];
{

  /* Connections */
  struct UCB *ft_server_ucb;
  struct UCB *alter_ucb;
  struct UCB *val_server_ucb;

  struct MOTN_RQST motn_rqst;
  struct MOTN_RPLY motn_rply;
```

```
struct FT_RQST ft_rqst;
struct FT_RPLY ft_rply;

/* Parameter variables */
int direction, distance, show_data;
float unit_speed, vertical_speed, max_z_force, min_z_force;
struct FT stop_forces, error_stop_forces;

/* local use */
float x_step, y_step;
boolean done = FALSE;
float distance_so_far = 0.0;

if(argc != 10)
{
  cerr("System: Incorrect number of arguments.");
  return;
}

direction = read_int(argv[1]);
distance = read_float(argv[2]);
unit_speed = read_float(argv[3]);
vertical_speed = read_float(argv[4]);
max_z_force = read_int(argv[5]);
min_z_force = read_int(argv[6]);
stop_forces.fx = read_int(argv[7]);
error_stop_forces.fx = read_int(argv[8]);
show_data = read_int(argv[9]);


stop_forces.fy = stop_forces.fx;
stop_forces.fz = stop_forces.fx;
error_stop_forces.fy = error_stop_forces.fx;
error_stop_forces.fz = error_stop_forces.fx;

/* verify some parameters as reasonable */
if((direction < 0 ) || (direction > 360))
{
  cerr("Direction is out of range.");
  return;
}
if((unit_speed < 0.0) || (unit_speed > 2.0) ||
   (vertical_speed <0.0) || (vertical_speed > 1.0))
{
  cerr("Vertical Speed must be < 2.0\nHorizontal < 1.0   ");
  return;
}

/* distance is translated from cm to mm */

distance *= 10.0;

/* calculate the step size for the X and Y directions
   given the direction and unit speed */

y_step = ((float) sin((extended) (DEG_TO_RAD * direction))) * unit_speed;
x_step = ((float) cos((extended) (DEG_TO_RAD * direction))) * unit_speed;

/* open the required servers */

/* open the force torque server */
if( ! (ft_server_ucb = _Open("FT_SERVER:",0)))
```

```
{
  cerr("System:  Open of FT_SERVER FAILED. ");
  _Abort( "  " );
  _Flush();
}

if( ! (val_server_ucb = _Open("VAL_SERVER:",0)))
{
  cerr("System: *** Open of VAL_SERVER FAILED.\n");
  _Abort( " " );
  _Flush();
}

if( ! (alter_ucb = _Open( "ALTER:", 0 )))
{
  cerr("System: *** Open of ALTER_SERVER FAILED.\n");
  _Abort( " " );
  _Flush();
}

/* get alter data flowing */
Start_alter( alter_ucb , val_server_ucb);

/******* Start up the force-torque sensor */
ft_rqst.STD_REQUEST.MSG_SIZE = sizeof(struct FT_RQST);
ft_rply.STD_REPLY.MSG_SIZE = sizeof(struct FT_RPLY);


/*Throw away the first ft reading */
ft_rqst.STD_REQUEST.MSG_TYPE = ONE_FT_RECORD;
_Send((char *)&ft_rqst, (char *)&ft_rply,
             ft_server_ucb->UCB_MAIN.UCB_SERVER);   /* get a record */

/* Set up a motion request to the Alter Machine controller */
motn_rqst.STD_REQUEST.MSG_TYPE = MOTION_RQST;
motn_rqst.EXECUTE = IMMEDIATE;
motn_rqst.MODE = MOVE;
motn_rqst.FRAME = TOOL;
motn_rqst.REPETITIONS = 1;
motn_rqst.EXCEPTION = 0;
motn_rqst.COLL_COUNT = 0;
motn_rqst.MOTION_BUFF[0] = x_step;
motn_rqst.MOTION_BUFF[1] = y_step;
motn_rqst.MOTION_BUFF[2] = vertical_speed;
motn_rqst.MOTION_BUFF[3] = 0.0;
motn_rqst.MOTION_BUFF[4] = 0.0;
motn_rqst.MOTION_BUFF[5] = 0.0;

motn_rqst.STD_REQUEST.MSG_SIZE = sizeof( motn_rqst );
motn_rply.MOTN_REQUEST.STD_REQUEST.MSG_SIZE = sizeof( motn_rply );

/* feedback loop */

while(!done)
{
  /* take a force reading */
  ft_rqst.STD_REQUEST.MSG_TYPE = ONE_FT_RECORD;
  _Send((char *)&ft_rqst, (char *)&ft_rply,
              ft_server_ucb->UCB_MAIN.UCB_SERVER);   /* get a record */

  /* check emergency stop */
  if(  (ABS(ft_rply.FT_DATA.fx) > ABS(error_stop_forces.fx))  ||
```

```
     (ABS(ft_rply.FT_DATA.fy) > ABS(error_stop_forces.fy))  ||
     (ABS(ft_rply.FT_DATA.fz) > ABS(error_stop_forces.fz)) )
{
  done = TRUE;
  _Close( ft_server_ucb  );
  _Close(alter_ucb);
  cleanup(val_server_ucb);
  _Close( val_server_ucb  );

  acknowledge(2);   /* stop due to error forces */
  return;
}

/* send the sensor data if it is requested */

if(show_data)
  put_ft(ft_rply.FT_DATA);

/* check normal stop forces*/
if(  (ABS(ft_rply.FT_DATA.fx) > ABS(stop_forces.fx))  ||
     (ABS(ft_rply.FT_DATA.fy) >  ABS(stop_forces.fy))  ||
     (ABS(ft_rply.FT_DATA.fz) >  ABS(stop_forces.fz)) )
{
  done = TRUE;
  _Close( ft_server_ucb  );
  _Close(alter_ucb);
  cleanup(val_server_ucb);
  _Close( val_server_ucb  );

  /* stop due to normal forces encountered */
  acknowledge(1);   /* acknowledge a value indicating
                       stop forces were encountered */
  return;
}

/* check Z forces */
if( (ft_rply.FT_DATA.fz  < min_z_force) &&
     (ft_rply.FT_DATA.fz  > max_z_force) )
{
  motn_rqst.MOTION_BUFF[2] = 0.0;  /* No Z motion */
}
else
{
  if(ft_rply.FT_DATA.fz  > min_z_force)
    motn_rqst.MOTION_BUFF[2] = vertical_speed;    /* approach */
  else
    motn_rqst.MOTION_BUFF[2] = -vertical_speed;  /* back off! */
}

/* Safety check for too large a move. Should never happen.  */
if((fabs((extended)motn_rqst.MOTION_BUFF[0])>fabs((extended)unit_speed)) ||
   (fabs((extended)motn_rqst.MOTION_BUFF[1])>fabs((extended)unit_speed)))
  _Abort("System: Serious error - Tried to move in too large a step! ");

/* make a move of x_step, y_step, and the calculated Z */
_Send( (char *)&motn_rqst, (char *)&motn_rply,
                    alter_ucb->UCB_MAIN.UCB_SERVER );

/* check if moved as far as desired. */
distance_so_far += unit_speed;
if(distance_so_far >= distance)
{
```

```
      done = TRUE;
      acknowledge(0);  /* default completion */

      /* Stop and close the Alter Machine Controller, the Val machine
         controller, and the FT sensor Driver */

      _Close( ft_server_ucb  );
      _Close(alter_ucb);
      cleanup(val_server_ucb);
      _Close( val_server_ucb  );

    }
  }
}
```

F.2 Follow Edge Skill

```
/* follow_edge.c
*  Follow edge moves the tool along an edge maintaining a constant position
*  and orientation with respect to the edge - in 5 degrees of freedom.
*  A distance to travel along the edge is specified, as well as the Alter
*  speed. It is assumed that the end-effector is in a position such that
*  the first edge found in the range profile that is more than 4 mm in
*  height is the edge to be followed.
*
*  The end-effector moves in a direction perpendicular to the range finder.
*  This is in tool X.  The size of each step depends on the translation
*  speed entered in the dialog box.
*
*  Parameters:
*    Distance to track along the edge
*    Tracking Speed  ( Alter Units )
*    Alter translation speed     (correction speed )
*    Alter rotation speed
*    Standoff distance from edge
*
*  Logical Sensor: Edge_point
*  Machine Driver: Val2
*  Machine Driver: Alter
*
*  Acknowledge:  Always acknowledges 0, unless a fatal error occurs.
*
*/


#include <Math.h>

Skill Follow_edge(argc,argv)
int argc;
STR argv[];
{
  uint_32 LS_Edge_point;  /* pointer to logical sensor */

  struct UCB *alter_ucb;
  struct UCB *val_server_ucb;
  struct UCB *Clock_ucb;

  struct  MOTN_RQST motn_rqst;
  struct  MOTN_RPLY motn_rply;
  struct  VAL_RQST val_rqst;
  struct  VAL_RPLY val_rply;
  struct  EDGE_RQST edge_rqst;
  struct  EDGE_RPLY edge_rply;

  /* parameter variables */
  float track_speed, alter_speed_translation, alter_speed_rotation;
  int distance, standoff, plot_data;

  /* local variables */
  int_32 i;
  float distance_so_far = 0.0;
  char tempstring[70];
  extended  target[6], corrections[6];  /* x, y, z, rx, ry, rz */
  extern void calc_correction(), align_to_seam();

  /* save the history of the MOVES for the past cm */
  extended history_buffer[100][3];
```

```
/* col 0 = previous moves in Y, col 1 = prev moves in Z */
int_16 hb_size, hb_index;

if(argc != 7)
{
  cerr("System: Incorrect number of arguments.");
  return;
}

distance = read_float(argv[1]);
track_speed = read_float(argv[2]);
alter_speed_translation = read_float(argv[3]);
alter_speed_rotation = read_float(argv[4]);
standoff = read_float(argv[5]);
plot_data = read_int(argv[6]);

if(track_speed > 3  ||  track_speed < .1){
  cerr("Track speed is in mm/28ms and cannot be larger than 3
        or smaller than 0.1\n");
  return;
}
if(alter_speed_translation > 3){
  cerr("Alter speed is in mm/28ms and cannot be larger than 3.\n");
  return;
}
if(alter_speed_rotation > 1){
  cerr("Alter speed rotation is in deg/28ms
        and cannot be larger than 1.\n");
  return;
}
if(standoff < 15 || standoff > 45){
  cerr("Standoff must be between 15cm and 45cm.\n");
  return;
}

/* establish the target
   x is the direction of the motion of the end effector*/

target[1] = 0.0;                  /*  stay in the middle of the scan */
target[2] = (extended) standoff * 10.0;  /*  all go to  mm           */
target[3] = 0.0;                  /*  all rotations are 0            */
target[4] = 0.0;
target[5] = 0.0;

distance *= 10;  /* to mm */

/* calculate the size of the history buffer */

/* the number of steps required to traverse 1 cm. */
hb_size = (int_16) ((10.0/track_speed) + .5);
hb_index = 0;

/* create logical sensor Edge_point */

edge_rqst.STD_REQUEST.MSG_SIZE = sizeof(struct EDGE_RQST);
edge_rply.STD_REPLY.MSG_SIZE = sizeof(struct EDGE_RPLY);
edge_rqst.plot_data = plot_data;

LS_Edge_point = _Create(EDGE_POINT);
```

```
/* Open connections to machine drivers, and clock server */

if( ! (val_server_ucb = _Open("VAL_SERVER:",0)))
{
  cerr("System: *** Open of VAL_SERVER FAILED.\n");
  _Abort( " " );
  _Flush();
}

if( ! (alter_ucb = _Open( "ALTER:", 0 )))
{
  cerr("System: *** Open of ALTER_SERVER FAILED.\n");
  _Abort( " " );
  _Flush();
}

if( ! (Clock_ucb = _Open("CLOCK:",0)))
{
  cerr("System: *** Open of CLOCK Server FAILED.\n");
  _Abort( " " );
  _Flush();
}

/* create and invoke the tool transform
   - makes the tool the origin of the rangefinder*/

val_rqst.STD_REQUEST.MSG_SIZE = sizeof(struct VAL_RQST);
val_rply.STD_REPLY.MSG_SIZE = sizeof(struct VAL_RPLY);

val_rqst.STD_REQUEST.MSG_TYPE = FULLTRAN;
_Sprintf(val_rqst.STR1,"%s", "SEAM");
val_rqst.FL_PARM[0] =  49.0;
val_rqst.FL_PARM[1] =  0.0;
val_rqst.FL_PARM[2] =  197.0 + (standoff * 10);   /* val is in mm */
val_rqst.FL_PARM[3] =  90;
val_rqst.FL_PARM[4] = -90;
val_rqst.FL_PARM[5] =   0;

_Send((char *)&val_rqst, (char *)&val_rply,
              val_server_ucb->UCB_MAIN.UCB_SERVER);

if(!val_rply.SUCCESS)
  cerr(val_rply.ERROR_MSG);  /* make sure the fulltran command worked */

val_rqst.STD_REQUEST.MSG_TYPE = TOOL1;
_Sprintf(val_rqst.STR1,"%s", "SEAM");

val_rqst.STD_REQUEST.MSG_SIZE = sizeof(struct VAL_RQST);
val_rply.STD_REPLY.MSG_SIZE = sizeof(struct VAL_RPLY);

_Send((char *)&val_rqst, (char *)&val_rply,
              val_server_ucb->UCB_MAIN.UCB_SERVER);

if(!val_rply.SUCCESS)
  cerr(val_rply.ERROR_MSG);  /* make sure the tool command worked */


/* get alter data flowing */
Start_alter( alter_ucb , val_server_ucb);

/* Set up a motion request to the Alter Machine controller */
motn_rqst.STD_REQUEST.MSG_TYPE = MOTION_RQST;
```

```
motn_rqst.EXECUTE = IMMEDIATE;
motn_rqst.MODE = MOVE;
motn_rqst.FRAME = TOOL;
motn_rqst.REPETITIONS = 1;
motn_rqst.EXCEPTION = 0;
motn_rqst.COLL_COUNT = 0;
motn_rqst.MOTION_BUFF[0] = 0.0;
motn_rqst.MOTION_BUFF[1] = 0.0;
motn_rqst.MOTION_BUFF[2] = 0.0;
motn_rqst.MOTION_BUFF[3] = 0.0;
motn_rqst.MOTION_BUFF[4] = 0.0;
motn_rqst.MOTION_BUFF[5] = 0.0;

motn_rqst.STD_REQUEST.MSG_SIZE = sizeof( motn_rqst );
motn_rply.MOTN_REQUEST.STD_REQUEST.MSG_SIZE = sizeof( motn_rply );

/* throw away the first range data which is stale */
_Send((char *) &edge_rqst, (char *) &edge_rply,  LS_Edge_point);

/* empty the history buffer */
for(i=0; i<hb_size; i++)
  history_buffer[i][0] =
  history_buffer[i][1] =
  history_buffer[i][2] = 0.0;

motn_rqst.MOTION_BUFF[0] = track_speed;

/* it is necessary to align the end effector on the target seam in y,
     z and slope, before the angular corrections can be calculated */

align_to_seam(target, alter_ucb, LS_Edge_point, alter_speed_translation,
              alter_speed_rotation);

while(distance_so_far < distance)  /* do until distance is traversed */
{
  /* calculate the corrections to take */

  /* find the edge point */
  _Send((char *) &edge_rqst, (char *) &edge_rply,  LS_Edge_point);

  /* find the corrections based on the history buffer
     and the current range data */

  calc_correction(edge_rply.x, edge_rply.z,edge_rply.slope,
                  target,alter_speed_translation,
                  alter_speed_rotation, history_buffer,
                  hb_size, &hb_index, corrections);

  /* make the motion  via alter */

  for(i=1; i<6; i++)
    motn_rqst.MOTION_BUFF[i] = corrections[i];

  _Send( (char *)&motn_rqst, (char *)&motn_rply,
                     alter_ucb->UCB_MAIN.UCB_SERVER );
  _Send( (char *)&motn_rqst, (char *)&motn_rply,
                     alter_ucb->UCB_MAIN.UCB_SERVER );

  /* consider distance so far to be the total moves in the X direction */
  distance_so_far += track_speed;

  /* calculate how many alter motions can be made 2 or 3
```

```
    _Settime(Clock_ucb,0);
    elapsed_time = _Gettime(Clock_ucb);    */
  }

  /* Stop and close the Alter Machine Controller, the Val
     machine driver, and the FT sensor Driver */

  acknowledge(0);   /* tell user interface we're done */

  _Close(alter_ucb);
  cleanup(val_server_ucb);
  _Close( val_server_ucb  );
  _Close( Clock_ucb  );
}

/* given the detected edge point and the target, determine the corrections
   that should be made to adjust the position of the end-effector to be in
   the target location */

void calc_correction(y, z, slope, target,
                     alter_speed_translation,
                     alter_speed_rotation,
                     history_buffer, hb_size,
                     hb_index, corrections)

int_32 y,z;   /* location of the edge point */
extended target[], corrections[];
float slope, alter_speed_translation, alter_speed_rotation;
extended history_buffer[][3];
int_16 hb_size, *hb_index;
{

  extern extended past_corrections();
  extended deltay, deltaz, averaged_slope;

  y -=  10000;  /* shift to origin of the range finder */

  /* switch to mm  form .1 mm */
  corrections[1] = target[1] - (extended) y  / 10.0;
  corrections[2] = (extended) z / 10.0 - target[2] ;

  /* if it wants to move too fast */
  if(ABS(corrections[1]) > alter_speed_translation)
    corrections[1] = SIGNOF( corrections[1] ) *
                     (extended) alter_speed_translation;

  /* if it wants to move too fast */
  if(ABS(corrections[2]) > alter_speed_translation)
    corrections[2] = SIGNOF( corrections[2] ) *
                     (extended) alter_speed_translation;

  /* store the moves in y and z */
  history_buffer[*hb_index][0] = corrections[1];
  history_buffer[*hb_index][1] = corrections[2];

  /* calculate corrections around the Z axis */

  deltay = past_corrections(0, history_buffer, hb_size,*hb_index);

/* dist in y / dist in x */
  corrections[5] = atan( deltay / 10.0 ) * RAD_TO_DEG;
```

```
  /* if it wants to move too fast */
  if(ABS(corrections[5]) > alter_speed_rotation)
    corrections[5] = SIGNOF( corrections[5] ) *  alter_speed_rotation;

  /* calculate corrections around the Y axis   */

  deltaz = past_corrections(1, history_buffer, hb_size,*hb_index);
  corrections[4] = -(atan(  deltaz / 10.0 ))* RAD_TO_DEG;

  /* if it wants to move too fast */

  if(ABS(corrections[4]) > (alter_speed_rotation))
    corrections[4] = SIGNOF( corrections[4] ) * alter_speed_rotation;

  /* calculate corrections around the X axis */

  corrections[3] = (extended) -slope;

  /* if it wants to move too fast */
  if(ABS(corrections[3]) > alter_speed_rotation)
    corrections[3] = SIGNOF( corrections[3] ) * alter_speed_rotation;

  /* keep the resulting correction in the history buffer */
  history_buffer[*hb_index][2] = corrections[3];

  (*hb_index)++;
  if(*hb_index == hb_size)
    *hb_index = 0;

}

/* return the total of the past corrections over the last cm in
*  either Y or Z or Slope (Rx).  This function does not change any
*  of the parameters.

*  The history buffer is a ring buffer that keeps the most recent
*  corrections which occured over the past 1 cm of tracking.
*/

extended past_corrections(y_or_z_or_slope, history_buffer, hb_size,hb_index)
int_16 y_or_z_or_slope, hb_size, hb_index;
extended history_buffer[][3];
{

  int_16 additions_so_far;
  extended total;

  additions_so_far = 0;
  total = 0.0;

  while(additions_so_far <= hb_size)
  {
    total += history_buffer[hb_index][y_or_z_or_slope];
    hb_index--;
    additions_so_far++;
    if(hb_index == -1)
      hb_index = hb_size-1;
  }

  return(total);
}
```

Bibliography

[1] K. S. Fu, R. C. Gonzalez, and C. S. G. Lee. *Robotics: Control, Sensing, Vision, and Intelligence.* McGraw Hill Book Co., New York, 1987.

[2] U. Rembold and K. Horman. *Languages for Sensor-based Control in Robotics, preface.* NATO ASI series. Springer-Verlag, Berlin, 1986.

[3] G. D. Hager. *Task-directed Sensor Fusion and Planning - A Computational Approach.* Kluwer Academic Publishers, 1990.

[4] E. Mazer, J.Pertin-Troccaz, J.-M. Lefevre, B. Faverjon, A. Ijel, C. Billier, B. Ferrari, M. Barret, and P. Sellers. ACT: A robot programming environment. In *Proc. of IEEE International Conf. on Robotics and Automation*, pages 1427–1432, Sacramento, CA, April 1991.

[5] G. S. Guler. Task level robot programming. In *Proc. of 3rd Int. Symp. on Computer and Information Sciences*, pages 621–630, Commack, NY, November 1988. Nova Science Publishers.

[6] R. W. Harrigan. The role of model-based control in robotics. *J. Robotics and Automation*, 5(1):11–15, 1990.

[7] Unimation Inc. *Users Guide to VAL II*, ver. 1.1, 398T1 edition, 1984.

[8] R. H. Taylor, P. D. Summers, and J. M. Meyer. AML: A manufacturing language. *Int'l J. Robotics Res.*, 1(3):19–41, 1983.

[9] R. A. Volz. Report of the robot programming language working group: NATO workshop on robot programming languages. *IEEE Journal of Robotics and Automation*, 4(1):86–90, February 1988.

[10] S. Bonner and K. G. Shin. A comparative study of robot languages. *IEEE Computer*, 15:82–96, December 1982.

[11] W. I. Gruver, B. I Soroka, J. J. Craig, and T.L. Turner. Evaluation of commercially available robot programming languages. In *Proc. of 13th International Sym. on Industrial Robots*, pages 12.58–12.68, Chicago, IL, April 1983.

[12] T. Lozano-Perez. Robot programming. *Proceedings of the IEEE*, 71(7):821–841, July 1983.

[13] Anonymous. 1990 census of robotic installations. Canadian Machinery and Metalworking Magazine, June 1990. pp. 132-136.

[14] R. J. Popplestone, A. P. Ambler, and Bellos. RAPT: A language for describing assemblies. In *Industrial Robot*, pages 131–137, September 1978.

[15] L. I. Lieberman and M. A. Wesley. AUTOPASS: An automatic programming system for computer controlled mechanical assembly. *IBM J. of Research and Development*, pages 131–137, July 1977.

[16] S. K. Sorenson. *An Offline Approach to Task Level State Driven Robot Programming.* Ph. D. thesis, Brigham Young University, 1989.

[17] W. M. Gentleman, C. Archibald, S. Elgazzar, D. Green, and R. Liscano. Case studies of realtime multiprocessors in robotics. In *Proceedings of Second International Specialist Seminar on the Design and Applications of Paralled Digital Processors*, pages 1–5, Lisbon, Portugal, April 1991.

[18] S. Elgazzar, D. Green, and M. Gentleman. Open system architecture of a multiprocessor robot controller. In *Proc. of 7th Canadian CAD/CAM and Robotics Conference*, pages 6.63–6.69, Toronto, Ont., June 1988.

[19] V. Hayward, L. K. Daneshmend, and S. Hayati. An overview of Kali: A system to program and control cooperative manipulators. In *Proc. of 4th International Conf. on Advance Robotics ICAR*, Columbus, OH, June 1989.

[20] E. Sollbach and A. Goldenberg. Real-time control of robots: Strategies for hardware and software development. *J. of Robotics & Computer-Integrated Manufacturing*, 6(4):323–329, 1989.

[21] S. Elgazzar and A. Castonguay. Maestro: An open system architecture for multirobot control. In *Proc. of 3rd Conf. on Military Robotics Applications*, pages 99–106, Medecine Hat, Alberta, 1991.

[22] W. Zhang. A software system for robot programming. In T. Kanade, F. C. A. Groen, and L. O. Hertzberger, editors, *Proc. of Intelligent Autonomous Systems II*, pages 219–229, Amsterdam, The Netherlands, December 1989.

[23] C. A. Malcolm and A. P. Fothergill. Some architectural implications of the use of sensors. In *Languages for Sensor Based Control*, NATO ASI, pages 102–122. Springer Verlag, Berlin, 1986.

[24] J. S. Albus, R. Lumia, and H. G. McCain. NASA/NBS standard reference model for telerobot control system architecture (NASREM). NBS Technical Note 1235, National Bureau of Standards, 1986.

[25] R. Lumia, J. Fiala, and A. Wavering. The NASREM robot control system standard. *Robotics and Computer Integrated Manufacturing*, 6(4):303–308, 1989.

[26] Anonymous. *Visualization Workbench Sales Brochures*. Paragon Imaging Inc., Lowell, MA, 1992.

[27] J. Kodosky and B. Dye. *Labview Product Description*. National Instruments, Austin, TX, 1987.

[28] J. Kodosky. Graphical system for modeling a process and associated method. United States Patent 4,914,568, April 1990.

[29] R. Maglica and N. Martensson. Visual off-line programming of industrial robots. In *Proc. Of 24th International Symposium on Industrial Robots (ISIR)*, pages 369–375, Tokyo, Japan, November 1993.

[30] A. Jain and M. Donath. Knowledge representation system for robot-based automated assembly. *J. of Dynamic Systems, Measurement and Control*, III:463–469, September 1989.

[31] B. Frommherz and G. Werling. A graphical implicit programming system for robot action planning. In *Proc. of Conf. on Intelligent Autonomous Systems II*, pages 196–207, Amsterdam, The Netherlands, December 1989.

[32] A. C. Kak, A. J. Vayda, R. L. Cromwell, W. Y. Kim, and C. H. Chen. Knowledge based robotics. *Int'l J. Production Research*, 26(5):707–734, 1988.

[33] R. Brooks. Elephants can't play chess. *J. Robotics and Autonomous Systems*, 6((1,2)):3–15, June 1990.

[34] C. Archibald and M. van de Panne. Tracking and grasping moving objects using reflex behaviour. In *Proc. of 5th Int'l Conf. on Advanced Robotics (ICAR)*, pages 643–648, Pisa, Italy, June 1991.

[35] R. A. Brooks. A robust layered control systems for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2(1):14–23, March 1986.

[36] D. O'Hara and R. Kurtz. Prismatic shaped block insertion: An application of a multiprocessor robot controller using harmony. Erb-1006, National Research Council of Canada, Ottawa, July 1987.

[37] S. Venkatesan and C. Archibald. Realtime tracking in five degrees of freedom using two wrist-mounted laser range finders. In *Proc. of IEEE Int'l Conf. on Robotics and Automation*, pages 2004–2010, Cincinnati, OH, May 1990.

[38] C. Merritt, C. Archibald, and T. Ng. Pose determination of a satellite graple fixture using a wrist-mounted laser range finder. In *SPIE Symp. on Advances in Intelligent Robotic Systems*, pages 583–590, November 1988.

[39] G. Roth, D. O'Hara, and M. D. Levine. A holdsite method for parts acquisition using a laser rangefinder mounted on a robot wrist. *J. Robotic Systems*, 6(5):573–599, 1989.

[40] T. N. Mudge, R. A. Volz, and D. E. Atkins. Hardware/software transparency in robotics through oblect-level design. In *Proc. of SPIE Technical Symposium West*, volume SPIE 360, pages 216–223, 1982.

[41] G. R. Meijer, G. A. Weller, F. C. A. Groen, and L. O. Hertzberger. Sensor based control for autonomous robots. In *Proc. of IEEE International Conf. on Control and Applications*, pages WP–3–4, 1–5, Jerusalem, April 1989.

[42] Y. Roth and R. Jain. Integrated architecture for autonomous systems. In *Proc. of International Symposium on Intelligent Robotics*, volume 1571, pages 628–639, Bangalore, India, January 1991. SPIE.

[43] T. Ogasawara, K. Kitagaki, T. Suehiro, T. Hasegawa, and K. Takase. Model based implementation of a manipulation system with artificial skills. In *Proc. of 2nd International Symposium on Experimental Robotics*, Toulouse, France, June 1991.

[44] N. Burtnyk and J. Basran. Supervisory control of telerobots in unstructured environments. In *Proc. of 5th International Conf. on Advance Robotics*, pages 1420–1424, Pisa, Italy, June 1991.

[45] F. Glantschnig. System integration in a world of vendor specific solutions and defacto standards. In *Proc. of Symposium on Manufacturing Application Programming Language Environments (MAPLE '93*, pages 73–89, Ottawa, Canada, October 1993.

[46] M. C. Sturzenbecker. A distributed C++ environment for automated manufacturing. In *Proc. of MAPLE '90*, pages 111–123, Ottawa, May 1990.

[47] P. Zave. A compositional approach to multiparadigm programming. *IEEE Software*, pages 15–25, September 1989.

[48] H. Inoue. Three approaches for robotic system integration. In *Proc. of the International Symposium and Exposition on Robots, the 19th ISIR*, pages 94–105, Sydney, Australia, November 1988.

[49] T. Henderson and E. Shilcrat. Logical sensor systems. *J. of Robotic Systems*, 1(2):169–193, 1984.

[50] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. STATEMATE: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, 1990.

[51] D. Gelernter and N. Carriero. Coordination languages and their significance. *Comm. ACM*, 35(2):97–107, 1992.

[52] C. Archibald, M. Krieger, and E. Petriu. Software design of sensor-based robot skills. In *Proceedings of IEEE Instrumentation/Measurement Technology Conference*, pages 175–178, Hamamatsu, Shizuoka, Japan, May 1994.

[53] M. Krieger. Multiactivity paradigm for the coordination of flexible manufacturing systems (FMSs). *J. of Computer Integrated Manufacturing Systems*, 6(3):195–203, 1993.

[54] M. Krieger and R. Joannis. Process activity diagrams: A tool for the specification of and design of multiple microprocessor systems. In *Proc. of the ISMM Int'l Symp. on Software and Hardware Applications of Microcomputers*, pages 157–161, Beverly Hills, CA, 1986.

[55] M. Gini and G. Gini. Programming for intelligent robot systems. In S. G. Tzafestas, editor, *Intelligent Robotic Systems*, chapter 6, pages 165–208. Marcel Dekker, Inc, New York, NY, 1991.

[56] I. J. Cox and N. H. Gehani. Concurrent C and robotics. In *Proceedings of IEEE Conference on Robtoics and Automation*, pages 1463–1468, 1987.

[57] V. Hayward and R. P. Paul. Robotic manipulator control under Unix RCCL:A robot control 'C' library. *International Journal of Robotics Research*, 5:94–111, 1983.

[58] S. Leake, T. Green, S. Cofer, and T. Sauerwein. Hierarchical Ada robot rprogramming system (HARPS): A complete and working telerobot control system based on the NASREM model. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1022–1028, 1989.

[59] M. Lloyd. Graphical function chart programming for programmable controllers. *J. Control Engineering*, 32(10):73–76, October 1985.

[60] P. Loborg and A. Torne. A hybrid language for the control of multimachine environments. Research Report ISSN-0281-4250, Linkoping University, Linkoping, Sweden, April 1991.

[61] D. B. Stewart, D. E. Schmitz, and P. K. Khosla. CHIMERA II: A real-time UNIX-compatible multiprocessor operating system for sensor-based control applications. Technical Report CMU-RI-AML-89-02, The Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, September 1989.

[62] R. SanMartin, M. Krieger, and F. Ferreira. A simulator for real-time embedded systems. In *Proc. of Int'l Conf. on Simulation in Engineering Education*, pages 123 - 129, La Jolla, CA, January 1993.

[63] D. J. Mayhew. *Principles and Guildelines in software user interface design.* P T R Prentice Hall, Englewood Cliffs, NJ, 1992.

[64] J. D. Gould and C. Lewis. Designing for usability: Key principles and what designers think. *Communications of the ACM*, 28(3):300–311, March 1985.

[65] J.M. Spool. Product usability: Survival techniques. Seminar notes, User Interface Engineering Inc., Andover, MA, 1994.

[66] L. Leifer, M. Van der Loos, and D. Lees. Visual language programming: for robot command control in unstructured environments. In *Proc. of Fifth International COnference on Advanced Robotics (ICAR)*, pages 31–36, Pisa, Italy, June 1991.

[67] M. Gertz, D. B. Stewart, and P. K. Khosla. An iconic programming language for sensor-based robots. In *Proc. of Sixth Annual Workshop on Space Operations Applications and Research (Soar 1992)*, pages 1–8, Houston, TX, August 1992.

[68] D. E. Mahling and W. B. Croft. A visual language for the acquisition and display of plans. In *Proc. of 1989 IEEE Workshop on Visual Languages,*, pages 99–104, Rome, Italy, October 1989.

[69] G. E. Maier. Towards graphical programming in control of mechanical systems. In *Proc. of Symposium on Dynamics of Controlled Mechanical Systems IUTAM/IFAC*, pages 77–90, Zurich, Switzerland, May 1988.

[70] J. L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice Hall Inc., 1990.

[71] D. C. Petriu, C. Archibald, and E. Petriu. Petri net model of a resource management system for a multi-robot assembly cell. In *Proc. of IEEE SMC (Systems, Man, and Cybernetics) Conference*, pages 409–414, Le Touquet, France, October 1993.

[72] D. Schillinger and F. Kaufman. A high level language based on functional programming. In *Proc. of IEEE Int'l Conf. on Industrial Electronics, Control and Instrumentation*, pages 788–793, San Francisco, CA, 1985.

[73] B. H. Thomas and C. McLean. Using Grafcet to design generic controllers. In *Proc. of First Internationial Conference on Computer Integrated Manufacturing*, pages 110–119, Troy, NY, May 1988.

[74] P. T. Cox, F. R. Giles, and T. Pietrzykowski. Prograph: a step towards liberating programming from textual conditioning. In *Proc. of IEEE Workshop on Visual Languages*, pages 150–156, Rome, Italy, 1989.

[75] D. A. Norman. *The Psychology of Everyday Things.* Basic Books, Inc, New York, 1988.

[76] T. Winograd. *Language as a Cognitive Process*, volume 1: Syntax. Addison-Wesley Publishing Co., 1983.

[77] P. Hoffman. *Extend: Performance Modeling for Decision Support.* Imagine That, Inc., San Jose, CA, Version 2 edition, 1992.

[78] W. M. Gentleman. Using the Harmony Operating System. ERB-966, National Research Council of Canada, March 1987.

[79] W. M. Gentleman, M. Wein, S. A. MacKay, D. A. Stewart, R. Parr, and D. Green. Harmony: An operating system for embedded industrial multiprocessor applications. In *Proc. of IEEE Montech - COMPINT '87*, pages 249–252, Montreal, November 1987.

[80] Apple Computer Inc., Cupertino, CA. *Macintosh Programmer's Workshop 3.0 Reference*, 1988.

[81] W. M. Gentleman. Message passing between sequential processes: The reply primitive and the administrator concept. *Software Practice and Experience*, 11(5):435–466, May 1981.

[82] C. Archibald, W. M. Gentleman, and D. H. O'Hara. Realtime feedback control using a laser range finder and harmony. In *Proc. of 7th Canadian CAD/CAM and Robotics Conference*, pages 6:56–6:62, Toronto, Ont., June 1988.

[83] P. Dario, M. Bergamasco, and A. Fiorillo. *Force and Tactile Sensing for Robotics*, volume F43 of *NATO ASI Series F*, pages 154–185. Springer Verlag, 1986.

[84] Anonymous. *Installation and Operations Manual for F/T: Intelligent Multi-axis Force/Torque Sensor System.* Assurance Technologies, Inc., Garner, NC, 1991.

[85] M. Rioux, G. Bechthold, M. Duggan, and D. Taylor. Design of a large depth of view 3-D camera for robot vision. *Optical Engineering*, 26(12):1245–1250, December 1987.

[86] M. Rioux. Laser range finder based on a synchronized scanner. *Appl. Opt.*, 23(21):3837–3844, 1984.

[87] S. Amid, B. Trethewey, and C. Archibald. A System for Calibrating the Wrist-mounted Laser Range Finder. ERB-1012, National Rescearch Council of Canada, November 1988.

[88] M. T. Mason. Compliance and force control for computer controlled manipulators. *IEEE Transactions on Systems. Man, and Cybernetics*, SMC-11(6):418–432, June 1981.

[89] M. H. Raibert and J. J. Craig. Hybrid position/force control of manipulators. *Transactions of the ASME*, 102:126–133, June 1981.

[90] Committee Draft ISO/CD 10562.2. Manipulating industrial robots intermediate code for robots (ICR). Technical Report ISO/TC 184/SC 2 N212, International Standards Organization, Sept. 1991.

[91] A. Matsumoto and T. Arai. Japanese standard robot language SLIM and its educational system. In *Proc. of Symposium on Manufacturing Application Programming Language Environments (MAPLE) '93*, pages 117–126, Ottawa, Ont., October 1993.

[92] C. Archibald and E. Petriu. Getting friendly with robots. In *Proc. of 7th IFAC Symp. on Information Control Problems in Manufacturing Technology (INCOM)*, pages 343–348, Toronto, Canada, May 1992.

[93] C. Archibald and E. Petriu. Skills-oriented robot programming. In *Proc. of Intelligent Autonomous Systems III*, pages 104–113, Pittsburgh, PA, February 1993.

[94] C. Archibald and E. Petriu. Model for skills-oriented robot programming (SKORP). In *Proc. of SPIE Conf. on Applications of Artificial Intelligence XI: Machine Vision and Robotics*, pages 392–402, Orlando, FL, April 1993.

[95] C. Archibald and E. Petriu. Robotic skills development using a wrist-mounted laser range finder. In *Proc. of IEEE Instrumentation/Measurement Technology Conf.*, pages 448–452, Irvine, CA, May 1993.

[96] C. Archibald and E. Petriu. Abstractions to reduce complexity in sensor-based robot programming. In *Proc. of MAPLE '93, Symposium on Manufacturing Automation Programming Language Environments*, pages 103–116, Ottawa, Ontario, October 1993.

[97] C. Archibald and E. Petriu. Computational paradigm for creating and executing sensor-based robot skills. In *Proc. of International Symposium on Industrial Robotics*, pages 401–406, Tokyo, Japan, November 1993.

[98] C. Archibald and E. Petriu. Functional abstraction for high-level on-line robot programming. In *Proc of 1993 DND Workshop on Advanced Technologies in Knowledge Based Systems and Robotics*, pages 309–314, Ottawa, Ontario, November 1993.

[99] C. Archibald, E. Petriu, and A. Harb. Robot skills development using a laser range finder. *IEEE Transactions on Instrumentation and Measurement*, 43(2):265–271, April 1994.

[100] D. C. Petriu, E. M. Petriu, and C. C. Archibald. Communicating state-machine model of resource management in a multi-robot assembly cell. In *Proceedings of ISCA International Conference on Computer Applications in Industry and Engineering*, pages 363–367, Honolulu, Hawaii, December 1993.