A BEHAVIORAL APPROACH TO OPEN ROBOT SYSTEMS: DESIGN AND PROGRAMMING

By

YING ZHANG

B.Sc. Zhejiang University, China, 1984
M.Sc. Zhejiang University, China, 1987

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES (DEPARTMENT OF COMPUTER SCIENCE)

We accept this thesis as conforming to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

October 1989
©Ying Zhang, 1989
In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of COMPUTER SCIENCE

The University of British Columbia
Vancouver, Canada

Date Oct. 13, 1989
Abstract

Current robots have very limited abilities to adapt to the environment, to survive in unstructured situations, to deal with incomplete and inconsistent information, and to make real-time decisions. One reason for this is the limited power of computation; another reason is an incorrect decomposition of the robot architecture and the unsuitable representation with rigid programming style. The core of this thesis is a new design and programming methodology which is needed for more robust, flexible and intelligent robot systems, called open robot systems.

We have developed a general framework for open robot systems and a programming methodology for building open robot systems.

Our approach to robot design is: decompose a robot system into a set of hierarchical behavioral modules according to the logical or physical structure; each of the module is possibly further decomposed into a set of concurrent processes representing sensors, motors and their coordination according to task-achieving behaviors.

Our approach to robot programming is: build the behavioral modules based on parallel objects. Programming is a way of designing each individual module and constructing the modules into a complex system.

We have built a software system on a real robot arm to simulate the tasks for forest harvesting, based on our approach to robot system design and programming. The system is written in Parallel C++ in a transputer-based parallel environment.
Contents

Abstract ii

Contents iii

List of Figures v

Acknowledgement vii

1 Introduction 1

2 Related Work 6

3 A Behavioral Approach to Robot Design and Programming 12
   3.1 Building an Open Robot System via Behavioral Modules 13
   3.1.1 Behavioral Decomposition 14
   3.1.2 Structural Decomposition 16
   3.1.3 Two-dimensional Decomposition 19
   3.2 Building Behavioral Modules via Parallel Objects 21
      3.2.1 Characteristics of Parallel Distributed Object Oriented Languages 21
      3.2.2 Programming on Parallel Distributed Object Oriented Languages 23

4 Design of an Autonomous Robot for Forest Harvesting 28
   4.1 Behaviors in Forest Harvesting 29
   4.2 The Robot 30
   4.3 Robot Arm Excalibur and its Task Design 33
   4.4 Decomposition of Transputer-based Robot Systems 37
      4.4.1 The structural decomposition 38
      4.4.2 The interface of each module 41
      4.4.3 Constructing the System from Behavioral Modules 45
   4.5 An Alternative Configuration 48
5 Programming a Real-Time Multi-Sensory Robot System in Parallel C++

5.1 Brief Introduction to Parallel C++ ........................................... 52
5.2 Constructs in Parallel C++ ..................................................... 54
5.3 Module Design ................................................................. 60
  5.3.1 SERVO Module ............................................................ 61
  5.3.2 JOINT Module ............................................................. 61
  5.3.3 ARM Module ............................................................... 70
  5.3.4 ROBOT Module ............................................................ 73
5.4 Current Performance and Further Improvement ............................ 75

6 Conclusions and Further Research ............................................. 81
  6.1 Conclusions ..................................................................... 81
  6.2 Contributions .................................................................. 83
  6.3 Further Research ............................................................. 83

A FCP++ — An Object Oriented Flat Concurrent Prolog ................. 85
  A.1 Brief Introduction to FCP++ ............................................... 85
  A.2 FCP++ Program for Figure 3.4 ............................................ 87
  A.3 FCP++ Program for Figure 3.6 ............................................ 87

B Parallel C++ for AFSM ........................................................... 91

C Functional Joint Control .......................................................... 98
  C.1 Projection Technique for Joints Controlling Orientation .......... 98
  C.2 FJC for Joints Controlling Length ...................................... 100

Bibliography ................................................................. 101
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Structural Decomposition of the Robot System</td>
<td>17</td>
</tr>
<tr>
<td>3.2</td>
<td>System Decomposition</td>
<td>19</td>
</tr>
<tr>
<td>3.3</td>
<td>Traditional Approach to Programming of Using Sensors and Plans</td>
<td>24</td>
</tr>
<tr>
<td>3.4</td>
<td>The Correct Use of Sensor and Plan</td>
<td>25</td>
</tr>
<tr>
<td>3.5</td>
<td>Traditional Approach to Motion Planning</td>
<td>25</td>
</tr>
<tr>
<td>3.6</td>
<td>Motion Planning under Communicative Process Structure</td>
<td>26</td>
</tr>
<tr>
<td>4.1</td>
<td>Functions of Excalibur (copyed from Excalibur manual)</td>
<td>33</td>
</tr>
<tr>
<td>4.2</td>
<td>Structural Decomposition of Our Robot System</td>
<td>38</td>
</tr>
<tr>
<td>4.3</td>
<td>Current Configuration of Transputer and Controller</td>
<td>40</td>
</tr>
<tr>
<td>4.4</td>
<td>Transputer-based Robot System</td>
<td>41</td>
</tr>
<tr>
<td>4.5</td>
<td>Interface of ROBOT Module</td>
<td>43</td>
</tr>
<tr>
<td>4.6</td>
<td>Interface of ARM</td>
<td>44</td>
</tr>
<tr>
<td>4.7</td>
<td>Interface of JOINT module</td>
<td>44</td>
</tr>
<tr>
<td>4.8</td>
<td>Interface of SERVO Module</td>
<td>45</td>
</tr>
<tr>
<td>4.9</td>
<td>Connection between Different Modules</td>
<td>46</td>
</tr>
<tr>
<td>4.10</td>
<td>Alternative Configuration of Transputer and Controller</td>
<td>49</td>
</tr>
<tr>
<td>4.11</td>
<td>Alternative Configuration of Behavioral Modules</td>
<td>50</td>
</tr>
<tr>
<td>5.1</td>
<td>The Head Declaration of Class frame</td>
<td>56</td>
</tr>
<tr>
<td>5.2</td>
<td>The Use of Class frame</td>
<td>57</td>
</tr>
<tr>
<td>5.3</td>
<td>Active Object</td>
<td>57</td>
</tr>
<tr>
<td>5.4</td>
<td>Communicated Processes Producer and Consumer</td>
<td>58</td>
</tr>
<tr>
<td>5.5</td>
<td>Active Objects Producer and Consumer</td>
<td>59</td>
</tr>
<tr>
<td>5.6</td>
<td>A Behavioral Chart</td>
<td>60</td>
</tr>
<tr>
<td>5.7</td>
<td>SERVO Module</td>
<td>62</td>
</tr>
<tr>
<td>5.8</td>
<td>JOINT Module</td>
<td>63</td>
</tr>
<tr>
<td>5.9</td>
<td>The Parallel C++ Codes for the Two Processes of SHOULDER</td>
<td>67</td>
</tr>
<tr>
<td>5.10</td>
<td>SHOULDER Module</td>
<td>68</td>
</tr>
<tr>
<td>5.11</td>
<td>The Parallel C++ Codes for Module SHOULDER</td>
<td>68</td>
</tr>
<tr>
<td>5.12</td>
<td>The Parallel C++ Codes for Module JOINT</td>
<td>69</td>
</tr>
<tr>
<td>5.13</td>
<td>Position Commands</td>
<td>70</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>5.14</td>
<td>Orientation Commands</td>
<td>72</td>
</tr>
<tr>
<td>5.15</td>
<td>Behavioral Chart for findObject</td>
<td>74</td>
</tr>
<tr>
<td>5.16</td>
<td>Behavioral Chart for releaseObject</td>
<td>75</td>
</tr>
<tr>
<td>5.17</td>
<td>Phase 1: Robot Begin to Search for Trees</td>
<td>77</td>
</tr>
<tr>
<td>5.18</td>
<td>Phase 2: No Object Found Along that Direction</td>
<td>78</td>
</tr>
<tr>
<td>5.19</td>
<td>Phase 3: Robot Begins to Search in Another Direction</td>
<td>79</td>
</tr>
<tr>
<td>5.20</td>
<td>Phase 4: Robot Grasps a Tree and Puts it Down</td>
<td>80</td>
</tr>
<tr>
<td>A.1</td>
<td>FCP++ Program for Figure 3.4</td>
<td>88</td>
</tr>
<tr>
<td>A.2</td>
<td>FCP++ Program for Figure 3.6</td>
<td>89</td>
</tr>
<tr>
<td>B.1</td>
<td>The Head File for AFSM</td>
<td>92</td>
</tr>
<tr>
<td>B.2</td>
<td>Finite State Machine in C</td>
<td>92</td>
</tr>
<tr>
<td>B.3</td>
<td>Processes for Resetting the Machine and Setting Registers</td>
<td>93</td>
</tr>
<tr>
<td>B.4</td>
<td>Initialization of AFSM</td>
<td>94</td>
</tr>
<tr>
<td>B.5</td>
<td>Suppress Function in Parallel C++</td>
<td>95</td>
</tr>
<tr>
<td>B.6</td>
<td>Multiplex Function in Parallel C++</td>
<td>96</td>
</tr>
<tr>
<td>B.7</td>
<td>AFSM system in Parallel C++</td>
<td>96</td>
</tr>
<tr>
<td>C.1</td>
<td>FJC Projection Technique</td>
<td>99</td>
</tr>
<tr>
<td>C.2</td>
<td>FJC for Joints Controlling Length</td>
<td>100</td>
</tr>
</tbody>
</table>
Acknowledgement

It is my luck to have Professor Alan Mackworth as my supervisor. It was Alan who first introduced me to the areas of Robotics and Logic Programming, which have become and will still be my major research interest. His perspective view and extremely wide knowledge has inspired my work during the years. Alan’s generous financial support is also greatly acknowledged.

Most work in this thesis was done under the guidance of Professor Peter Lawrence. It was always pleasant to work with Peter. In fact, the most fruitful result of the thesis comes from the discussions with him.

Many people have contributed to this thesis. Don Acton helped me to integrate Parallel C with C++. Real Frenette built the hardwire for interfacing the Arm with transputers and the touch sensor on the gripper. Joe Poon told me how to use the transputer system of E.E. Alex Kean, Jane Mulligan, Felix Jaggi and Herbert Kreyszig read the draft of this thesis and made valuable comments.

Last but not least, I would like express my thanks to Runping for his friendship, love and academic cooperation during the years. He was always the first reader of any part of the thesis. Most of the pages embody his contribution from the contents to the style.

This thesis is dedicated to my parents and brother. Without their constant support and encouragement, I can never fulfill my goal.
Chapter 1

Introduction

A new form of computation is emerging. Propelled by advances in software design and increasing connectivity, networks of enormous complexity known as distributed computational systems are spreading throughout offices and laboratories, across countries and continents.


The ongoing research in Open Systems explores a new form of computation [Hub88], which has inspired and will increasingly influence other areas such as AI and Robotics.

We call a robot system an Open Robot System if the robot can

- work in unstructured and unpredictable environments. By unstructured environment, we mean that there is no complete description of the world in which the robot is working, since the real world is too complicated to be described completely and accurately. By unpredictable environment, we mean that there are some events such as obstacles moving around, dynamically changing of the environment, which are impossible to be predicted in advance.
• deal with incomplete and inconsistent information. In a real world, one can receive information from a variety of sources. Information from each of the sources is partial or incomplete, and also changes with time. Besides, there is inconsistency or redundancy in the information flowing from different sources;

• accomplish complex tasks and respond in real-time. The robot is able to deal with complex situations (reasoning and planning if necessary) and to react to the stimuli from various sources in real-time.

We call this kind of robot an open system because it works in a non-closed world. An open robot should be able to coordinate with other open robots working in the same environment or communicate with human beings. It has incomplete knowledge about its environment but can get various information by its multiple effective sensors. It has no accurate dynamic model nor fine trajectory planner, but can chase moving objects, avoid unpredictable obstacles, and achieve its goal. It should also have the capability of protecting itself from being damaged and of avoiding hurting human beings. This kind of robot is extremely useful for space exploration, undersea applications, and forest harvesting.

Most existing robot systems are not open robot systems. Current robots have very limited abilities to adapt to the environment, to survive in unstructured situations, to deal with incomplete and inconsistent information, and to make real-time decisions. One reason for this is the limited power of computation; another reason is an inappropriate decomposition of the robot architecture and the unsuitable representation with rigid programming style.

In previous approaches to robot system design, a robot system is decomposed into a set of sub-systems according to the external functions — sensors (obtaining information from the environment), central brain (reasoning on the knowledge base), and manipulators (acting on
A Behavioral Approach for Open Robot Systems

the environment). There are two disadvantages of this kind of decomposition. Firstly, such a decomposition is not suitable for large and complex systems. The interface between the sub-systems would be extremely complicated as each of the sub-system grows. Secondly, there is no tight coordination between sensors and manipulators so that the system cannot deal quickly with unstructured and unpredictable situations.

In previous approaches to robot programming, the action sequence of the robot is precisely determined by the program, and the position of the actuator has to be explicitly represented. There are two serious practical problems and limitations. Firstly, there is no explicit structure for sensor–motor coordination and the use of sensors to handle uncertainties is ad hoc. Secondly there is no explicit structure for multiple layers of control and decision making.

The core of this thesis is a new kind of design and programming methodology needed for open robot systems. We have developed a general framework for open robot systems and a programming methodology for building open robot systems.

Our approach to robot design is:

*decompose a robot system into a set of hierarchical behavioral modules according to the logical or physical structure; each of the module is possibly further decomposed into a set of concurrent processes representing sensors, motors and their coordination according to task-achieving behaviors.*

A behavioral module can be incrementally constructed by adding more processes and communication links. Each behavioral module is independent, but communicating with the rest of the system via message passing through channels. Multiple behavioral modules can be assembled into a complex system, whose overall behavior is achieved by the communication among the behavioral modules.
This approach takes the coordination between sensor, motor and memory as the first principle of any "living" system, rather than building sensor, motor and memory independently. The hierarchical architecture provides an interpretation of describing a complex behavior from many different levels of details, and also provides a way of implementing a robot system with complex behaviors.

Our approach to robot programming is: build the behavioral modules based on parallel objects. Programming is a way of designing each individual module and constructing the modules into a complex system.

Parallel distributed and object oriented programming provides both effectiveness and expressiveness for building real-time and multi-sensory robot systems. Object oriented methodology supports modularity and information hiding technique for large and complex systems. Parallel distributed processing supports parallel processes and communication for constructing motor sensor coordination and various control strategies for real-time systems. As a whole, parallel distributed object oriented facilities provide the preliminary constructs for programming open robot systems.

We have built a software system for a real robot arm — Excalibur to simulate the tasks in forest harvesting, based on our approach to open robot system design and programming. The system is written Parallel C++, and implemented in a transputer-based parallel environment.

The goal for building such a system is two-fold:

- Investigate and explore the suitable framework and the programming methodology for open robot systems. The forest is a good environment for our research and experiment because it is a typical unstructured and unpredictable environment.

- Build forest robots which are intelligent, flexible and robust. The forest industry is a
big market for robot systems, and autonomous robots have prospective future in the
application to forest harvesting.

This thesis is only the beginning of our long-term research, whose goal is to build open
robot systems with intelligence, flexibility and robustness, which can work in unstructured and
unpredictable environments.

Outline of This Thesis

This thesis is organized as follows: Chapter 2 gives a survey of related work on robot system
design and programming, which is most influential to ours. Chapter 3 discusses our approach
towards the problem — building the robot system via behavioral modules and building behav­
ioral modules via parallel objects. Chapter 4 describes how to build an autonomous robot
for forest harvesting in the transputer-based parallel environment, which demonstrates our ap­
proach to design an open robot system. Chapter 5 illustrates how to use parallel C++ — a
parallel distributed object oriented language, to model the robot system based on our design.
Chapter 6 summarizes our current work and point out the directions for the further research.
Chapter 2

Related Work

After all, a robot is generally reckoned to be a machine 'made in man's image'.
— Geoff Simons "Is Man a Robot?" John Wiley & Sons, Page xv.

The problem of developing a kind of open robot system has been recognized as a critical area in robotics. The solution will be challenging to the fundamental theory of Artificial Intelligence. While little work on representing such a robot system has been done, some notable contributions have recently been made along this line.

Brooks and his colleagues did very interesting work on building artificial creatures. Brooks [Bro86] [BC86] proposed a robust layered control system, called subsumption architecture, for mobile robots. Layers in this architecture are made up of asynchronous modules that communicate over low-bandwidth channels. Each module is an instance of a fairly simple computational machine. Higher-level layers can subsume the roles of lower levels by suppressing their outputs. Unlike the traditional decomposition of a mobile robot control system into functional modules, Brooks decomposed of a mobile robot control system into task-achieving behaviors. Such a decomposition meets the requirements of multiple goals, multiple sensors and robustness. With this idea, Brooks [Bro87a] further proposed a hardware retargetable distributed
layered architecture for mobile robot control, and successfully built several systems based on this kind of architecture. Herbert, a second generation mobile robot [BCN88], is a completely autonomous mobile robot with an on-board parallel processor and special hardware support for the subsumption architecture, as well as an on-board manipulator and a laser range scanner. The robot is capable of real-time three-dimensional vision, while simultaneously carrying out manipulator and navigation tasks. Connell [Con88] described a behavior-based arm controller that is composed of a collection of fifteen independent behaviors which run, in real-time, on a set of eight loosely-coupled on-board 8-bit microprocessors. Brooks [Bro88b] further suggested one possible mechanism for analogous robot evolution by describing a carefully designed series of networks, each one being a strict augmentation of the previous one. The six legged walking machine, built under this framework, is capable of walking over rough terrain and following a person who is passively sensed in the infrared spectrum. Brooks [Bro87b] also pointed out that robots with insect-level intelligence can be very practical to a large class of jobs in unstructured environments such as home, agriculture and space. The philosophy behind the approach is:

- The fundamental decomposition of the intelligent system is not into independent information processing units which must interface with each other via representation. Instead the intelligent system is decomposed into independent and parallel activity producers which all interface directly to the world through perception and action. The notions of central and peripheral systems evaporate — everything is both central and peripheral [Bro88a].

- The separation of planning and plan execution is an inappropriate approach for the behavior design. We must move from a state based way of reasoning to a process based way of acting, i.e., to set up appropriate, well conditioned, tight feedback loops between sensing and acting [Bro87c].
However, Brooks did not explore general purpose manipulators. A robot from Brooks' company can only perform particular tasks when it is built. This is very similar to the biological case (e.g. a rabbit is born to be a rabbit), but is not practical for a wide range use. Besides, Brooks did not pay attention to parallel distributed techniques. His augmented finite state machines are simulated in conventional Lisp [Bro89], without the explicit notion of parallel processes and communications.

Rosenschein [Ros85] proposed the situated-automata approach, which seeks to analyze knowledge in terms of relations between the states of a machine and the states of its environment over time, using logic as a metalanguage in which the analysis is carried out. This approach, in contrast to the interpreted-symbolic-structure approach which has prevailed in the research of Artificial Intelligence over the decades, provides a way of compromising the power of representation and real-time execution of AI systems. By this approach, the specification written in modal logic is first compiled into a circuit which is composed of parallel, serial or feedback components [RK87]. The compiled circuit, which has direct links to the environment, can run efficiently. Kaelbling further developed this idea [Kae88] and pointed out that classical planning is inappropriate for generating actions in a dynamic world. He presented a formalism, called Gapps, that allows a programmer to specify an agent's behavior using symbolic goal-reduction rules which are compiled into an efficient parallel program. Using the relationship between logic and situated-automata, Rosenschein [Ros89] proposed an approach of synthesizing information-tracking automata from environment descriptions. By this approach, a description of the automata can be derived automatically from a high-level declarative specification of the automata's environment and the conditions to be tracked. The output of the synthesis process is the description of a sequential circuit which at each clock cycle
updates the automata's internal state in constant time, preserving the correspondence between the states of the machine and conditions in the environment. The proposed approach retains much of the expressive power of declarative programming while ensuring the reactivity of the run-time system. This approach is interesting in the sense that an inefficient but expressive program can be compiled into an efficient program which can be executed in real-time and react to the environment. However, the syntax of the specification language is based on modal logic which is not easily comprehended. Furthermore, there is no methodology for constructing programs in that language. Therefore, applicability of the approach is limited. Rosenschein and Kaelbling further discussed the architecture for robot systems, in which planning and reactive control are combined [RK89]. They showed that the distinction between planned and reactive behaviors is largely in the eye of the beholder: systems that seem to compute explicit plans can be redescribed in situation-action terms and vice versa. Kaelbling [Kae87] proposed an architecture for intelligent reactive system which is motivated by the desires for modularity, awareness, and robustness: the system should be built incrementally from small components that are easy to implement and understand; at no time should the system be unaware of what is happening; it should always be able to react to unexpected sensory data; the system should continue to behave plausibly in novel situations and when some of its sensors are inoperative or impaired. Kaelbling discussed the relationship between planning and perception and proposed a framework for adaptive hierarchical control. Thus, if the more competent levels fail or have insufficient information to act, the robot will be controlled by a less competent level that can work with weak information until the more competent components recover and resume control.

Brooks and Rosenschein did not give a clear view or methodology of programming a robot system, therefore these ideas have not been used widely in Robotics. Up to now, a lot of work has
been done on robot programming and real-time operating systems for robot control. Hayward [HH88] investigated the approach to programming multiple arms using Kali — an environment for programming cooperative control systems. However, based on very loosely coupled computers, Kali has limited ability to support multi-processing and communication. Narasimhan et.al. [NSH88] described a standard architecture for controlling robots. But this architecture is more like “device-host” oriented, rather than parallel distributed objects. Salkind [Sal88] from the Courant Institute developed an operating system, SAGE, which runs on a Motorola 68020 processor board, and provides a variety of services, including process and memory management, protocol support, and precise timing facilities. The overall communication is, however, based on shared memory approach.

Object oriented methodology provides both modularity and data abstraction which can be very useful for programming a large and complex system. Kafura [Kaf88] proposed a way of building real-time systems with a concurrent object oriented programming language. The language’s underlying model of concurrency reflects the distributed and concurrent nature of real-time systems while the language’s object orientation and the reusability properties of class inheritance address the embedded and evolutionary aspects of real-time systems. Languages, however, only cover half of the story, i.e., the basic constructs which can be used. The other half, programming methodology in the language, which usually reflects the theme of the architecture of the system, are equally important. Smithers and Malcolm [SM87] proposed a behavioral approach to robot task planning and off-line programming. This approach, corresponding to Brooks’ decomposition, is believed to be robust and suitable for uncertainty handling. However, they neither described underlying languages, nor gave any examples to illustrate the programming methodology.
Here in the departments of Computer Science and Electrical Engineering, some important contributions have been done on Robot Vision and overall architectures.

Mulligan et. al. [MML89] described a vision system for visual control of the manipulator arm that achieves a tight coordination between perception and action, illustrating a typical module that could be part of an open robot system. Poon [Poo88] proposed a kinematics control algorithm — Functional Joint Control (FJC), in which each joint iteratively computes its movement. FJC has two advantages over the other inverse kinematics algorithms. Firstly, it is possible to implement this algorithm in a parallel computing environment, which will speed up the computation. Secondly, it is possible to incorporate the tight coordination between sensors and actuators into this algorithm. These advantages will become obvious in Chapter 5, where we present a distributed control system for a robot arm, using a method derived from FJC.
Chapter 3

A Behavioral Approach to Robot Design and Programming

Whichever model we take as primary is often largely a psychological matter — a reflection of taste, hobbies, and habits.

— Geoff Simons “Is Man a Robot?” John Wiley & Sons, Page 30

In this chapter, we propose our general framework of robot system design and programming. Section 1 presents our approach, a two-dimensional hierarchical architecture to robot system decomposition. We show that a robot system is generally a typical distributed system and robot control should be considered as a communicative process structure. Furthermore, we argue that such a structure should have a tight coordination between sensors and manipulators, as well as a multiple layer of description of behaviors. Section 2 discusses the characteristics of a parallel distributed object oriented language for robot programming. We illustrate, through two typical examples, the advantages of a parallel distributed object oriented framework over the traditional approach to robot programming.
3.1 Building an Open Robot System via Behavioral Modules

An open robot system would be equipped with multiple sensors for effectively reacting to the change of the environment and communicating with other agents (robots or human beings). Robot systems of this kind are inherently distributed since multiple sensors and manipulators are often distributed on different processors. Unlike previous approaches in which the computer was taken as a central computing machine, while sensors and manipulators were considered as input–output peripherals, we represent an open robot system as a distributed system. The processes of sensors and manipulators are designed to be independent, but communicating with each other.

There are several differences between a sequential and distributed version of robot control. In the sequential programming paradigm control is programmed as an algorithm with suitable data structures for solving algebraic and differential equations of robot kinematics and dynamics:

\[
control = data\_structures + algorithm
\]

In the distributed programming paradigm control is accomplished by a set of processes and their communication:

\[
control = process\_structure + communication\_protocol
\]

*Feedback* is considered as a central theme in conventional control theory. However, we consider *coordination* among multiple processes in a distributed control system as a general mechanism. Feedback, inherent in processes, is a special case of coordination.
3.1.1 Behavioral Decomposition

Unlike most previous approaches, which decompose the system into sub-systems according to the external functions: sensors, effectors, knowledge base, reasoning and planning mechanisms etc., Brooks [Bro86] proposed an approach of behavioral decomposition, which decomposes the system into task-achieving behaviors. A behavior, modeled as an augmented finite state machine — AFSM, is an integration of perception, action and cognition. An AFSM is composed of a set of input registers, a set of output channels, and a finite state machine. Several behaviors can be connected directly, or via a suppress or inhibit mechanism. All the behaviors are distributed in layers of a hierarchical structure. Primitive behaviors such as avoiding obstacles are distributed in lower layers, while advanced behaviors such as building maps are distributed in higher layers. A system can be built incrementally by starting building most primitive behaviors and adding more and more advanced behaviors.

There are several advantages of this approach over previous ones:

- Under behavioral decomposition, we can build incrementally complex behaviors and connect them in an appropriate way. This architecture is much clearer than the approach that adds unrelated facts and rules to a single database and builds more and more complicated sensors and effectors.

- Behavioral decomposition can represent tight coordination between sensors and motors, which is essential for dealing with unstructured and unpredictable environments. Besides, various control strategies can be used to make the robot fulfill certain kind of tasks, while retaining the basic survival mechanisms. Multiple layers of decision making for complex behaviors can also be constructed.
• Systems with behavioral decomposition can be robust. Since each behavior is independent, any behavior only affects the system locally. Even though one behavior may stop functioning, the rest of the system will continue working.

We propose a more general approach for behavioral decomposition. Unlike Brooks, we decompose the system into a set of cooperative processes, instead of AFSMs, suppress or inhibit nodes, which are just the special kinds of processes. Each process consists of a set of input and output channels, as well as a body, which can be any set of functions including channel communication, creating new processes, etc. Processes can communicate with one another through input–output channels or via shared memories. A process can function in sensing, acting, as well as coordinating. A group of processes can be taken as an integrated module, which can be used for constructing more complex behaviors.

Basically, there are two kinds of processes. One is called a permanent process, which is created whenever the system is instantiated. The other one is a dynamically generated temporal process, which is created for certain tasks and dies when the task finishes. On the other hand, there are two kinds of action and perception. One is called a purposive action or perception, which is task-dependent. The other one is called a reactive action or perception, which is task-independent, but for primitive survival or adaptive mechanisms. In general, a dynamical process corresponds to a purposive action or perception, and a permanent process is for a reactive action or perception.

The overall behavior is described by functionality of each process, and the coordination among processes, i.e., how various processes are communicating through channels.

The advantages of our generalization are:

• the existing theory and methodology on parallel and distributed systems can be applied
to robot systems;

- A robot system can be built on the top of a general computing environment, instead of on a special purpose hardware;

- Part of the system can be dynamically configured via dynamic processes, while Brooks' systems only can have fixed configuration;

- The approach of decomposing the behavior into sensorimotor coordinated processes has a strong background with results from psychology and physiology [Bra84]. Building systems based on this approach can lead us to a more general (probably more fundamental) understanding on the questions such as “What is a living system?”, and finally “What is intelligence?”.

3.1.2 Structural Decomposition

An open robot system often consists of many physical or logical parts, for example, a robot can have one or more arms, each of which is associated with a hand at the end; besides, it can have a mobile base, one or more visual sensors; furthermore, the arm is composed of several joints, each of which has an angle position sensor; the hand can be composed of several fingers, each of which is composed of several joints and touch sensors; the mobile base can be composed of wheels or legs, and so forth. A behavior usually involves the operations on these parts and the coordination between these operations.

When the structure of the system becomes complicated, it will be difficult to represent a behavior in one level. For example, the “avoid obstacles” behavior would be easily modeled

\[\text{Currently, there are some interesting discussions in the news group called "Cybernetics and Systems", in the computer network, on the relationship between cognitive structures and communicative process structures which can be considered as another support to our approach.}\]
as one process, if the structure of the robot is abstracted as a point. However, if we try to represent the same behavior for a more complex robot, which has a base, an arm and a hand, “avoid obstacles” would mean “base avoid”, “arm avoid” as well as “hand avoid” obstacles. Finally such a behavior is carried out via the servo motor control.

Since an open robot system can be composed of hierarchically distributed logical or physical devices, and exhibits various layers of behaviors, we decompose the system into a hierarchical structure of behavioral modules. From higher to lower levels, for instance, we can have

![Figure 3.1: Structural Decomposition of the Robot System](image)

**Figure 3.1: Structural Decomposition of the Robot System**

**ROBOT; ARMs, HANDs, BASE and EYEs; JOINTs, FINGERs, LEGs etc.** as the type of behavioral modules over the layers (see Figure 3.1). This particular decomposition, however, is only one example of how the system looks. Different systems with different configurations can have different decompositions. As an example, an arm can be one part of a base, a hand can be one part of an arm, so and so forth. But the principle is the same.

Each module has a set of inputs and outputs for communication. All the modules are

---

2Here we only represent the structural decomposition. A line doesn’t mean a link between two modules.
working in parallel. The communication between any two modules is achieved by message passing through channels, so that the control of the whole system is totally distributed.

Generally, the characteristics of structural decomposition are,

- **abstraction**: the higher level a module is on, the more abstract the operations it describes. Usually the abstract operations can be carried out by several lower level modules. In our example, the behavior of ROBOT could be carried out by lower level behaviors of BASE, ARM, HAND and EYE, the behavior of ARM could be further carried out by the behaviors of JOINTs.

- **coordination**: the higher level a module is on, the more coordinations between different parts take place in it. In our example, ROBOT is the coordinator of BASE, ARM, HAND and EYE, ARM is the coordinator of JOINTs. The lowest level SERVO only considers the control of DC motors.

The advantages of structural decomposition are:

- even though the system is designed to be totally distributed, i.e., modules can be physically on different processors and each module starts functioning independently, there are still multiple levels of coordination between different parts.

- we can explain the overall behaviors in different layers, from coarse to fine; on the other hand, we can implement a complex behavior through many layers of abstraction. As an example, the behavior of "grasp" can be implemented by a dextrous hand or a simple gripper; however, this makes no difference to a higher level.
3.1.3 Two–dimensional Decomposition

We extend behavioral decomposition to a two–dimensional hierarchical structure by adding another kind of decomposition — structural decomposition (see Figure 3.2). The overall system consists of two orthogonal hierarchies: one is structural decomposition (vertical), which represents the abstraction or the coordination hierarchy; the other is behavioral decomposition (horizontal), which represents the evolution hierarchy — one can incrementally add more behaviors on each behavioral module.
An open robot system is developed along both dimensions.

Structural decomposition decomposes the system into a hierarchy of behavioral modules according to system's physical or logical structure. A system can be built with a few simple devices at the beginning, and then be elaborated by adding more devices such as sensors and manipulators, or by replacing the simple devices with complex ones.

Behavioral decomposition decomposes the behavioral modules into a set of behaviors which are the integration of perception, cognition and action. A behavioral module can be built with simple primitive behaviors at the beginning, and elaborated by incrementally adding more advanced behaviors.

For the modules on lower levels of our structural hierarchy, their behaviors are probably independent of any particular task, referred to as reactive behaviors. These behavioral modules have the basic mechanisms for reacting to the environment and for surviving under different situations, as well as following the instructions from higher levels. Higher level behavioral modules usually consist of both task-related behaviors and reactive behaviors. Brooks' system corresponds to the top level module in our structural hierarchy.

The two-dimensional decomposition has the advantages of both of structural decomposition and behavioral decomposition. The system built in this structure is a totally distributed system with multiple layers of coordination. Besides, such an architecture is much more robust than both of the one-dimensional decompositions. Even though a module or a behaviors may stop function, the rest of the system will continue working.
3.2 Building Behavioral Modules via Parallel Objects

3.2.1 Characteristics of Parallel Distributed Object Oriented Languages

The distinction between parallel and distributed languages is very vague. With parallel, we emphasize the parallel computations; with distributed, we emphasize the independence of each computation and the communication between the computations. We call a language parallel and distributed, if we emphasize both aspects. Furthermore, we call an object in a parallel distributed object oriented language a parallel object if the object is defined as a combination of processes and communications.

A parallel distributed object oriented language, in general, provides both concurrency and modularity. Usually, it should have the following basic primitives:

- primitives for defining classes, which support data abstraction and modularity;
- primitives for process definition, creation and synchronization which support concurrency;
- primitives for communication between different processes via channels, which support locality.

With these primitives, the language has the following characteristics which are very important for behavioral modeling, since they are corresponding to our principles to system decomposition:

1. supporting concurrent multi–agent architecture

An object in a parallel distributed object oriented paradigm can model an active agent with a set of communication channels which support message passing between objects. An object can be taken as an instance of a device or a service which is independent, but communicating with the other parts of the system through the designed interfaces.
Such a concurrent multi-agent architecture provides a kind of organization with a large collection of computational services and interactions.

The resultant system based on this model can be robust, able to deal with uncertainty and local failures, and makes it possible to accomplish tasks under situations with inconsistency and incompleteness.

The role of a parallel object in a parallel program is similar to that of an integrated circuit in a large electronic system. This kind of modularity is critical for both the system's design and maintenance. From a design point of view, system programming is simply a way of choosing suitable objects, and assembling them to achieve some specified behaviors. From the maintenance point of view, changes can be done locally: just "pull-out" the "wrong" component, repair or change it, and "plug-in" the "right" one.

2. supporting communication via channels and decentralized control mechanism

For a real-time system, it is often inefficient to keep a global database for communication. The centralized control would become a bottleneck of an open robot system with multiple sensors and manipulators. One of the most important advantages of using message passing via channels and decentralized control is to keep locality. This ensures that each component can be kept relatively independent. The control and communication interface would be simple and clear. Furthermore, asynchronous control and channel communication provides the basic constructs for building sensorimotor coordination and multiple layers of decision making.

3. supporting information hiding and abstraction

Object oriented programming provides techniques for information hiding [PCW87]. Using
these techniques we can easily represent the hierarchical behaviors of the system. On the other hand, parallel distributed techniques provide concurrency, so that each of the behavior modules can run in parallel and distributed over different processors.

3.2.2 Programming on Parallel Distributed Object Oriented Languages

Programming a robot system under the parallel distributed and object oriented framework is totally different from programming using a conventional language. The former provides explicit structures for parallel execution of commands and sensorimotor coordination.

- The overall program is not a sequence of commands, rather, the program is a structure which represents the processes and their communication.

Under this framework, off-line programming for uncertainty or incompleteness can be explicitly represented. We can have many processes to deal with various kinds of sensory information, and the resultant sequence of actions for the manipulators partly depends on the sensory inputs. On the other hand, suitable mapping of processes to processors will make it possible to react to any sensory inputs in real-time.

To state the characteristics clearly, let's consider a typical example—a robot manipulating the blocks world, and compare the differences between the traditional program and the parallel distributed object oriented program:

If the robot is asked to fulfill certain goals in the blocks world, the previous approach to the programming would be illustrated through a program shown in Figure 3.3. Here, the sensory information is only derived from the first function call. This program generates a sequence of commands: using the sensors to find out the initial condition of the block

\footnote{We use Prolog to give the example. The functionality would be the same as any other sequential languages.}
Figure 3.3: Traditional Approach to Programming of Using Sensors and Plans

world, generating a plan based on the condition and the desired goal, and executing this plan which is a sequence of action commands such as “move to A”, “pick up B” etc. The problem of this program is that the use of sensor information is once and for all. The robot is unaware of the environment when it concentrates on generating the plan and executing the plan. A robot of this kind cannot be used in unstructured and time-varying or unpredictable environments, since the sudden changes can cause an accident or even damage the robot.

In the parallel and distributed paradigm, we can construct a program which uses sensors to get the continuous information, use a planner to continuously generate actions and uses manipulators to performs the actions. These three components are working in parallel communicating with one another. So that any changes of the conditions would affect the plan generated and also affect the action sequences. Our program generates a communicative process structure as shown in Figure 3.4, where each box is a parallel object (agent), or a process. Lines between two boxes are communication channels.

• Since sensors and manipulators can have an explicit coordination under the new programming paradigm, the program for motion becomes easier and clearer.
As an example, consider the execution of a task "pick up a block, put it on the central place". In the conventional approach, the program have to specify the the complete trajectory of motion (see Figure 3.5). This program first sees the position and orientation of the object, which is represented as a kinematic frame F1, then issues a command to the robot for moving to that frame, followed by another command for grasping the object, etc. The same problem occurs as the one in the previous example. If, during the execution,
the object is moved (by other robots or human beings) or an obstacle suddenly appears on the path of the motion, the task will simply fail. Besides, this kind of representation for motion makes the vision system extremely difficult to build, since the visual processing is asked to give the precise position and orientation which have six degrees of freedom.

In parallel and distributed framework, the program represents a communication structure between the processes move, see etc. The function of vision is to give continuous information (probably not so precise at times) to guide the motion. The "move-to" task is coordinated with the built-in permanent process "avoid obstacles" as shown in Figure 3.6. SENSOR1 continuously obtains the information about the object, such as direction

![Diagram](image)

Figure 3.6: Motion Planning under Communicative Process Structure

and distance; SENSOR2 continuously outputs the signals of obstacles, which will affects the direction of the movement.

The corresponding programs for Figure 3.4 and Figure 3.6 written in FCP++ — Object Oriented Flat Concurrent Prolog are described in Appendix A.
Robot System Design and Programming:

In the next two chapters, we present an application of the framework we proposed here. A robot system design and programming involves the following steps, which can be interleaved during the whole procedure:

- Decomposing the robot system into a hierarchical structure of behavioral modules based on configuration of the robot hardware and parallel distributed computing environments.

- Designing the interface of each module and communication protocols between different modules.

- Modeling the behavioral module using parallel distributed object oriented languages — implement behavioral modules via concurrent processes and communication.
Chapter 4

Design of an Autonomous Robot for Forest Harvesting

*General system theory is a set of related rules, definitions, assumptions, propositions, etc. — all of which relate to the behavior of organizations, of whatever type.*

— Geoff Simons *Is Man a Robot?* John Wiley and Sons, Page 93

In this chapter, we illustrate our approach, the structural decomposition, by going through the design procedure for an autonomous robot in a transputer-based multicomputer environment. Section 1 describes the desired behavior of an autonomous robot in a forest environment; Section 2 discusses the installation of robots for such an environment in general; Section 3 concentrates on a particular robot arm in our lab — Excalibur \(^1\), and the tasks we want the robot to perform for simulating the forest harvesting work; Section 4 discusses the structural decomposition of the current system, the interface design for each behavioral module. Section 5 proposes an alternative structural decomposition under another configuration of the transputer system and the robot controller.

\(^1\) Robotics Systems International, Sydney, B.C. Canada
4.1 Behaviors in Forest Harvesting

Much research has been done on the application of tele-robotics (tele-operation) to forest harvesting by a group led by P. D. Lawrence in Electrical Engineering at UBC. We propose here an approach to building totally autonomous robots which can take over the dangerous and tedious work without the assistance of human beings. Some tasks or behaviors for such robots are:

- **moving logs**: the robot walks around, puts the scattered logs into a pile of logs, or puts the logs onto a truck;

- **cutting down trees**: the robot looks for the trees which should be harvested, and cuts them down;

- **planting new trees**: the robot digs a hole at the right place, and puts a new seedling.

An autonomous robot, unlike a tele-robot, performs these tasks independently. It is a sensor-guided agent with the essential mechanisms for surviving and avoiding hurting human beings. On the other hand, an autonomous robot for the forest environment, unlike one for the assembly industry, performs tasks which do not require high accuracy or speed, but require real-time reaction to the changes of the environment. There are no well-defined world models for the forest environment, and there are no pre-planned action sequences for the robot manipulator.

Each autonomous robot is programmed for performing a particular task or the combination of tasks. We can have many such autonomous robots working together in a large forest: some moving logs, some cutting down trees and some planting new seedlings etc. These autonomous robots may replace the tele-robots some day.
4.2 The Robot

An autonomous robot for forest harvesting should be an open robot which may be equipped with multiple end-effectors, multiple manipulators, as well as flexible and robust sensors.

Physically, we prefer to use multiple simple sensors, each of which needs very little computation, rather than a complex one which requires a great amount of computation. Logically, we need multiple meaningful sensory signals and distributed memory, instead of a complicated description of the whole world in a large database. Currently, there is a variety of sensors which are very useful for our autonomous robot [FGL87]:

- **range sensor** — a range sensor measures the distance from a reference point (usually on the sensor itself) to the objects within the operational range of the sensor. Range sensors are mostly used for robot navigation and obstacle avoidance.

- **proximity sensor** — a proximity sensor outputs a binary signal which indicates the presence of an object within a specified distance interval. Typically, proximity sensors are used for near-field work in connection with object grasping or avoidance.

- **touch sensor** — touch sensors can generally be classified under two categories: binary and analog. A binary sensor is like a switch which responds to the presence or absence of an object. An analog sensor, on the other hand, outputs a signal in proportion to a local force. Touch information can be used to locate and recognize objects, as well as to control the force exerted by a manipulator on a given object.

- **force and torque sensor** — a force and torque sensor measures the Cartesian components of force and torque acting on the joints of a robot arm or measures the deflection of the mechanical structure of a robot due to external forces. Force and torque sensors are
A Behavioral Approach for Open Robot Systems

used primarily to feedback the reaction forces to the robot controller for adaptive force control.

- **joint position sensor** — a joint position sensor measures the current joint angles which are mainly used in the kinematics loop for the position control. A system using visual sensing for measuring joint angles is presented in [MML89].

- **vision sensor** — the advantage of using vision sensors, in most cases, cameras, is that a vision sensor can capture various kinds of information. It is generally believed that for human beings ninety percent of sensory information is from the visual input. However, visual perception needs much more computation than other sensing. There are two ways to alleviate the problem: one is to restrict the visual task to recognize only the simple features related to a particular task; the other is to use an independent computational unit — a processor in a parallel environment, for performing the major computation and sending out the important features in real-time.

The reason for using more than one of these sensors for an autonomous robot is two folds:

- robustness and fault-tolerance are two of the major concerns for the robot working in a hazardous environment. With multiple sensors and the appropriate interconnections, a robot can continue functioning even though some sensors are broken.

- By using multiple sensors processing the sensory information in parallel we can improve the real-time response of a robot to some extent.

In addition to multiple sensors, an autonomous robot may need multiple manipulators and multiple end-effectors which are coordinated to fulfill a particular task. As for the job of "cut
down trees”, the robot needs a mobile base (a legged robot or a wheeled vehicle) to “walk” around to look for the trees, and an arm with a gripper to grasp the tree and a saw to cut down the tree.

It is not necessary to have the closed-form inverse kinematics for the arm of such a robot. There are two drawbacks of using closed-form inverse kinematics for open robot systems:

- there are heavy computations which cannot be distributed over parallel processors;
- the calculation is once and for all. There is no sensory feedback during the execution.

Instead, a functionally decomposable [Poo88] arm would be very useful, since each of its joints can independently calculate the movement and react to the related sensory information. For example, the joints mainly for position can react to the range sensor to track an object and to avoid obstacles while the joints mainly for orientation can react to the proximity sensor to guide the end-effectors to grasp objects. Besides, an arm would be better to have six degrees of freedom so that it can reach any position in space and its end-effectors can take any orientation.

A robot may have various kinds of tools (end-effectors), including graspers and hands, for fulfilling various kinds of tasks. For example, a robot with a gripper which can open and close can perform the useful tasks such as “pick up” or “release” objects. A hand may be used for more flexibly grasping of objects of irregular shape.

The mobile base, usually a wheeled vehicle or a legged robot, is needed for an autonomous robot working in forest. A wheeled vehicle would be easier to design but not be suitable for rough terrain. A legged robot, however, is more expensive to build.
4.3 Robot Arm Excalibur and its Task Design

Even though our goal is to build open robot systems for forest environments, we would like to build an autonomous robot system in our lab at the first stage, which can simulate tasks for forest harvesting.

Currently, we have a six joint PUMA-like arm, Excalibur, with a simple one-switch touch sensor on its gripper (see Figure 4.1). The functions of the joints [RSI88] are:

- **swing** — this joint connects the base to the turntable, and rotates the entire robot clockwise or counterclockwise on the base;

- **shoulder** — this joint connects the turntable to the upper arm, and provides for the
main pitching motion of the arm;

- **elbow** — this joint connects the upper arm to the upper forearm, and provides a pitching motion too;

- **upper—rotate** — this joint connects the upper forearm to the lower forearm, and provides a rotational motion about the longitudinal axis of the entire forearm;

- **wrist—flex** — this joint connects the lower forearm to the wrist. It may provide a pitching or a yawing motion, depending on the orientation of the upper—rotate;

- **tool—rotate** — this joint connects the wrist to the end—effector, and provides a continuous rotate mode;

- **end—effector** — the current Excalibur is equipped with a parallel—jaw gripper. However, this can be replaced by the user with any number of tools designed to do particular jobs. We installed a simple one—switch touch sensor on the gripper to obtain the information of the presence of an object.

From the above description, we see that this arm is functionally decomposable and can be divided roughly into two independent parts: the first three joints, swing, shoulder and elbow, are for positioning the arm and the last three joints, upper—rotate, wrist—flex and tool—rotate, are mainly for the orientation of the end—effector.

**Excalibur has a master arm,** a scale model of the manipulator, used to control the manipulator manually: the manipulator is designed to follow the master arm in real—time. We did not use it for the current system, but considered its use in an alternative configuration of the system which we will describe in Section 4.5 of this chapter.
The controller of Excalibur consists of a digital computer and several analog circuits for control each joint and the gripper.

The controller accepts a set of commands which can be directly issued from a host computer or an ASCII terminal. On the other hand, the controller can bypass the digital computer by connecting the analog circuits with the manual port. The cable from the master arm can plug in to this port so that analog position signals from the master arm can be used to control the joint movements.

Two of the important commands for kinematics control are:

- “moveby $j_1$ $j_2$ $j_3$ $j_4$ $j_5$ $j_6$” causing joints to move, where $j_i$ is the amount of the $i$-th joint’s movement, and

- “manip” returning the current angles of the six joints from the controller. These angles are used in the kinematics loop.

The commands for controlling the gripper are: “open step” and “close step” where step is the number of steps the gripper will open or close. We can, through the parallel digital input/output port, get the current state of the touch sensor using the command “inp”.

One of our research goals is to program the robots like Excalibur to perform various tasks such as the touch-sensor guided search described as below.

When the robot begins to work, it starts to search for the “trees” within certain range. The arm goes forward with the gripper opening and closing constantly. If touch sensor detects an object when the gripper is closing, then the gripper will keep holding the object. In this situation the hand turns downward and the arm moves down towards ground until an outside signal (a simulated range sensing from the terminal) comes, which represents that the arm is almost
touching on ground. This same signal also triggers the gripper to open so that the object is released on the ground. If the touch sensor detects no object ("tree") before the arm reaches its limited position, the arm moves back and turns an angle (say about 15 degrees), then the robot starts to search along the new direction.

There are two ways to program such robots. In the traditional way the programmer has to know exactly where the object is, and the commands issued to the robot controller cause the robot to move to the positions specified in the commands. This kind of programming paradigm is obviously not appropriate for our purpose.

In programming a robot to perform the task described previously, we have no prior information on the locations of the objects, and do not know when the gripper should grasp or release objects. The action sequence for achieving the task depends on the sensory information (here signals are from the touch sensor and the simulated range sensor).

However with the current installation Excalibur can only perform some simple tasks (like the one described previously). We have no range sensor to measure the distance of the object, no proximity sensor to detect the possibility of the presence of the object, no vision sensor to recognize the shape of the object, no force and torque sensor to measure the external force acting on the end-effector. The robot is blind and numb.

We are planning to install more sensors such as a range sensor and a proximity sensor, a force or torque sensor, or a vision sensor on the Excalibur in future so that it can perform more interesting tasks. In programming a robot with multiple sensors we can follow the same methodology as the one which is discussed in Chapter 3 and will be further illustrated through case study presented in the rest sections of this chapter and in the next chapter.

As an example, if we have a range sensor and a proximity sensor, in order to carry out the
same task, the robot can use the range sensor to detect the position of the tree, thus to guide the motion of the arm (forward and turn); at the same time the robot can use the proximity sensor to detect the existence of the object and to guide the operation of the gripper. When the gripper grasps an object the robot can use the range sensor to detect the distance from the ground and to guide the release operation (gripper opening).

Further, if we add another vision sensor or an array of analog touch sensors, which are used to detect the orientation of the trunk of tree, we can design the robot to perform the tasks such as cutting off the branches of trees. This is accomplished by the robot arm moving along the trunks of trees and performing the cutting operation.

A force and torque sensor is needed if the load (log) may be very heavy so that it will cause the change of the dynamics of the arm. In this case force control becomes necessary.

4.4 Decomposition of Transputer–based Robot Systems

In this section we present the design of our robot control system which is to run on a transputer–based multiprocessor computing environment. A transputer based multicomputer is a general purpose parallel and distributed computing environment which has become popular in the recent years. A transputer is a VLSI building block for concurrent processing systems, comprising of a processor, the on–chip memory and four serial communication links. Two transputer can be connected by connecting a link of one transputer to a link of the other. The flexibility of the reconfiguration of the transputer network makes it possible to support various kinds of parallel and distributed applications. For a real–time application, we need not only consider the logical decomposition of the system, but also consider the mapping of the logical decomposition to the physical transputer network so that the overall computation and communication can be
optimized.

4.4.1 The structural decomposition

We decompose our robot system into a hierarchy of behavioral modules, according to the principles of structural decomposition discussed in Chapter 3 (see Figure 4.2). SERVOs, at the lowest level in the hierarchy shown in Figure 4.2, are responsible for sending or getting signals to or from Excalibur. ROBOT, at the highest level, is designed to perform the particular tasks. ARM, at the intermediate level, plays a role of the coordination of the six joints. Each joint module, from SWING to TOOL, is for calculating the joint movement. Such a decomposition is based on the fact that the joints in Excalibur are functionally decomposable: each joint plays a particular role in the whole movement of the arm, as we described the previous section.
This structural decomposition is highly flexible for further extension and modification. We can install Excalibur on a mobile BASE without changing any other modules except ROBOT. Or we can use a dextrous hand instead of the gripper without modifying any modules of the arm and the joints.

The mapping of the structural decomposition into the physical network of transputers, however, depends on the interface of robot devices and transputers. We can make the mapping as the final step in the design process of a robot control system (by using multiplexing etc.). But for the consideration of efficiency, we need to take this problem into account at the stage of decomposition.

There is no general approach of mapping modules into transputers. However, there are some basic heuristics:

- It is better to map modules which have shared memory communication into one transputer;
- It is better to map modules which share communication channels onto transputers which have direct hardware links;
- It is better to distribute computations with loose dependence over several transputers.
- It should also be taken into account that modules on one transputer should not have more than four links connecting to the modules distributed on other transputers because each transputer has only four hard links.

One of the easiest ways of connecting the robot to the transputer system is to use one link of a transputer to communicate with the robot controller through RS232 board (see Figure 4.3). Based on this configuration, we designed the physical connection of the transputers and
Figure 4.3: Current Configuration of Transputer and Controller

The mapping of the modules to the transputers (see Figure 4.4). In Figure 4.4, we have four transputers connected as a ring, on each of which, there is one integrated module corresponding to one or several logical modules in Figure 4.2. The transputer with SERVO on it has a link directly connected to (the computer of) the robot controller. SERVO is responsible for issuing commands to and receiving the current joint angles and the touch state from the robot controller. JOINT, consisting of six joint modules, from SWING to TOOL, is mainly for computing the joint movements according to the desired position from ARM and the current joint angles received from SERVO. ARM receives the commands from ROBOT, determines the current goal position or orientation and sends them to JOINT. ROBOT, at the highest level, is the module for performing particular tasks. All the inverse kinematics is done in JOINT, which will be described in the module design stage.
4.4.2 The interface of each module

The interface for a module consists of its port descriptions and communication protocols. Only modules with consistent protocols can be connected together. In the rest of this section, we describe the interface of each module in Figure 4.4, and the connections between those modules.

ROBOT module

ROBOT has four ports (see Figure 4.5), whose descriptions are as follows:

- **port1** is an output port, for sending out the commands such as "move forward", "turn left", etc. which can be handled by ARM module;

- **port2** is an input port, for getting the current position state of the arm's end-effector;
• **port3** is an output port, for sending out the desired gripper operations such as “open” and “close”;

• **port4** is an input port, for getting the current state of the touch sensor.

**ROBOT** has two task-independent processes, the permanent processes which are created once **ROBOT** is initialized. These processes are considered as information processors of the environment, which continuously send out the state or the change of the environment. Such processes are critical for real-time reactive mechanisms. In the current design, we have one permanent process for getting the current end-effector position state from **port2**, and the other for getting the touch state from **port4**.

The commands sent out through **port1** or **port3** are determined by the particular tasks. Currently, we have designed two sequential tasks for this robot: “find and grasp an object” and “release the object”. During the process of “find and grasp an object”, **ROBOT** sends out the commands “hand forward”, “arm forward”, “gripper open” and “gripper close” constantly. If the gripper does not touch anything when the arm reaches its limit position, **ROBOT** sends out commands “arm backward” and “arm turn”, then repeats the “find and grasp an object” process; otherwise, this process stops, and the process for “release object” comes into control; During the “release the object” phase, **ROBOT** sends out the commands “hand down”, “arm down”, and finally sends the command “gripper open”, when a signal comes from the terminal (the gripper is touching the ground).

**ARM module**

**ARM** has four ports (see Figure 4.6), whose descriptions are as follows:
• port1 is an input port, for getting the commands such as “arm forward”, “arm turn”, “hand down” etc. which can be processed by ARM module;

• port2 is an output port, for sending out the arm’s current end-effector position state;

• port3 is an output port, for sending out the desired end-effector position or orientation;

• port4 is an input port, for getting the current end-effector position and orientation.

ARM gets commands from port1 and calculates the desired end-effector goal position or orientation and sends the results out through port3. At the same time, ARM continuously reads the current end-effector position and orientation from port4 and sends out the current end-effector position state through port2.

JOINT module

JOINT has four ports (see Figure 4.7), whose descriptions are as follows:

• port1 is an input port, for getting desired end-effector position or orientation;

• port2 is an output port, for sending out the current end-effector position and orientation;
Figure 4.6: Interface of ARM

- **port3** is an output port, for sending out the desired movements of each joint;

- **port4** is an input port, for getting the current joint angles.

**JOINT** constantly reads the current joint angles from **port4**, calculates the arm’s current end-effector position and orientation, sends the result out through **port2**. At the same time, **JOINT** gets the desired end-effector goal position or orientation from **port1**, and calculates each joint’s desired movement, and sends the result out through **port3**.

Figure 4.7: Interface of JOINT module
SERVO module

**SERVO** has four ports (see Figure 4.8), whose descriptions are as follows:

- **port1** is an input port, for getting the desired movements of each joint;
- **port2** is an output port, for sending out the current joint angles;
- **port3** is an input port, for getting the commands of gripper operations.
- **port4** is an output port, for sending out the current state of the touch sensor.

**SERVO** has a process for serializing the commands to the controller and getting the results from the controller. Besides, **SERVO** has several processes for organizing the commands for motion, the commands for getting the current joint angles, the commands for operating the gripper, as well as the commands for getting the current touch state.

![Figure 4.8: Interface of SERVO Module](image)

### 4.4.3 Constructing the System from Behavioral Modules

The connection between different modules, in fact, is straightforward (See Figure 4.9). We just connect the output ports of one module to the input ports of another module, if the port descriptions of both are consistent.
Figure 4.9: Connection between Different Modules
The result of the whole system is totally distributed. There is no central control over these four transputers. Each of the transputers starts working regardless of the functioning of the others. If a behavioral module (on a transputer) stops working the rest of the system will still continue functioning. The theme behind this is: any individual module only affects the whole system locally.

There is a hierarchical coordination between different parts of the system. At the highest level ROBOT coordinates the operations carried out by lower level modules HAND and ARM through sensors (whenever the gripper touches something, the arm stops moving forward, instead, the hand starts turning downward, then the arm starts moving down towards ground); besides, the gripper is coordinated with the vision sensor or the range sensor (currently a signal from the terminal) to fulfill the task of releasing objects.

ARM would be the coordinator of the six joints if we put these joints into different transputers. We put them into one transputer because they communicate with one another through shared memory. We did not put ARM and JOINTS into one transputer, because the computations of these two modules are relatively independent, thus we can explore parallelism with little communication overload.

The operation of the gripper in the current system is very simple. In order to accomplish the gripper operation ROBOT can issue two commands, "open" and "close", to SERVO module which passes them to the robot controller.

There are two disadvantages of the current configuration:

- the communication with the robot controller becomes a bottle neck, since all the commands have to be serialized by the computer within the controller, and executed one by one;
• the controller provides only one kind of point to point movement, which causes the arm stops between segments of the movement if we decompose the whole motion into a sequence of "moveby" commands.

4.5 An Alternative Configuration

The disadvantages of our current system mentioned in the previous section are due to the fact that all the commands issued from a transputer to the controller are serialized by an interfacing computer within the controller (see Figure 4.3). This connection guarantees that the robot carries out the previous command before accepting the next one. However we want to get rid of this property in our system because of two reasons. Firstly we want the gripper opens and closes constantly during the arm is moving. Secondly we prefer that the arm can move continuously. In order to solve this problem, we can use the control cable of the master arm which are originally designed for sending out the analog signals of the joint's positions of the master arm during manual control. We can design a D/A convertor to convert the digital position signals from the transputer into analog signals, and an A/D convertor to convert the analog signals from the cable into digital signals. With the help of these two convertors, The transputer can directly communicate with the physical control part of the arm (see Figure 4.10 where the box in the middle is for A/D and D/A convertors. The cable from that box is plugged in to the manual slot of that controller). In this configuration, the interfacing transputer has two permanent processes. One is constantly sending out the desired joint angles and gripper operations at some rate to the D/A convertor and the other is constantly getting the current joint angles and touch state from the A/D convertor.

The advantages of this configuration are that we can get rid of the digital part of the
Figure 4.10: Alternative Configuration of Transputer and Controller
controller which forces the commands to be executed sequentially, and we can design our own distributed position control algorithm to achieve a better performance of motion.

Considering the efficiency of the line transmission between the arm and the transputer network, we can distribute the joint modules into two transputers since the amount of computation is increased in a joint module when we bypass the controller to design our own position control. The behavioral modules and their connection for this configuration is shown in Figure 4.11.

Figure 4.11: Alternative Configuration of Behavioral Modules
Chapter 5

Programming a Real-Time Multi-Sensory Robot System in Parallel C++

"Programs, of one sort or another, underlie everything that happens."
— Geoff Simons "Is Man A Robot?" John Wiley & Sons, Page 133

In this chapter, we first present Parallel C++, a parallel distributed object oriented language implemented on transputers, then illustrate how to use the language to program a real-time, multi-sensory robot system. Section 1 gives a brief introduction to Parallel C++; Section 2 discusses some useful constructs of parallel C++ for programming distributed real-time systems; Section 3 gives the design and implementation of the modules of our robot control system discussed in the previous chapter; Section 4 discusses the current performance and approaches for further improvement.
5.1 Brief Introduction to Parallel C++

C++ [Str86] is an upward compatible superset of C that provides object oriented facilities. Parallel C [Moc88] from Logical Systems is an extension of C which supports parallel and distributed processing on the transputer network. We have implemented Parallel C++ [ZQ89], a combination of Parallel C and C++, on the transputer network. Parallel C++ not only supports both object oriented methodology and parallel distributed programming techniques, but also introduces new concepts for modeling and constructing the complex systems. It is not only an expressive and effective language, but also a practical one for applications in a parallel and distributed environment [ZQ89].

In this section we give a brief overview of the main features of C++ and Parallel C, then introduce the programming language Parallel C++.

In addition to the facilities of C, C++ provides a flexible and efficient mechanism for defining new types — classes. A class is a type of a set of objects composed of two parts: private components and public components. An object is an instance of some class. The public components of an object are the only interface through which the object is manipulated. The class mechanism encourages data hiding and enhances modularity. The form of the class specification is:

```
class name {
    private components
    public:
        public components
}
```

where the private components can be data items and functions that are local to the objects of the class; the public components can be data items, various constructors for initializing an instance (object) of the class, and destructors for disposing of an object of that class. Besides,
there may be a number of functions or operators defined on the data items.

Parallel C from Logical Systems provides facilities to support the concurrency features of the transputer [Moc88]. The facilities are implemented as a set of library functions that allow the user to specify the definitions, communication and synchronization of concurrent processes.

Parallel C provides three new data types: `Process`, `Channel`, and `Semaphore` as well as the library functions for allocating the processes (`ProcAlloc`), for executing the parallel processes (`ProcRun`, `ProcPar` etc.), for process communication through channels (`ChanIn`, `ChanOut` etc.) and for process synchronization through semaphores.

The syntax for process definition is the same as that for functions except that the first parameter of a process function must be specified as a dummy variable of `Process` type (see the following examples). A process can be created by invoking function `ProcAlloc`. Communication among processes through channels is performed by invoking functions like `ChanIn` or `ChanOut`.

The implementation of Parallel C++ is quite simple. We compile the source files in C++ library to assembly codes of transputers which can be link to the user programs as a part of the concurrency library of Parallel C. We accomplish this by translating C++ codes into C codes via the C++ preprocessor and compiling the resultant C codes by the Parallel C compiler. We have changed some include files of parallel C to make them compatible with C++ preprocessor.

A user's program written in Parallel C++ is first preprocessed by the C++ preprocessor and then compiled by the Parallel C compiler. Finally the compiled codes are linked with both Parallel C library and C++ library (which is already compiled into the codes compatible to Parallel C).

This integration is clear and powerful. There is no syntactic conflicts nor semantic confusion. It is an upward compatible superset of both C++ and Parallel C. The following are some
examples of the use of this combination:

- Class variables can be used as the parameters of a (process) function.

- Class variables, as well as the member functions, operators and constructors of classes can be used in the body of a process.

- Any functions supplied by Parallel C, such as those for channel and process allocation, process creation and communication, can be called in the definitions of the member functions, constructors, and operators of a class.

- The data types supported by Parallel C, such as `Process` and `Channel`, can be used in the same way as any other data types of C++ in a class specification.

Note however that processes cannot be defined as member functions of a class. This restriction results from the inconsistency of the current version of C++ preprocessor and the Parallel C compiler.²

5.2 Constructs in Parallel C++

Two kinds of objects can be defined in Parallel C++: passive objects and active objects. The former is the model of data structures; the latter is the model of devices.

A passive object is composed of private data and public operations defined on the data. It is called passive object because it is initialized as a chunk of memory without control threads.

²Parallel C requires that the first parameter of a process definition be a dummy variable of `Process` type, but C++ preprocessor translates the member function of a class into a function with a special variable `this` as its first parameter. This inconsistency results in the awkward way to define active objects as we will see later. However a slight change of the Parallel C compiler or the C++ preprocessor will get rid of this problem. So the problem is not considered as a syntax conflict.
The interface for a passive object is the public functions or operators whose implementational details are transparent to the user of the object.

As an example of passive objects, the rotation matrix in the robot arm kinematics can be represented as a class, frame (see Figure 5.1). Here class frame is composed of three vectors for representing each axis of the frame. Also there are two ways of initializing the instances of that class: we can initialize an instance with three vectors, or simply declare an object of frame class without initializing any data. Besides, we defined one operator for multiplying a frame by another, which corresponds to the transformation of one frame into another. We can incrementally add more functions of these kinds into that class.

Class frame is easy to use (see Figure 5.2). The use is the same as the use of any other data types in the program. Here we just need to declare the frame objects, and to apply the defined functions or operations on them.

An active object, also called a concurrent object or a parallel object, consists of private data items including channels. It is called active object because it is initialized as a set of processes which actively communicate with each other or communicate with other objects, through channels or via shared memory. The interface or external behavior of an active object is defined by its communication convention, i.e., its port description, communication protocol etc. An active object can be considered as a module with a set of connections (see Figure 5.3). Any two active objects can be connected together as long as they have a consistent communication convention.

As an example, consider the simple producer and consumer problem. The definitions of the two communicative processes are shown in Figure 5.4. Process producer continuously produces the data, and process consumer continuously consumes the data. Two processes
A Behavioral Approach for Open Robot Systems

Figure 5.1: The Head Declaration of Class frame
communicate through a channel. These two processes can be defined as two classes of active objects, Proc and Cons as shown in Figure 5.5, corresponding to processes producer and consumer respectively. From the programs (see Figure 5.4 and 5.5) we can see that the use of the latter is easier and clearer than the use of the former. Active objects can be used to model any logical or physical devices which are independent, but communicating with other devices through designed ports. A system consisting of multiple devices can be modeled as a program with a large set of active objects devices; the communications between active objects corresponds to the communications between different devices.

Active objects are used here as the major constructs for programming a robot system with multiple sensors and manipulators. The language augmented with these constructs is powerful for representing any complex and distributed systems. In fact, it is straightforward to map Brooks’ AFSM [Bro89], Augmented Finite State Machine, into our language, which will be
The definition of processes

producer(Process *p, Channel *ch)
{
    int x;
    for(;;){
        x = proc();
        /* produce data x */
        ChanOutInt(ch,x);
        /* send out the data through the channel */
    }
}

consumer(Process *p, Channel *ch)
{
    int x;
    for (;;) {
        x = ChanInInt(ch);
        /* get data from the channel */
        cons(x);
        /* use data x */
    }
}

The use of processes

init()
{
    Process p1,p2;
    Channel ch;

    ch = ChanAlloc();
    /* initializing the soft channel */
    p1 = ProcAlloc(producer, stack, 1, ch);
    p2 = ProcAlloc(consumer, stack, 1, ch);
    /* create the two processes */
    ProcRun(p1); ProcRun(p2);
    /* running these processes in parallel */
}

Figure 5.4: Communicated Processes Producer and Consumer
A Behavioral Approach for Open Robot Systems

class Proc {  
    Channel *ch;  
public:  
    Proc(Channel *);  
}  

Proc::Proc(Channel *ch)  
{  
    Process *p;  
    p = ProcAlloc(producer, stack, 1, ch);  
    ProcRun(p);  
}  

class Cons {  
    Channel *ch;  
public:  
    Cons(Channel *);  
}  

Cons::Cons(Channel *ch)  
{  
    Process *p;  
    p = ProcAlloc(consumer, stack, 1, ch);  
    ProcRun(p);  
}  

init()  
{  
    Channel *ch;  
    
    ch = ChanAlloc();
    Proc(ch); Cons(ch);  
}  

Figure 5.5: Active Objects Producer and Consumer
5.3 Module Design

Module Design is the stage of implementing the desired behavioral specification of each module. Module design involves the details of the logical or physical devices related to individual module. A module can consist of a set of sub-modules. An atomic module is an active object with a set of cooperative processes or simply one process (such as the producer and consumer case). For real-time multi-sensory robot systems, there are two kinds of processes: permanent processes and task-related processes. Permanent processes deal with task-independent sensory processing and updating, which are very important for the reactive mechanisms. Task-related processes are usually created to perform a particular task. The coordination between different processes makes it possible to achieve several goals in parallel. The basic procedure for module design is to decompose the behavioral module into a set of active objects (or simply processes), as well as the coordination between these objects (processes). A behavioral chart (see Figure 5.6) helps for building this communicative process structure. The coding is straightforward — simply by implementing each box in the behavioral chart as an active object or a process. The wires

![Figure 5.6: A Behavioral Chart](image)
between them are communication channels.

In rest of this section, we describe the module design for each of the the modules in figure 4.4.

5.3.1 SERVO Module

SERVO module has five ports, one of them connected to the robot controller through RS232 (see Figure 5.7). SERVO module is performing the low level coordination between different commands through the robot controller. There are two kinds of commands: one is for organizing actions such as “moveby”, “open”, “close”, etc. and the other is for getting sensory inputs such as “manip”, “inp” etc. On the other hand, we can classify the commands into two categories, one is about the arm and the other is about the gripper.

SERVO module is decomposed into three communicative processes (see Figure 5.7). Process servo_process serializes all the commands through the hard link, using a set of multiplexing channels. Process gripper_process coordinates the commands between the gripper operations and the touch sensor. Process arm_process coordinates the commands of getting current joint movements from port1, then issuing them to servo_process, and getting current joint angles from servo_process, then sending them out through port2. The principle for setting the coordination is to guarantee that the action is always based on the current sensory information. All these three processes are permanent processes, as they are created once the SERVO module is created. The code of this module is straightforward from this chart, which is omitted here.

5.3.2 JOINT Module
The behavioral chart of JOINT module is shown in Figure 5.8. JOINT has six active objects as its sub-modules: SWING, SHOULDER, ELBOW, UPPER, WRIST and TOOL, modeling the six physical joints respectively; two shared data memories for the current and the desired position and orientation; and three parallel processes whose functions are as follows:

- **cur.pos** is a process repeatedly reading the current joint angles from port4, calculating the current position and orientation, restoring them in a shared data memory, and sending the result out through port2;

- **cur.mov** is a process repeatedly receiving the joint movements from the six active objects then sending them out through port3;

- **cur.goal** is a process repeatedly receiving the desired goal position or orientation from port1, and updating them in a shared data memory.
Figure 5.8: JOINT Module
Each joint has two ports, an input port for synchronization signals, and an output port for sending out the incremental amount for the joint movement. Each joint repeatedly performs the following tasks: receives the synchronization signal from \texttt{cur.pos} — which means that the current position has been updated, calculates the desired movement according to the current and the desired position and orientation in the shared memory, and sends the result out.

\textbf{JOINT} module involves two ways of process communication — message passing through channels and communicating via shared memory. Besides the explicit message passing shown in Figure 5.8, the current position and goal position are shared by the six joints and two processes: \texttt{cur.pos} and \texttt{cur.goal}.

Each of the active objects for the joints has its private data storing the current orientation frame of the joint, and two functions for updating the current orientation frame and for calculating the desired joint movement respectively. Each joint consists of two cooperative processes for the updating and calculating respectively. For example, class SHOULDER is defined as:

```cpp
class SHOULDER {
    frame F;

public:
    SHOULDER(Channel *, Channel *);
    void cal_Dif(double *);
    void cal_F();
    void ini_F();
};
```

where \texttt{F} is a kinematic frame of this joint, representing the current orientation of the joint; \texttt{cal.F} and \texttt{ini.F} are the functions for calculating and initializing the current frame respectively\footnote{The reason that we need to initialize the joint frame is that the JOINT module begins its function independently. We should guarantee that the frame is set so that the parallel processes will not get into trouble with undefined data, before the first set of joint angles comes.}; \texttt{cal.Dif} is the function for computing the incremental amount for the joint movement based
on the current goal, the current position and the current frame. We use iterative inverse
kinematics — Functional Joint Control [Poo88], rather than the closed form inverse kinematics
for this calculation because:

- by using FJC we can explore the potential parallelism. In this approach each joint cal-
culates the joint angles independently. By mapping the joint modules into several trans­
puters we can speed up the computation (for example, see Figure 4.11);

- FJC encourages modularity, for we can module each joint independently;

- many arms have no closed inverse kinematics solution or infinitely many of them (for a
redundant arm), while iterative inverse kinematics still works for these arms;

The algorithm for calculating the movement of the joint angles is similar to Poon’s [Poo88],
except that we set every step of the motion less than a certain amount (for example, fifteen
degrees). We did this for the following two purposes:

- making the motion more uniform. In Poon’s algorithm the motion at the beginning is fast
and slows down with time for a small adjusting. However, we prefer to have a continuous
uniform motion;

- making a better use of sensory information during the movement. Because each step is
small and the sensor information can affect each step, the change of the environment can
be recognized within the small step of motion.

We notice that we still can use Poon’s algorithm for each small segment along the trajectory.
However, in these case, the higher level modules have to set the goal position and orientation
too frequently, which would in fact slow down the system. Our algorithm for the calculation
is described in Appendix C. We have no fine trajectory for this arm and no path planner or explicit calls to the inverse kinematics. What we have is an infinite loop, similar to the servo loop, for end-effector position and orientation control. The joints just constantly adjust themselves to achieve the current goal.

The definitions of the two cooperative processes of SHOULDER, shoulder_process and process_cal_F_Sh, are shown in Figure 5.9. After receiving a synchronization signal, process_cal_F_Sh updates its current frame (by calling cal_F), and then sends the ready signal out. After receiving the ready signal, should_process calculates its desired movement (by calling cal_Dif which uses the algorithm we describe in Appendix C), and then sends it out.

The initialization of the module SHOULDER is done by creating and executing these two processes. The SHOULDER module and the corresponding codes are shown in Figure 5.10 and Figure 5.11 respectively. Other joints are designed and implemented similarly.

The initialization of the JOINT module is done by just initializing the six joints and three processes. The corresponding Parallel C++ codes are shown in Figure 5.12.

JOINT module is also a task-independent module. Even though the set of the current goal is task-related, the incremental movement of each joint (probably 0 if the goal has been achieved) is continuously calculated and sent out through the respective port, once the module is initialized.

The resultant behavior for these six joints is (refer to Figure 5.8). Each time after receiving current joint angles, process cur_pro calculates the current position and orientation of the end-effector using forward kinematics, which updates the shared memory, sends the result to port2 and sends a synchronization signal to each joint module. This signal triggers each joint to calculate the current frame, then to calculate its incremental movement. Process cur_mov
private:

shoulder_process(Process* p, SHOULDER sh, Channel* in, Channel* out)
{
    double an;
    for(;;)
    {
        ChanIntInt(in);
        /* waiting for ready signal */
        sh->cal_Dif(&an);
        /* calculating the desired movement */
        ChanOutDouble(out, an);
    }
}

process_cal_F_Sh(Process* p, SHOULDER sh, Channel* in, Channel* out)
{
    for(;;)
    {
        ChanInInt(in);
        /* waiting for signal */
        sh->cal_F();
        /* calculating the current frame */
        ChanOutInt(out, 1);
        /* sending out ready signal */
    }
}

Figure 5.9: The Parallel C++ Codes for the Two Processes of SHOULDER

collects these movements from six joint modules, and sends them to port3 to drive each joint
to move, so that the end-effector will be one step closer to the desired goal. The setting of the
goal position is asynchronous with the movement calculation. However, when the desired goal
is reset by ARM, the movement is automatically retargeted to this new goal. At any time, each
joint calculates the movement according to the current end-effector position and orientation,
as well as the current desired end-effector position and orientation.
Figure 5.10: SHOULDER Module

```c
#include <stdio.h>

#define MAX_BUFFER 100

int main() {
    int a = 5;
    int b = 10;
    int c = a + b;
    printf("The sum is: %d\n", c);
    return 0;
}
```

Figure 5.11: The Parallel C++ Codes for Module SHOULDER
/***********************************************************************************/
JOINT::JOINT(Channel *port1, Channel *port2,
               Channel *port3, Channel *port4)
{
    Channel *in[6], *out[6];
    Process *p1, *p2, *p3;

    for(i=0;i<6;i++){
        in[i] = ChanAlloc();
        out[i] = ChanAlloc();
    }

    /* initializing internal channels */

    p1 = ProcAlloc(cur_pos, stack_size, 3, in, port2, port4);
    p2 = ProcAlloc(cur_mov, stack_size, 2, out, port3);
    p3 = ProcAlloc(cur_goal, stack_size, 1, port1);

    SWING(in[0], out[0]);
    SHOULDER(in[1], out[1]);
    ELBOW(in[2], out[2]);
    UPPER(in[3], out[3]);
    WRISTFLEX(in[4], out[4]);
    TOOLS(in[5], out[5]);

    ProcRun(p1);
    ProcRun(p2);
    ProcRun(p3);
}
 /*******************************************************************************/

Figure 5.12: The Parallel C++ Codes for Module JOINT
5.3.3 ARM Module

ARM is on an intermediate layer, whose main function is to interpret the command from ROBOT module, and send the desired goal position or orientation of the end-effector to JOINT module. There are two kinds of commands from ROBOT: one is related to position and the other is related to orientation. All these commands are about continuous motion (e.g. "move_forward" command does not specify an absolute position; its semantics is to cause the arm moving forward) which can be stopped by using stop command.

Position Commands

Position commands include arm_forward, arm_backward, arm_up, arm_down and arm_turn (see Figure 5.13).

![Position Commands Diagram]

- **arm_forward** (arm_backward) — the arm continuously moves forward (backward) without changing the orientation of the hand. ARM responds this command by setting the desired position to the forward (backward) position on the current plane of shoulder.
and elbow.

\[ \text{position}_{\text{desired}} = (x_{\text{new}}, y_{\text{new}}, z_{\text{new}}) \]

\[ x_{\text{new}} = \frac{x}{\sqrt{x^2 + y^2}} \times \text{len}_{\text{forward}}(\text{len}_{\text{backward}}) \]

\[ y_{\text{new}} = \frac{y}{\sqrt{x^2 + y^2}} \times \text{len}_{\text{forward}}(\text{len}_{\text{backward}}) \]

where \( x \) and \( y \) are the current position coordinates.

- **arm_up** (arm_down) — the arm continuously moves up (down) without changing the orientation of the hand. ARM responds this command by setting the desired position to the highest (lowest) position on the current plane of shoulder and elbow, i.e.

\[ z_{\text{new}} = z_{\text{high}}(z_{\text{low}}) \]

- **arm_turn** — the swing continuously turns an angle \( \alpha \). ARM responds this command by setting both the desired position and orientation according to the forward kinematics\(^3\).

### Orientation Commands

Orientation commands include **hand_forward**, **hand_up**, **hand_down**, **hand_right**, **hand_left** and **hand_turn**. The first five commands are related to the approach vector of the hand — \( a \); the last command is related to the normal vector of the hand — \( n \) (see Figure 5.14).

- **hand_forward**: the hand continuously turns to the forward position. ARM responds this command by setting \( a \) to \((x, y, 0)\) where \( x \) and \( y \) are the current position coordinates.

---

\(^3\)Readers may ask: Why did we not simply send a command which issues the swing to move an angle, instead of doing so much computation here? The answer is: we have to set the goal to new position and orientation, otherwise, the stupid JOINT module will try to move back to achieve its “goal”, so that the arm can’t go anywhere.
- **hand_up(hand_down)**: the hand continuously turns to up (down) position. ARM responds this command by setting \( \mathbf{a} \) to \((0,0,1)\) \((0,0,-1)\).

- **hand_left(hand_down)**: the hand continuously turns left (right). This command is carried out by setting \( \mathbf{a} \) to \((-y,x,0)\) \((y,-x,0)\) where \(x\) and \(y\) are the current position coordinates.

- **hand_turn**: the tool continuously turns an angle \( \alpha \). ARM responds this command by setting \( \mathbf{n} \) to new goal, \((n'_x,n'_y,n'_z)\):

\[
\begin{align*}
    n'_x &= n_x \cos(\alpha) + n_x \sin(\alpha) \\
    n'_y &= n_y \cos(\alpha) + n_y \sin(\alpha) \\
    n'_z &= n_z \cos(\alpha) + n_z \sin(\alpha)
\end{align*}
\]

![Figure 5.14: Orientation Commands](image)
Stop Command

All the commands for the arm are about continuous movement as we described previously. However we can stop the current movement by setting a new goal before the current goal is totally fulfilled. The command stop is only one of the examples of this idea. ARM responds this command by setting the goal to the current position and orientation of the end-effector. It is obvious that the arm stops moving, since the “goal” has been “achieved”.

ARM is implemented as an active composed of two permanent processes: processes command and cur_pos. Process command constantly gets the commands from ROBOT, calculates the desired goal, and sends the goal to JOINT. Process cur_pos continuously gets the current position and orientation from JOINT module, and updates old one in the current data memory. These two processes are running in parallel and communicating through the shared memory: cur_pos updates the current position and orientation which command uses.

5.3.4 ROBOT Module

In the design of the ROBOT module, we are facing the problems of how to encode the task into a set of cooperated processes. We have described the task and ports of ROBOT module in Section 4 of the last chapter.

ROBOT is implemented as an active object consisting of two permanent processes, cur_touch and cur_position, respectively for getting current touch state and current position state constantly. Besides, there are some task-related procedures running in parallel with and communicating with the permanent processes for getting the information about the environment. Currently, we have two procedures, findObject and releaseObject, for detecting objects releasing the objects respectively.
findObject task can be decomposed into a set of sequential processes (see Figure 5.15 where bond arrows represent the control flow) — hand_forward, search_process, backward_process and turning_process. At the beginning, the hand directs forward, which makes the right direction for search; search_process is composed of two parallel actions: arm moves forward, and gripper opens and closes constantly. This process stops when the arm has arrived the forward position or the gripper touches an object when it closes. If the gripper touches an object, the arm stops moving and the gripper stops opening, the findObject process finishes; otherwise, the arm moves back and turns a small degree and begins to search along the new direction.

releaseObject task (shown in Figure 5.16 where the bond arrows represent the control flow) begins with the hand directs downward. Afterwards, two processes runs in parallel: arm moves down and checks that whether there is a signal from the terminal which indicates that it is the right time to release the object. When such a signal is received, the gripper opens, so that the object is released.
5.4 Current Performance and Further Improvement

After we down-load each module to the transputer network connected as shown in Figure 4.4, modules start working independently and communicating with each other through the channels. The initial position of each joint is set to zero, then the hand starts turning forward and the arm moves forward with the gripper opening and closing at times (see Figure 5.17). If there is no object along this line, the arm comes back and turns a small degree and repeats the task (see Figure 5.18). If the gripper touches something, the arm stops moving forward immediately and the hand starts turning downward (see Figure 5.19). Afterwards, the arm moves down until a signal of touching ground comes from the terminal, which immediately triggers the gripper to open (see Figure 5.20).

From the experiment, we see that the sensory feedback works extremely well. The system
is quite successful in the aspect of reaction to the environment. We believe that we can obtain this reactivity for the robots with more sensors, as long as we distribute the computation appropriately and arrange the communications in relatively short paths.

The problem with the current system is that the motion is not smooth. Since we use FJC to get the iterative movement of each joint, the controller cannot make the motion continuously (without stop) between the two segments of the movement. One way to solve the problem is re-write the controller. However, this still cannot solve the problem of sequential execution of the commands (the operation of the gripper and motion of the joints cannot be executed in parallel). We will take the second approach — using the master arm’s control cable to send continuous signals to the manipulator which was discussed in the previous chapter. Under this new configuration, hopefully, we will get a better performance.

Even though our system is designed for multiple sensors, we now only have joint angle sensor and one-switch touch sensor. We are planning to install more sensors so that the robot can perform more interesting and realistic work. Tasks like “cut off branches” will be designed for the robots with suitable proximity sensors or more complex touch sensors.
Figure 5.17: Phase 1: Robot Begin to Search for Trees
A Behavioral Approach for Open Robot Systems

Figure 5.18: Phase 2: No Object Found Along that Direction
Figure 5.19: Phase 3: Robot Begins to Search in Another Direction
begin to release the object

hand down

Figure 5.20: Phase 4: Robot Grasps a Tree and Puts it Down
Chapter 6

Conclusions and Further Research

Only experiments with real Creatures in real worlds can answer the natural doubts about our approach. Time will tell.
— Rodney Brooks “Intelligence Without Representation”

6.1 Conclusions

It has been shown to be possible to control a robot system by means of a parallel distributed object oriented language, and to implement that control on a transputer–based distributed computing environment.

We proposed a structural decomposition, decomposing a system into a structural hierarchy according to the physical or logical components of the system, in addition to a behavioral hierarchy (as described by Brooks). The advantages of two–dimensional, rather than one dimensional system design and development are:

- Similar to the idea of structural programming, by structural decomposition, one can design a system with the description of behavior from coarse to fine. Such a decomposition is extremely important from a system design and development point of view, especially when
the structure of the system becomes complicated.

- With structural decomposition, one can represent different layers of coordination. The system with this design can be more robust. Any module only affects the system locally. Even though higher level coordination may stop working, lower coordination can still keep functioning.

We implemented this two-dimensional structure using a parallel distributed object oriented language — Parallel C++. It has been shown that an active object (parallel object) can be used for modeling various kinds of logical and physical devices. There is no distinction between software and hardware design at this point. The function of an active object in a complex system is like a VLSI chip in a large electronic circuit.

We further implemented a prototype of such a system in a transputer-based computing environment. It has been shown that such an environment is suitable for the development real-time, multi-sensory and multi-manipulator robot systems.

We used iterative inverse-kinematics in the form of Functional Joint Control. The behavior of the arm has these properties:

- The motion of the arm is not described as a point-to-point motion, rather as a continuous movement which can be stopped by sensory information at any time.

- There is no fine trajectory planner. The arm is set to always achieve the current goal as far as it can. For example, arm_forward does not necessarily moves along a straight line, but moves as far as it can and can be stopped by a sensory input at any time (for example, seeing an obstacle).

Functional Joint Control can describe such behaviors directly and clearly.
the coordination between the task decomposition and the built-in reaction to the unpredicted situations can be represented and formalized. This formalization is intended for the exploration of the underlying mechanisms of behaviors in various "living" systems. It provides a theoretical framework for synthetic behaviors.

The languages which are mostly related to the framework of Concurrent Constraint Logic are FCP and STRAND. FCP, Flat Concurrent Prolog developed by Shapiro [Sha87] is an expressive language for parallel processing. However, up to now there has been no implementation in a transputer-based environment. STRAND is a parallel logic programming language with new concepts in parallel programming [FT89], supported for various parallel environments including transputers and hypercube machines. We would like to use STRAND as the basic language for implementing and testing our theoretical framework.

There is still a long way to go to achieve our goal. However, the dream of yesterday, is the hope of today, and the reality of tomorrow.
the coordination between the task decomposition and the built-in reaction to the unpredicted situations can be represented and formalized. This formalization is intended for the exploration of the underlying mechanisms of behaviors in various “living” systems. It provides a theoretical framework for synthetic behaviors.

The languages which are mostly related to the framework of Concurrent Constraint Logic are FCP and STRAND. FCP, Flat Concurrent Prolog developed by Shapiro [Sha87] is an expressive language for parallel processing. However, up to now there has been no implementation in a transputer-based environment. STRAND is a parallel logic programming language with new concepts in parallel programming [FT89], supported for various parallel environments including transputers and hypercube machines. We would like to use STRAND as the basic language for implementing and testing our theoretical framework.

There is still a long way to go to achieve our goal. However, the dream of yesterday, is the hope of today, and the reality of tomorrow.
Appendix A

FCP++ — An Object Oriented Flat Concurrent Prolog

A.1 Brief Introduction to FCP++

FCP— Flat Concurrent Prolog is a logic programming language which replaces the backtracking search mechanism of Prolog with communicating concurrent processes [KM88]. It can represent incomplete messages, unification, direct broadcasting and concurrency synchronization, with the declarative semantics by a sound theorem prover [Sha87]. It is a good candidate as the language for Open Systems [KM88].

FCP++, a concurrent logic object oriented language based on FCP was designed and implemented [Zha88]. FCP++ inherits all the capabilities of concurrent logic programming techniques supported by FCP and provides a set of new and powerful facilities for building object-oriented systems. The facilities include

1) class definition
2) function for making instance
3) function for halting instance
4) method definition
5) message passing by commands

6) message passing by data through channels

The programs in FCP++ can be translated into FCP by a preprocessor we implemented in Quintus Prolog. Following is the syntax for FCP++

Italics is used to indicate a part of a template which users are to fill in. The items of upper case italic are to be filled in with variables and the items of lower case italics are to be filled in with constants, and identifiers starting with a capital letter followed by some lower case letters can be filled in with any kind of terms.

    class definition
    class(name,[var1, var2, ..., varn])

    function for making instance
    className:make([Value1, Value2, ..., Valuen], INSTANCE<:>state(Id))

    function for halting instance
    className:halt([Value1, Value2, ..., Valuen])
    className:halt([Value1, Value2, ..., Valuen]) → Body

    method definition
    className::MessagePattern
    className::MessagePattern → Body
message passing through instance

\textit{INSTANCE:MessagePattern}

message passing through communication channel

$\text{var}>\text{Message}$

$\text{var}<:\text{Message}$

getting object identifier

\textit{INSTANCE}<:>:ID

**A.2 FCP++ Program for Figure 3.4**

Figure A.1 is the skeleton of the program for Figure 3.4. We can see that in this program, \textit{SENSOR0} continuously outputs the current conditions; \textit{PLANNER} continuously generates partial plans; and \textit{EXECUTOR} continuously executes such plans and causes the current conditions approaching to the goal conditions. All these processes are running in parallel. We did not write down the clauses for \textit{see\_cond}, \textit{planning} and \textit{execute}, which depend on the details of the system.

**A.3 FCP++ Program for Figure 3.6**

The program in Figure A.2 is an implementation of the process structure shown in Figure 3.6. \texttt{avoidObstacles} is one of the permanent processes which are created whenever the object \texttt{robot} is created. \texttt{moveto} is the task designed for this robot. \texttt{avoidObstacles} and \texttt{moveto} are running in parallel. If there is no task running, this robot avoids any objects. If task
achieve([Goal|NextGoal]) :-
    sensor0(Goal, Conditions),
    planner(Conditions, Goal, Plans),
    executor(Plans),
    achieve(NextGoal).

sensor0(Goal, [done]) :- see_cond(Goal, done).
% current condition is the goal itself

sensor0(Goal, [Cond|Next]) :-
    see_cond(Goal, Cond),
    sensor0(Next).

planner([done], Goal, []).
% if the goal is achieved, no plan is generated

planner([Cond|NextCond], Goal, [Plan|NextPlan]) :-
    planning(Cond, Goal, Plan),
    planner(NextCond, Goal, NextPlan).
% generate a partial plan according to the current condition
% and the Goal condition

executor([]).
% if there is no plan, nothing is executed.

executor([Plan|Next]) :-
    execute(Plan),
    executor(Next).
% execute the partial plan.
% during the execution, other sensory information
% can be also used.

Figure A.1: FCP++ Program for Figure 3.4
class(robot, [obstacleSignals, ......]).
% robot is a class with one of the internal states as channel for
% signals of detecting the current obstacles

robot::init :-
    avoidObstacles(obstacleSignals).
% the built-in mechanism which makes the robot always
% avoids the obstacles and sends out the signal about
% the obstacle

robot::moveto(Object) :-
    see(Object, Directions),
    move(Directions, obstacleSignals).
% one of the task for this robot is to move to the
% described object. Certainly it also should
% notice the obstacle signals when fulfilling this task

see(Object, []) :- sensorSee(Object, done).
% if already moved to the object, no direction sent out

see(Object, [Dire|NextDires]) :-
    sensorSee(Object, Dire),
    see(Object, NextDires).
% otherwise one direction sent out

move([], _).

move([Dire|NextDires], [Obs|NextObs]) :-
    moveby(Dire, Obs),
    move(NextDires, NextObs).

moveby(Dire, Obs) :- move_avoid(Dire, Obs).
% make one step movement, of course shouldn't
% move to the obstacle

Figure A.2: FCP++ Program for Figure 3.6
moveto is running, the robot moves to the specified object, and avoids the others.
Appendix B

Parallel C++ for AFSM

Brooks [Bro89] described semantics for the AFSM — Augmented Finite State Machine. However, his representation only generates a simulated parallel program. The user has to explicitly write down the stop points of each process (using event-dispatch). We now show that we can write a more clear and truly parallel program for AFSM using Parallel C++.

The definition of network of AFSM consists of two parts: the definitions of the AFSMs and the definitions for the wires which connect different AFSMs into a system.

The AFSM mainly consists of a finite state machine and a set of input registers, output channels and internal variables. This machine can be reset by an outside signal at any time.

It is obvious that an active object fits perfectly to an AFSM. We define the head file of an AFSM as in Figure B.1.

The finite state machine corresponds to the case statement in C (see Figure B.2).

There are two permanent processes for resetting the machine and restoring input signals to registers (see Figure B.3).

Finally, the initialization of an AFSM is accomplished by initializing an active object of class AFSM (see Figure B.4).
class AFSM {
    int State; /* the state of the finite state machine */
    int R[N]; /* registers */
    Channel *out[M]; /* output channels */
public:
    void AFSM(Channel *[N], Channel *[M], Channel *);
    void resetState(){State = -1};
    void initStateO {State = 0};
    void setState(int NewState){ if (State == -1) State = 0;
        else State = NewState;};
    void setR(int i, int val){R[i] = val;}
    void machine();
}

Figure B.1: The Head File for AFSM

void AFSM::machine()
{
    State = 0;
    for(;;) {
        switch State of
        case 0: ...
            /* any expressions including ChanOutInt */
            /* any variables in private part can be used */
            
            setState(NewState); break;
        case 1: ...
            setState(Newstate); break;
            .......
        }
    }

Figure B.2: Finite State Machine in C
int reset_process(Process *p, AFSM *A, Channel *reset)
{
    for(;;) {
        ChanInInt(reset);
        A->resetState();
    }
}

int input_process(Process *p, AFSM *A, Channel *input[N])
{
    int index, value;

    for(;;) {
        index = ProcAltList(input);
        value = ChanInInt(input[index]);
        A->setR(index, value);
    }
}

Figure B.3: Processes for Resetting the Machine and Setting Registers
AFSM::AFSM(Channel *input[N], Channel *output[M], Channel *reset)
{
    Process p1, p2;
    int i;

    for(i=0; i<M; i++)
    {
        outCi] = ChanAlloc();
        out[i] = output[i];
    }
    /* initialize out channels */

    p1 = ProcAlloc(reset_process, stack_size, 2, this, reset);
    p2 = ProcAlloc(input_process, stack_size, 2, this, input);

    ProcRun(p1); ProcRun(p2);
    this -> machine();
}

Figure B.4: Initialization of AFSM
suppress(Process *p, Channel *in1, Channel *in2, Channel *out)
{
    int index, value;

    for(;;) {
        index = ProcAlt(in1,in2,0);
        if (index == 0) {
            value = ChanInInt(in1);
            ChanOutInt(out, value);
            ChanInIntTime(in2, time);
        } else {
            value = ChanInInt(in2);
            ChanOutInt(out, value);
        }
    }
}

Figure B.5: Suppress Function in Parallel C++

The AFSM consists of two permanent processes and a finite state machine which is also permanent.

Similarly, we can have some process definitions corresponding to the definitions for the wires. There are three kinds of processes: suppress, inhibit and multiplex. The parameters for suppress process include two input channels and one output channel. One of the input channel can suppress the other for a period of time (see Figure B.5). The parameters for inhibit is the same as that for suppress. The parameters for multiplex process include one input channel and a set of output channels. The input from the input channel will be sent out to all the output channels (see Figure B.6). The function of inhibit is similar to that of suppress, except that no output comes out during the suppressing period. Each of the wire processes is a permanent process. The whole system is constructed by initializing all the channels, active objects (AFSM) and processes (wires connections) (see Figure B.7). Each box is an AFSM,
multiplex(Process *p, Channel *in, Channel *out[N])
{
    for(;;) {
        val = ChanInInt(in);
        for(i=0; i<N; i++) ChanOutInt(out[i], val);
    }
}

Figure B.6: Multiplex Function in Parallel C++

Figure B.7: AFSM system in Parallel C++
each circle is a connection of wires. S stands for suppress; M stands for multiplex.
Appendix C

Functional Joint Control

Functional Joint Control proposed by Poon [Poo88] is very powerful for a large class of manipulators which are functionally decomposable.

In this thesis, we use the idea of FJC for the kinematics loop of the joint movement. We set the step of the movement almost equal during the whole motion, instead of using adaptive gain factors in Poon's algorithm. The reason of this change has been stated in Chapter 5.

C.1 Projection Technique for Joints Controlling Orientation

Suppose the function of joint \( i \) is to control the orientation of a vector, \( v \). The target orientation is \( v_d \), and the current orientation is \( v \) (see Figure C.1). For joint \( i \), the current kinematics frame is \( T_i \), with axes \( X_i, Y_i, Z_i \). \( v_d \) and \( v \) can be projected onto the \( X - Y \) plane of \( T_i \). Let the projections be \( v_{pd} \) and \( v_p \).

\[
\begin{align*}
v_{px_i} &= v.X_i \\
v_{py_i} &= v.Y_i
\end{align*}
\]
Figure C.1: FJC Projection Technique

All that joint $i$ can do to help align $v$ with $v_d$ is to align $v_p$ with $v_{pd}$. The angle between $v_p$ and $v_{pd}$, $\Delta \theta_i$, is given by

$$\Delta \theta_i = \arctan(v_{pdyi}/v_{pdxi}) - \arctan(v_{pyi}/v_{pxi}),$$

where $v_{pxi}$ stands for the x-component of $v_p$ on the frame of joint $i$, etc. We set a step of movement at most fifteen degrees, so that if

$$\Delta \theta_i > 15, a_i = 15$$

otherwise

$$a_i = \Delta \theta_i$$

where $a_i$ is the actual step movement.

In Excalibur, swing and shoulder control the orientation of $p$; upper and wrist control the orientation of $a$; tool controls the orientation of $n$. 
C.2 FJC for Joints Controlling Length

The function of some joints is to alter a certain length, $L$ (see Figure C.2).

\[ L^2 = L_{i-1}^2 + L_i^2 - 2L_{i-1}L_i \cos(\theta_i), \]

and

\[ L_d^2 = L_{i-1}^2 + L_i^2 - 2L_{i-1}L_i \cos(\theta_d) \]

where $L_i$ is the length of link $i$, $L$ is the current length, $L_d$ is the desired length, and $\theta_i$ is angle between link $i$ and $i-1$.

\[ \Delta \theta_i = \theta_d - \theta_i \]

Similarly, if

\[ \Delta \theta_i > 15, a_i = 15 \]

otherwise

\[ a_i = \Delta \theta_i \]

where $a_i$ is the actual step movement.

In Excalibur, **elbow** is the joint for controlling the length.
Bibliography


A Behavioral Approach for Open Robot Systems


