

Basic Set of Behaviours for Programming Assembly Robots

Giovanni Cosimo Pettinaro



Ph.D.
University of Edinburgh
1996



Abstract

We know from the well established Church-Turing thesis that any computer programming language needs just a limited set of commands in order to perform any computable process. However, programming in these terms is so very inconvenient that a larger set of machine codes need to be introduced and on top of these higher programming languages are erected.

In Assembly Robotics we could theoretically formulate any assembly task, in terms of moves. Nevertheless, it is as tedious and error prone to program assemblies at this low level as it would be to program a computer by using just Turing Machine commands.

An interesting survey carried out in the beginning of the nineties showed that the most common assembly operations in manufacturing industry cluster in just seven classes. Since the research conducted in this thesis is developed within the behaviour-based assembly paradigm which views every assembly task as the external manifestation of the execution of a behavioural module, we wonder whether there exists a limited and ergonomical set of elementary modules with which to program at least 80% of the most common operations.

In order to investigate such a problem, we set a project in which, taking into account the statistics of the aforementioned survey, we analyze the experimental behavioural decomposition of three significant assembly tasks (two similar benchmarks, the STRASS assembly, and a family of torches). From these three we establish a basic set of such modules.

The three test assemblies with which we ran the experiments can not possibly exhaust all the manufacturing assembly tasks occurring in industry, nor can the results gathered or the speculations made represent a theoretical proof of the existence of the basic set. They simply show that it is possible to formulate different assembly tasks in terms of a small set of about 10 modules, which may be regarded as an embryo of a basic set of elementary modules.

Comparing this set with Kondoleon's tasks and with Balch's general-purpose robot routines, we observed that ours was general enough to represent 80% of the most common manufacturing assembly tasks and ergonomical enough to be easily used by human operators or automatic planners. A final discussion shows that it would be possible to base an assembly programming language on this kind of set of basic behavioural modules.

Acknowledgements

There is a large number of people which I am greatly indebted to for their help and encouragement during the past four years that I spent here at the Department of Artificial Intelligence. These few lines cannot fully acknowledge the innumerable ways in which I have benefitted from the company of my fellow students and the staff. The following paragraphs can just try to recognize the most tangible aspects of this support.

First of all, I would like to thank my supervisor Chris Malcolm for his enlightening help first during the development of the project and then during the writing of this thesis.

I wish then to thank David Wyse and the rest of the staff of the mechanical workshop (Hugh Cameron, Neil Wood, and Douglas Howie) for having designed and built the two electric grippers and the electric turntable used in my research. They were faced with many frantic requests for help, and this was always offered promptly and pleasantly despite their enormous workload. On this regard I am also in great debt with Tom Alexander and Sandy Colquhoun who designed and developed all the interfacing circuits needed in my project.

A special acknowledgement goes to Edvaldo Marques Bispo, Dìbio Leandro Borges, Edgar Ramirez-Dominguez, Taehee Kim, and David Wren with whom I shared the ups and downs of the student life first in my MSc course and then in my PhD. I am in great personal debt with them and I want to thank them all for having been my best friends.

I wish then to thank my parents and my brother Luigi for their moral support which helped me to go through the good and bad times of this four years.

Finally, but not last, I would like to thank the two institutions which shared the burden to support me financially during my PhD: the Commission of the European Communities for the first two years and the University of Milan (Università degli Studi di Milano) for the last one.

Declaration

I hereby declare that I composed this thesis entirely myself and that it describes my own research.

Giovanni Cosimo Pettinaro
Edinburgh
August 17, 1996

Contents

Abstract	ii
Acknowledgements	iii
Declaration	iv
List of Figures	xii
1 Introduction	1
1.1 Socio-economic Motivations	2
1.2 Topic Problem	5
1.3 Thesis Claim	8
1.4 Thesis Layout	9
1.5 Summary	11
2 Literature Review	13
2.1 Robot Programming	13
2.1.1 Classification	14
2.1.2 Joint-Level Programming	16
2.1.3 Manipulator-Level Programming	17
2.1.4 Object-Level Programming	18
2.1.5 Task-Level Programming	20
2.2 System Architectures	21
2.2.1 Classical Approach	21
2.2.2 Behaviour-Based Approach	24

2.2.2.1	Handey	29
2.2.2.2	Subsumption Architecture	33
2.2.2.3	Behaviour-Based Assembly	38
2.3	Behavioural Decomposition	45
2.3.1	Assembly Operations	47
2.3.2	Balch's General-Purpose Modules	53
2.3.3	Related Research	56
2.4	Summary	61
3	Definition of the Project	65
3.1	Discussion of the Problem	66
3.2	Project	68
3.2.1	Project Description	69
3.2.1.1	Basic Default work-cell Description	70
3.2.2	Aims and Objectives	71
3.2.3	Choice of the Experiments	73
3.2.4	Extra Hardware Required	78
3.3	Summary	81
4	Behaviours	84
4.1	Behaviours as Behavioural Modules	84
4.1.1	Graphic Representation of Behavioural Modules	89
4.1.2	Composition of Behavioural Modules	91
4.2	Hierarchy of Behavioural Modules	95
4.2.1	Hierarchical Level of the Basic Behavioural Modules	96
4.3	Generality of Behaviours	98
4.4	Summary	99
5	Project Experiments	101
5.1	Guarded Motion	102
5.1.1	Guarded Motion Analysis	103
5.1.2	Guarded Motion Implementation	107

5.1.2.1	VAL II Implementation of Guarded Move	109
5.1.3	Guarded Motion Summary	114
5.2	Benchmark Assembly Family	115
5.2.1	Peg in Hole	116
5.2.1.1	Strategies to find a Hole	122
5.2.1.2	Strategies to insert a peg	128
5.2.2	Partial Benchmark Assembly	136
5.2.2.1	Pick-Up Module	139
5.2.2.2	Stack Module	143
5.2.2.3	Partial Benchmark Assembly Experiments	144
5.2.3	Full Benchmark Assembly	147
5.2.3.1	Full Benchmark Assembly Experiments	152
5.2.4	Benchmark Assembly Family Summary	154
5.3	STRASS Assembly	156
5.3.1	STRASS Analysis	157
5.3.2	STRASS Retaining and Snapfit Implementations	160
5.3.3	STRASS Assembly Experiments	163
5.3.4	STRASS Assembly Summary	164
5.4	Torch Assembly Family	165
5.4.1	screw Module	168
5.4.1.1	Strategy for Screwing	169
5.4.1.2	Screwing Experiments	172
5.4.1.3	Summary of Screwing	172
5.4.2	Torch Assembly Plan	173
5.4.3	Torch Assembly Family Experiments	182
5.4.3.1	Big Torch Kit	182
5.4.3.2	Medium Torch kit	183
5.4.3.3	Smallest Torch kit	184
5.4.4	Torch Assembly Family Summary	184
5.5	Summary	185

6	Analysis of the Results	189
6.1	Existence of the Basic Set	190
6.2	Synthesis of the Experimental Results	192
6.2.1	Development of a Guarded Motion	193
6.2.2	One Assembly Program for Similar Assemblies	194
6.2.3	Synthesis of our Basic Set	195
6.2.3.1	peg-in-hole as a Basic Module	195
6.2.3.2	pick-up as a Basic Module	197
6.2.3.3	stack as a Basic Module	198
6.2.3.4	grasp and ungrasp as Basic Modules	199
6.2.3.5	insert-peg-with-spring-retainer and snapfit as Basic Modules	200
6.2.3.6	screw as a Basic Module	201
6.2.3.7	flip as a Basic Module	202
6.2.4	Basic Set	203
6.2.4.1	Comparison with Kondoleon's Set	206
6.2.4.2	Comparison with Balch's General Purpose Modules	208
6.3	Behavioural Robot Language	212
6.4	Summary	218
7	Conclusions and Further Work	220
7.1	Summary of the Results	220
7.2	Scientific Contributions	222
7.3	Further Work	224
7.4	Conclusions	226
A	Adept Characteristics	228
B	Robot Electric Gripper	230
C	Two-Handed Robot System	233
C.1	Robot/Two-Grippers Subsystems	233
C.2	Robot/Left-Wrist Subsystem	236

Index	240
Bibliography	243

List of Figures

2.1	Curves describing the Planner-Agent System Complexity.	25
2.2	General Solution for drawing Triangles on a Computer Screen.	26
2.3	Solution in Navigational Terms vs. Solution in Cartesian Terms.	27
2.4	Brooks' Approach.	34
2.5	Levels of Competence in the subsumption Architecture.	36
2.6	Brooks' Finite State Machine.	36
2.7	Behaviour-Based System example.	40
2.8	Basic Manufacturing Assembly Operations Identified by Kondoleon in 1976.	47
2.9	Qualitative Results of the Statistics emerged from Kondoleon's Survey. .	49
2.10	Assembly Tasks Percentage.	53
3.1	Robot System used for the project.	70
3.2	Benchmark Family Assembly.	74
3.3	Benchmark Assembly.	75
3.4	Strass Assembly.	76
3.5	Torch Family Assembly.	77
3.6	Two-Handed Robot System Diagram.	79
3.7	Gripper equipped with Kim's Piezo-Film Sensor.	81
4.1	General Diagram of a Behavioural Module.	90
4.2	Example of a Behavioural Module Units Location.	91
4.3	Behavioural Module Basic Operations.	95
5.1	Guarded Move Diagram.	105

5.2	Example of Straight Line Motion impossible to be carried out.	107
5.3	Piezo-film bendings and correspondent operations.	109
5.4	Mean Distance covered because of Hardware and Software Delays. . . .	111
5.5	Benchmark Family Parts.	115
5.6	Peg in Hole Mating Stages.	117
5.7	Peg in Hole Diagram.	121
5.8	Peg on Hole.	123
5.9	Example of Chamferless Peg Tilting because of a Contact.	125
5.10	Zigzag and Spiral Searches Paths.	125
5.11	Trajectory followed during the Search which wrongly acknowledges a Hole.	127
5.12	Different Strategies for solving the Peg Insertion Problem for a Cham- fered Peg, but a similar Diagram applies for a Chamferless one.	130
5.13	Diagram of the Stuck Situation.	133
5.14	Diagram of Shifting Directions.	133
5.15	Metal and Wooden Partial Benchmarks.	136
5.16	Peg Shaft Length Determination Technique.	138
5.17	Partial Benchmark Assembly Diagram.	139
5.18	Pick Up Diagram.	142
5.19	Stack Diagram.	143
5.20	Full Benchmark Assembly.	147
5.21	Procedure to Compute the Tilt Angle.	149
5.22	Full Benchmark Assembly Diagram.	151
5.23	Two-dimensional Diagram of the STRASS Parts.	158
5.24	Two-dimensional Diagram of the Insertion with the Retainer.	160
5.25	Diagram of Insertion with Spring Retainer.	161
5.26	Diagram of Snapfit.	162
5.27	Torch assembly Kit.	166
5.28	Diagram for describing a Torch Assembly.	167
5.29	Screw Diagram.	171
5.30	Torch Subassemblies.	174
5.31	Pick-and-Place Operation Diagram.	174

5.32	Diagram for both Subassemblies.	176
5.33	Screw Fastening of the Two Torch Subassemblies.	178
5.34	Behavioural Decomposition of the Torch Assembly.	179
6.1	Proposed Formats for our Composition Operations.	216
A.1	AdeptRobot Joints.	229
B.1	DAI Electric Gripper.	230
C.1	Robot/Two-Grippers Sub-Systems.	234
C.2	Fingers' Gap Reading Subsystem.	236
C.3	Left Wrist Driving Subsystem.	237
C.4	Example of rotation anti-clockwise.	237
C.5	Wrist Potentiometer Reading Subsystem.	238
C.6	Grippers and wrist driving sub-system.	238
C.7	Potentiometers' reading sub-system.	239
C.8	Two-Handed System Architecture.	239

Chapter 1

Introduction

We are all aware that most of our everyday life is nowadays greatly affected by products which are mainly manufactured by industrial processes. At the beginning of this century science fiction depicted a futuristic world where human labour was completely replaced by automata capable of tirelessly performing industrial work. Today, we are still very far away from such a picture. Indeed the kind of automaton we came out with is not so general purpose as the one predicted by the fiction. It was named *robot* after Karel Čapek's satirical drama "*Rossum's Universal Robots*" performed in 1921. The name *robot* was actually borrowed from the Czech word *robota* which literally means "work". Such a play initiated a trend in science fiction literature which depicted these automata as androids incredibly strong and insensitive to pain. However, a robot is still up to now very different, and in many ways very limited, compared with such a general purpose anthropomorphic automaton. As far as industry is concerned, the Robot Institute of America gives the following definition:

Definition 1.1 (Robots) *A robot is any reprogrammable multi-functional manipulator designed to move materials, parts, tools, or specialized devices, through variable programmed motions for the performance of a variety of tasks.*

Thus, an industrial robot has to be thought of as a reprogrammable general-purpose computer-controlled manipulator which can move objects and assemble parts. Such a definition, because of its generality, applies for a wide range of cases from extraction of raw material to manufacturing of end products. Our investigation concentrates

basically on those concerning manufacturing industry.

Robots have so far failed to reach the kind of generality depicted in Čapek's play because of the many problems still to be solved. In this respect, the problem of finding a general and easy way to program them to accomplish industrial tasks is still one of the hardest. Today there are many ways available to solve it, but none is very user-friendly. What we are going to explore in this thesis is how to make robots perform work using means closer to humans than to a machine.

A great deal of money and research efforts has been dedicated for many years to investigate the many issues raised by the development and implementation of robots, and the scientific field which studies them takes the name of robotics. But why is it so important for our society? What we aim in this chapter is to show the economic reasons behind our interest in it and to present a brief synthesis of the current state-of-the-art of the research in robotics. We conclude the chapter presenting the particular problem we are going to tackle and the claim we are going to make.

1.1 Socio-economic Motivations

With this section we start our journey towards our final goal: the identification of a possible toolkit of elementary software units with which to simplify robot programming. The first step in this direction will be for us to show which context robotics fits in and which socio-economic motivations pushed to its development.

We are all aware of the fact that most of the wealth in modern societies is nowadays generated mainly by the industrial sector, and in particular by the manufacturing industry. Thus, it is clear that we need to improve the industrial production, if we want the general economy to grow. This means that industries need to increase the production of better quality end-products, that is to sell more goods at a more competitive price. The logical answer to such a need is the automation of the production cycle by means of dedicated machinery known as *hard automation*. In this regard, the increasing competition for shares of the global market acted as a sound economic ground for most of the manufacturing industries to pursue such a policy ([Owen 86]). However, we need to point out that even if the introduction of automation in the manufacturing

cycle of a product may be helpful, it should not be seen as the ultimate solution to all the problems. The employment of a new technology should always be accompanied by a careful study of what is really required and what the new technology can provide. For instance if a proper analysis of the manufacturing cycle is not done in advance, the adoption of machinery for the sake of automating some processes may result in a financial loss for the company involved.

We may all agree that automation provides industry with the capability of mass production, nevertheless there is another characteristic which modern industry should have: flexibility. In this ever changing world market, just the companies which are capable of reacting quickly enough to the needs of the customers can survive. Today, the decreasing costs of computational power has allowed, and in certain ways pushed, the development of a flexible conversion sector capable of processing any product within its power. Such a computerized flexibility shows itself in many ways such as *Computer-Aided Engineering* (CAE), *Computer-Integrated Manufacturing* (CIM), *Flexible Manufacturing Systems* (FMS), or *Flexible Assembly Systems* (FAS). In this respect, our attention will be focussed on the last two of them which directly involve robotics.

Flexible manufacturing systems were first conceived by Theo Williamson in the early 1960s with his System 24. However, because of economical and technological reasons, they were abandoned until they became towards the end of seventies a viable reality. Quoting the definition of FMSs given in [Owen 86], we can say that

Definition 1.2 (Flexible Manufacturing System) *An FMS is a computer controlled automated machining system for converting raw material into components of known and desired geometric quality.*

According to this definition an FMS is a production system made of identical and/or complementary *Numeric Controls* machines connected together by means of an automated transportation system ([Tempelmeier & Kuhn 93]). Each production system process, which is controlled by a dedicated computer, is usually monitored by computers at higher hierarchical levels within the system network. An FMS should be capable of processing workpieces within a certain range in an arbitrary sequence with

a negligible setup delays between operations¹. Because of the automatization of the tool exchange operations, the tedious and time-consuming interruptions for substituting or preparing a tool are performed while the machine is operating. Thus, while a workpiece is being processed, the next one has already been collected and prepared for the next cycle.

Flexible assembly systems are very similar to FMSs, indeed they make use of the same philosophies and technologies. The main difference lies in the fact that FASs are involved with the processing of components and materials into a final product. In this respect, quoting the definition of FASs given in [Owen 86], we can say that

Definition 1.3 (Flexible Assembly System) *An FAS is a computer controlled automated assembly system for converting raw material and purchased components into final products of a known desired functional quality.*

Comparing definitions 1.2 and 1.3, we notice that the emphasis in the latter is on compatibility of components and on automatic testing for functional quality. Basically, it does not include any shaping or forming of raw materials except where such a process is an essential and integral part of the assembly task.

Considering manufacturing industry as a whole, *assembly* represents the major activity accounting for more than half of the time and for almost a quarter of the total labour involved in the manufacture of a typical product ([Owen 84]). Because these figures are likely to increase in the near future, we can expect assembly activities to become even more predominant in the future manufacturing industry. Thus, it is important to develop more reliable and efficient assembly systems capable of adapting to the changes happening in the market. Robots provide the right flexibility but suffer from some complex problems such as position inaccuracy of the manipulator, dimensional variation of the mating parts and their physical interactions. In other words, robots, despite offering great promises, still struggle to cope with the uncertainty embedded in the environment. This is a problem which requires an answer, if we want to develop assembly systems capable of taking advantage of the enormous flexibility offered by

¹ This is possible because a set of pre-adjusted tools is made available through tool magazines with short access time at the machine with direct access.

them.

Modern industries have greatly employed automation in order to improve their production performance and constrain costs. Robotics, which may be viewed as a technological evolution of *hard automation*, offers the considerable advantage of providing the production cycle with the right flexibility to follow the market needs. This means that a certain task performed by a robot, if for any reason requires to be altered, may be easily reprogrammed without changing significantly the hardware.

Manufacturing industry has extensively used robotics in its assembly lines. Unfortunately, robots have still today failed to become the means of creating the promised completely automated industry. Among the many reasons accounting for such a failure, there is the intrinsic difficulty of programming them to accomplish tasks in the presence of uncertainty in the real world.

Nowadays the development of modular *hard automation* is increasingly challenging the employment of robots in an assembly line. In this regard, the possibility of reusing most of the same hardware for accomplishing a different task, which may be regarded as another way of reprogramming an assembly line, is the same characteristic offered by robotics. However, because of costs of development, the scales tip not in its favour and its future may well be put at stake. If some of the limitations which robots suffer would be solved, they could still stand and win this challenge.

1.2 Topic Problem

As mentioned above one of the biggest limitations suffered by robots is still their programming. This activity was initially thought of as innocuous, but it turned out to be unpredictably difficult in such a way that many researchers nowadays regard it as the main reason accounting for the failure of such a technology to take off.

When robots were first developed in the beginning of the fifties, they represented the natural evolution of hard automation. Compared with this last, robots had the advantageous characteristic of being programmable, that is the characteristic of performing a task by executing a program. Such a program was actually a series of point coordinates

in space, and its execution consisted in moving the robot arm through this sequence. Thus, the first robots were mere position playback machines which were largely employed in many industrial applications. However, surprisingly, they failed to take off in the field of assembly tasks, which many people hoped it would have been their greatest employer.

The development of computer-like languages raised the level of programming by replacing the mere sequence of work-cell points with an actual program developed and debugged out of the factory floor on a computer terminal with the aid of simulators. This possibility allowed to reduce the time which a real manipulator was kept off-line because the program was basically just down-loaded and then tuned. However, as manufacturing tasks got more and more complex, robots ran into a very difficult and unforeseen, or at least neglected, problem: *uncertainty* (cf. chapter 2 on page 13). In order to cope with it, sensors were introduced, but this did nothing else than increasing the complexity of programming. Indeed, this last became a real issue whose solution was not at all as easy as it was initially thought.

Several partially successful robotic systems were designed and implemented up to the mid-eighties, but they all ran into not negligible difficulties as soon as they tried to take into account the uncertainty of the objects and the surrounding environment. In this regard, all of them shared the same characteristic: the *functional decomposition* of the robot control system (subsection 2.2.1 on page 21). Many researchers, confronted with this problem, stepped back and reconsidered the whole approach to the design of a robot architecture. In this respect, a different answer was proposed by Brooks with his subsumption architecture ([Brooks 86a]). He criticized the main stream of research which was taking for granted that the obvious way of designing a robotic system was the implementation of a *sense-think-react* cycle, which from now on we refer to as the classical approach, without justifying the reasons why it was the optimal solution, or the only possible one (cf. subsubsection 2.2.2.2 on page 33). His idea was to forget completely about such a cycle and to replace it with a series of *sense-react* modules capable of being added to a system without requiring modifications to any parts of this last (incremental addition). He was extensively inspired by animal evolutionary theory in which “*intelligence*” was more a development of capabilities than a monolithic entity.

However, this kind of ideas were not the only sign of novelty introduced: his approach to the design of a new architecture with its justification was the actual breakthrough compared with what we called classical approach.

Brooks' architecture, although showing interesting results, was not really suitable to the highly structured world of an assembly domain where the majority of the tasks require specific plans to be accomplished. In this regard, Malcolm proposed his behaviour-based assembly system ([Malcolm *et al.* 89]), which may be considered as a hybrid solution between classic and behavioural approaches (cf. subsection 2.2.2.3 on page 38). In this respect, Brooks' work played an inspiring role, however Malcolm still assumed to use a central entity to plan its interactions with the surrounding world and a behavioural agent to carry it out. The presence of such an agent was the real break with the classic cycle because this allowed a plan to be expressed not in terms of manipulator motions required to accomplish a particular task, but in terms of actions on the objects in the domain of the task. These actions are packaged in the form of task-achieving units called behavioural modules. Their structure has not been completely defined as yet, but, as discussed later in chapter 4 on page 84, they may be regarded as a combination of hardware and software.

There is an important point associated with the design of a behaviour-based assembly system: the number of modules required in order to have a usefully programmable system. This is a crucial question because only if a limited number of them can be found on a reasonable large domain, we can actually erect a general-purpose behavioural language capable of doing useful work.

The research described in this thesis will tackle this particular problem within the framework of Malcolm's paradigm. Such a problem has similarities with that faced by Turing in Computer Science where his hypothesis of few commands for running a general-purpose computer is now widely accepted (cf. section 2.3 on page 45). If such a set had not existed, we would have been forced to define customized languages to solve certain problems to be run on specific machines. However, the main point is not just the existence of such a small set but is the possibility of conveniently packaging these operations in an opportune form of high-level language, where no more than 30 or 40 basic instructions are required to do anything programmable. In this respect, we

wonder if, at least in principle, we could do the same thing for robots too. So far we have been forced to define and use particular languages to solve certain problems to be run on specific machines. Thus, if a set of basic high-level robot operations packaged in modules exists, we may be able to define a language on top of it and to constrain all the complexities within the software.

Considering the problem, we have to observe that it would be too ambitious to face it on the whole assembly task domain. Thus, based on the fact that the number of most common manufacturing tasks is rather limited, as resulted from several surveys carried out since the seventies, we restrict our research just to a group of tasks which accounts for 80% of the total.

1.3 Thesis Claim

As discussed in subsection 2.2.1, the classic approach to robot control showed its limits in the attempt to define general-purpose robot programming languages. In this respect, the behaviour-based assembly architecture was proposed as a possible answer to those limitations. However, in order to be a viable alternative, a crucial point still needs to be solved: the number of modules required by such a system.

As resulted from several surveys, most of the assembly operations cluster into just a few classes. Since an operation may be viewed within the behaviour-based assembly approach as the outcome of the execution of a behavioural module, our problem is to investigate how many of them are required in order to cope at least with the few aforementioned classes on a certain assembly domain. In this respect, our assembly world will be limited due to hardware constraints just to polyhedrons with parallel faces and to cylinders, and our task domain just to round tandem peg-in-hole, screw fastening, stacking, snapfit, and retaining.

The claim that we make in this thesis is that a small group of 9 behavioural modules, which copes with the tasks listed above, is elementary and general enough on our restricted assembly world to be considered an embryo of a basic set on which to base the construction of an assembly behavioural language.

Anyhow, these 9 modules should not be viewed as the definitive elements of such a set, since we expect, in order to extend its domain of applicability, to add a few more to it in the future.

The modules of this set, as emerged from our experimental work, can be opportunely composed so that to achieve the assembly of three different assembly test beds: two benchmarks, a particular retaining assembly (STRASS), and three electric torches.

Given different tasks than these, and a robot work-cell with different capabilities, a different set of basic behavioural modules would emerge. They would be different in detail, because of local differences in task and technology, but would, in terms of purposes, approximate the same basic set of assembly sub-tasks that several investigators have identified. However, what would prevent each new assembly task requiring one more addition to the basic set? First, it should be noted that the utility of a human engineer's tool-kit is not compromised by the need for the occasional special tool for a special task: the benefits of modularity and generality are sufficient to outweigh the occasional exception. There is also the natural tendency to design tasks to fit the tools available, *e.g.*, the tendency in modern assembly to reduce the necessary scope of automation by largely restricting part-fitting to vertical downwards motion. Thus, this thesis shows, by example, how to construct one particular general set of behavioural modules, claiming that the combined resources of technology and human ingenuity will find the construction of industrial versions well within their scope. It would be useful if some formal test existed, such as Turing-equivalence in computation, to prove that this will always be possible. The domain of engineering ingenuity has not developed that kind of formality yet. The best that can be done is to attempt to convince experts by example that this is the sort of task they can be confident, one way or another, of solving.

1.4 Thesis Layout

The research presented in this document is organized in 6 more chapters. The next one will present and discuss in detail the problem of robot programming at task level and will compare the behaviour-based approach to the classic one. Another point examined

in that chapter is the task decomposition in terms of behaviours. As discussed there, the most common assembly processes cluster in just few classes. Each of them may require quite a few modules in order to be performed on a reasonable assembly domain. However, as emerged from other work in the field, it is possible to define a group of assembly commands coping with certain classes of tasks on a certain domain.

Chapter 3 will discuss clearly the topic problem we are tackling and will define the project which addresses it. In this respect, it will state the basic hardware and software assumptions, the assembly test beds to be performed, and the extra hardware required.

Chapter 4 will analyze the theoretical viewpoint of behaviours and behavioural modules. In this respect, it will present a way of describing their structure and composing them together. Another important point also discussed there is the concept of hierarchy of modules and hierarchical level which are central to our research of the basic modules.

Chapter 5 will describe the experiments conducted on the three testbed chosen (benchmarks, STRASS, torches). In this regard, it will present an important element in our research: a guarded motion. This may be viewed as a low-level basic unit on which more complex entities, such as many of our basic modules, may be defined. Another important point discussed will be the implementation of a round peg-in-hole module capable of achieving tandem insertions of a peg in detached coaxial holes. Other modules emerged from the assembly experiments with the three test beds will also be described and discussed there.

Chapter 6 will analyze the results emerged from our experimental work. In this respect, it will argue the existence of a basic set of modules on our restricted assembly world and will synthesize its 9 elements. It will also present at the end the outline of a possible behavioural language which may be constructed on top of a more complete basic set.

Chapter 7 will finally summarize the results achieved and point out the scientific contributions brought in by our work. It will also present some possible areas of further research development. The chapter will then draw final conclusions.

1.5 Summary

This chapter showed how, under the pressure of the market, industries were basically forced to introduce automation in their production line. The technological evolution of dedicated machinery culminated with reprogrammable general-purpose machines, today known as robots, which provided the production line with that flexibility which the dedicated *hard automation* was lacking. In industry just spray painting and welding, plus some pick-and-place, turned out to be the biggest employer of this technology. Unfortunately, the use of robots in manufacturing assembly revealed instead to be very disappointing.

The joint use of computers, automation and robotics affected so much the manufacturing industry that nowadays we can talk of *Computer-Aided Engineering* (CAE), *Computer-Integrated Manufacturing* (CIM), *Flexible Manufacturing Systems* (FMS), or *Flexible Assembly Systems* (FAS). The last two aspects listed here above are more directly concerned with robotics, and are also the only two ones considered. We gave the definitions of both of them and showed that they are basically following the same philosophies and technologies. The only difference is that whilst FMS is involved with converting raw materials into components, FAS is concerned with converting raw materials and sub-components into final products.

By analyzing the limitations which robots still suffer, we showed that the main reason which accounts for the failure of robotics to take off is due to a lack of reliable architectures capable of coping with the presence of uncertainty in the environment (section 1.2 on page 5). In this respect, we introduced Malcolm's behaviour-based assembly system which may be an interesting alternative to what we called classic approach to control system. However, in order to be as such, an important issue needs to be solved: the number of behavioural modules required to have a working system. We pointed out that this is the topic problem which we are going to tackle in this thesis. In this regard, we restricted our research just to a limited number of tasks which, according to several surveys, occur frequently in manufacturing industry and account for 80% of the total assembly processes. Moreover, due to hardware limitations, we constrained our assembly world just to polyhedrons with parallel faces and cylinders. The claim

that we made on such a domain was that we could define a group of 9 behavioural modules capable of coping with the aforementioned 80% of tasks: such a group has to be regarded as an embryo of a basic set (cf. section 1.3 on page 8).

We concluded the chapter outlining the layout of the thesis which synthesizes the path pursued in our quest for the basic set (cf. section 1.4 on page 9). To this purpose, we organized the thesis in the following chapters:

- **2 Literature Review**
- **3 Definition of the Project**
- **4 Behaviours**
- **5 Project Experiments**
- **6 Analysis of the Results**
- **7 Conclusions and Further Work**

Chapter 2

Literature Review

The purpose of this chapter is to present and examine the relevant literature to the work developed in this thesis. In order to do so, we will have to talk first about robot programming and its historical background (section 2.1), then about system architectures with particular reference to the behavioural one (section 2.2), and finally about behavioural decomposition of manufacturing tasks (section 2.3).

2.1 Robot Programming

The initial hope of creating a fully automated industry by introducing robots on the factory floor was not meant to live long, indeed the whole idea had soon to be reconsidered ([Owen 86]). There are many reasons for this, however it is widely reckoned that the limitation was mainly due to a lack of good and easy to use programming tools with which to instruct robots to perform industrial tasks.

Since the first applications of robots on the factory floor it was realized that sensorless robots were unable to cope with all the uncertainties embedded in the environment typical of the assembly world. Thus, it seemed logical to solve this issue by introducing sensors and programming languages incorporating their use. However, this made things even more complicated because the use of sensors for constraining the influence of uncertainty proved to be not easy to implement, indeed it just increased the complexity of the entire programming system. So, it became clear that coping with uncertainty was the real key bottleneck of robotics. Some researchers then stepped back and

reconsidered the entire robot architecture from its foundation. This realization is an important point and this section aims to show it in more details. To this end we will talk first about a hierarchic classification of the different programming methods in four levels (subsection 2.1.1) and then for each of them we will examine the most salient characteristics (subsections 2.1.2 to 2.1.5).

2.1.1 Classification

We already know that the main advantage of robots compared with *hard automation* is the possibility of being re-adaptable to perform a different task simply by changing the *program* describing it. When robots made their first appearance, the difficulty of this activity was largely underestimated. Indeed, most of the scientific community thought that it was just a matter of time before this technology took over the old special-purpose machinery. Unfortunately, as time went by, industry became more demanding and its unsatisfied expectations showed soon the limits of robotics. As we mentioned at the beginning of section 2.1, it is nowadays well acknowledged that programming assembly robots for coping with uncertainties in order to perform industrial work has been the key bottleneck in the development of robotics. But why has programming robots been such a limiting factor? We reckon two main reasons:

- first, the inherent difficulty of programming and debugging tasks on the factory floor by means of guiding electro-mechanical devices, which were common up to the mid-seventies¹, or by means of off-line textual languages, which started being a viable alternative only from the second half of the seventies and are still used nowadays;
- second, the difficulty of dealing with a world where uncertainty plays a determinant role.

A general solution to the issue of robot programming has not yet found a satisfying answer and it is still in great part a matter of research. However, we can say that since the first days of robotics, technological development has progressively changed the way

¹ This kind of devices are still today widely used for programming arc welding and spray coating applications.

in which this activity is performed. Lozano-Pérez identified roughly three different levels at which robots may be programmed ([Lozano-Pérez 82]).

Teach-by-Showing At this level (the lowest and oldest one) the operator gets the robot to perform work by recording the path to be followed by the manipulator. Hardware tools, such as a teach pendant², are used to this purpose. This level of programming is still today largely employed for welding and spray coating applications.

Robot Level Here the operator programs the robot in terms of manipulator motions by using software tools such as textual programming languages similar to those used in computer science. This represents the current state-of-the-art in industry.

Task Level In this last level the operator programs the robot by expressing what the manipulator has to do in terms of tasks and not in terms of the actual manipulator motions required for accomplishing it. As we can easily understand, the operator in this case is not anymore concerned with the actual robot hardware, indeed the only problem which he needs to face is the description of the final task in terms of what he wants to achieve in the assembly world. This level of programming is still the subject of most of the current research in robotics and artificial intelligence.

These three categories are actually a crude classification of robot programming. A deeper examination conducted in [Malcolm & Fothergill 86] leads to the finer categorization described here below.

- **Joint-Level Programming** in which the position of the end-effector is specified in terms of the joint angles and joint displacement of the manipulator. This is usually achieved first by physically guiding the robot end-effector through a path in space by means of specialized hardware and then by letting the robot playback such a path.
- **Manipulator-Level Programming** in which the motions of the robot are specified in terms of the required position of the end-effector expressed in various general co-ordinate systems.

² Device which is used to drive and control the robot joint motors and to power the manipulator arm and wrist.

- Object-Level Programming in which the motions of the robot are specified in terms of the spatial relations which are required between features of the object to be assembled, that is for instance how parts fit together, or how the gripper has to hold a part, and so on.
- Task-Level Programming in which the robot motions are specified in terms of the assembly to be performed.

The largest part of research pursued in robot programming falls in the first three classes of this second classification. The work presented here in this thesis, in particular, falls in the last two groups.

2.1.2 Joint-Level Programming

As we just mentioned above, a robot is programmed at joint-level by manually driving the manipulator to the desired locations within the work-cell and by recording the internal joint coordinates corresponding to each location where the robot arm was driven. End-effector operations, such as *opening/closing* a gripper or *activating/inactivating* a welding gun, are then specified at some of these locations. The program which results at the end is a mere sequence of vectors of joint coordinates plus the activation/inactivation of external equipment, such as a particular end-effector. The robot performs its job by moving progressively to each specified joint coordinate and eventually issuing signals at some of them.

Programming methods which belong to this level are called *teach-by-showing* methods, or sometimes *guiding* methods (cf. page 15). They are very simple to use and do not require particular knowledge skills by the programmer, but simply a good ability to perform the very task which the robot is being taught to carry out. When industrial robot applications evolved and incorporating computers into industry became cost-effective, using guiding methods for programming robots started being quite limited, particularly regarding the use of sensors. In this method, in fact, because of the structure of a program there is no way to make the robot act according to the surrounding world sensed by its sensors. The programmer can just specify simply a single sequence of joint coordinates. He cannot make the robot perform any loops, or conditionals or

computations. In some applications, such as spray painting, arc welding, or materials handling, guiding methods are more than enough, and indeed in these cases it is still the best way. However, in more modern applications, such as assembly for instance, a robot needs a way to act according to sensory inputs, data retrieval or computations, and the only sensible way to provide these features is by using a general-purpose programming language related to those used in Computer Science. This last method, which belongs to a higher level of abstraction, is the subject of our discussion next.

2.1.3 Manipulator-Level Programming

At the end of the previous subsection we mentioned that modern robotic industrial applications require programs more sophisticated than those obtainable with *teach-by-showing* methods ([vanAken & vanBrussel 88], [Kochan 95]). Towards the end of the sixties the employment of computers at industrial sites allowed textual robot programming languages capable of accessing sensory data, making computations, and specifying robot motions to become a viable alternative to guiding methods. A typical program is a description of the robot actions and motions specified in terms of the required positions of the end-effector. Because of this characteristic this kind of programming is classified as *manipulator-level programming* ([Malcolm & Fothergill 86]), or sometimes also as *robot-level programming* ([Kempf 81], [Lozano-Pérez 82], [Craig 89]). Since a program at this level is usually developed on a computer and debugged simply with the aid of a computer simulator ([Hollington 94]) without needing a robot to be accessible on-line, we often talk in this case of *off-line programming* ([Regev 95]).

Compared to guiding methods textual programming offered the enormous advantage of being able to modify the robot behaviour according to the current readings of sensors. This enabled robots to cope with a world with greater uncertainty, such as the position of external objects, enhancing in this way the whole spectrum of applications.

Nowadays one of the most common ways of programming an industrial robot is by a combination of textual programming and teach pendant programming ([Gini 87], [Wittenberg 95]). The former, which may be done off-line in front of a terminal, is used for specifying the logic and the sequence of steps defining the task to be accomplished, whereas the latter, which can only be done on-line with the teach pendant, is used for

storing eventual point locations of the workspace needed for the working cycle.

2.1.4 Object-Level Programming

Basically all the robot languages at the manipulator-level, although being able to describe all modern manufacturing processes, are not at all user-friendly. Because of the low level of abstraction involved, they have to describe tasks in terms of the point-to-point motions and actions within the work-cell of the robot manipulator. Generally speaking, most of the robot languages currently available at this level are hard to learn and offer rudimentary programming tools compared with those offered by the more familiar computer programming languages³. Moreover, they require an end user who is both a skilled programmer in that particular language and a skilled strategy designer in manufacturing processes.

In order to solve the limitations of the manipulator-level programming, many attempts have been made to raise the level of abstraction from robot motions to spatial relationships between features of the parts ([Popplestone *et al.* 80], [Latombe & Mazer 81], [Mazer 83], [Hayward & Paul 83], [Metta & Oddera 93]). Programming in these terms implies describing an assembly task as a sequence of the required spatial relationships between parts, such as for instance *align the axis of the peg with the axis of a hole*, or *place bottom face of object 1 against upper face of object 2* ([Popplestone & Ambler 83]). The actual sequences of robot motions required to achieve these relationships are hidden to the end-user and are automatically generated by the language compiler. This is for a human operator a much more natural way to reason about the assembly than a list of point-to-point motions of the end-effector within the work-cell.

The best example of object-level programming system is represented by the language RAPT ([Popplestone *et al.* 78]), whose syntax was directly taken from the *NC* machine language APT. RAPT was capable of translating symbolic specifications of geometric goals into a sequence of end-effector positions, and this was its main concern. It did not attempt to deal with obstacle avoidance, nor with grasp planning, and nor with sensory operations planning. An object, which the robot had to deal with in order to

³ This second drawback is solved by those robot languages which are extensions of computer languages such as *AdaTM*, *C*, or *LISP*.

accomplish a certain assembly task, had to be represented by explicitly describing all its faces, their position and orientation. RAPT as originally implemented did not take advantage of eventual symmetries in the assembly. This was a limitation which was overcome only after it was rebuilt by making use of *labelled solid geometry* which was based on group theory ([Poppstone 83]). A body was defined as a function from \mathbb{R}^3 to a set of labels which served to identify the original features. The use of group theory enabled RAPT to simplify the description of the task by taking advantage of eventual symmetries in it.

One of the goals aimed at by the entire RAPT project and also aimed at by other programming systems sharing the same level of abstraction was the simplification of the task description, so that even users who had little familiarity with programming languages would be able to program the assembly task. Unfortunately, RAPT did not achieve this goal, being hard to learn and use even for an experienced programmer. This was partly overcome when a graphic modelling system for visualizing the feature currently referred to was introduced. It was in fact relatively easy for a programmer to forget about which feature of an object was referred to as, for instance, *side A* when the object itself had been rotated to match some previously specified relationship. A visual modelling system greatly helped showing and examining each time the relationships holding in the assembly, however even with this aid RAPT did not manage to become easy enough and therefore it failed to be accepted by industry for commercial development.

The main drawback of an object-level system is the assumption that the real world of the assembly work-cell conforms to the geometric model employed by the system itself. However, this is not in general the case, indeed the uncertainty embedded in the real world can not conform with any ideal model of it. The attempt of introducing geometric tolerance analysis into RAPT in order to handle the uncertainty ([Fleming 87]) proved in fact to be too computationally expensive. More successful was instead the introduction of vision sensing into RAPT for detecting translational and rotational variations in the initial position of the parts ([Yin *et al.* 84]). Nevertheless, the incorporation of sensors for changing the actions of the robot rather than its mere destination remained a difficult problem. In fact, as long as the sequence of positions and motions of the robot

are determined in an off-line system which knows little about the on-line system and its sensor availability, it is difficult to include alternative strategies dependent on sensor data unless the system is told more about the work-cell conditions.

2.1.5 Task-Level Programming

The kind of textual programming examined and described in the previous subsections was essentially concerned with programming languages at the robot level (cf. page 15), in other words with languages which needed a careful specification of the task to be performed by the robot in terms of the manipulator movements necessary for carrying it out (manipulator-level programming), or in terms of the spatial relationships between geometric features of the parts (object-level programming). The former is still today the most widely employed in industry for manufacturing tasks⁴. However, it is closer to the machine than to the end-user, and this makes the activity of writing reliable robot programs very hard. Object-level programming is more appealing, but unfortunately, as we showed at the end of the previous subsection, is unable easily to exploit the sensing capabilities of a robot system in order to react promptly to the work-cell conditions with alternative strategies.

In subsection 2.1.1 we mentioned a higher level of abstraction: *task-level programming*. It started to be investigated around the end of the seventies ([Park 77]), but unfortunately so far none of the prototypes built (AUTOPASS [Lieberman & Wesley 77], LM-Geo [Mazer 83], ESTEREL [Berry *et al.* 87], Handey [Lozano-Pérez *et al.* 92], ROPSII [Nakano *et al.* 94]), or simply just proposed (TWIN [Lozano-Pérez & Brooks 85]), has managed to reach commercial use. Basically at this level the program which a robot has to run in order to accomplish a task is no longer thought of in terms of motions and operations of the manipulator, but instead in terms of the operations on the *objects* in the task. In other words a task such as, for instance, the notorious *peg-in-hole* would be described simply as `Insert Peg-In-Hole` without specifying the sequence of instructions coding the insertion strategy.

At this level of programming the operator does not need any knowledge of the under-

⁴ The more archaic joint-level programming (cf. page 15) is still today preferred for some spray coating and welding applications.

lying assembly execution agent, nor does he need to know how the agent senses the world and carries out the required task. This of course requires the off-line system to have an extensive knowledge of the robot environment, of the parts involved in the task, and of the actions which can be legally performed on them within the work-cell. A program written using a task-level specification has to undergo a translation process which transforms a high-level description into corresponding low-level manipulator commands, which are the only ones understandable and executable by a robot controller.

Most of the systems developed at this level of abstraction rely on the fundamental assumption that the task has to be described in terms of point-to-point motions in the robot envelope. As we mentioned at the beginning of this section, the problem of dealing with uncertainty made clear that architectural choices, such as those implied by the assumption above, play an important role in the development of a reliable and easy to use robot system. This leads in a natural way to discuss the issue of robot architectures, and it is here that we are going to focus our attention next.

2.2 System Architectures

Robot programming, as we explained at the beginning of the chapter, has been one of the major bottlenecks in the development of robotics applications. This activity, initially thought of as an easy task, turned out to have unpredicted complexities, which are still today research topics.

The effort of raising the abstraction up to the task level (section 2.1) leads us to face the issue of developing an appropriate system architecture. In the following two subsections we examine the two main paradigms in the architectural design: the classical approach and the behavioural approach.

2.2.1 Classical Approach

Since the beginning of the seventies most of the research efforts in robotics aimed to create a machine capable of carrying out manufacturing work, such as assembly, simply by executing a description of the task in terms of the operations onto the objects to

be manipulated and not in terms of the motions needed to achieve it. However, since a program expressed in part-motion terms has to be run by a physical agent which understands only manipulator-level commands, i.e. robot motions, the program needs first to be translated into these terms. This is achieved by means of a *task planner* which generates for each sub-task listed in the program a corresponding reliable plan expressed in terms of the manipulator motions and actions necessary to accomplish it.

Basically, a planner acts as a compiler which converts task-level specifications into their corresponding manipulator-level ones. In order to perform such a translation, the planner requires a great deal of information about the work-cell, the robot, the objects it has to deal with, and also about the initial state of the environment and the desired final goal to reach. However, a planner designed in this way conceptually does not take into account the fact that the real world in which the agent has to perform the required task is not ideal. Whatever strategy we may devise in order to accomplish a certain manufacturing task, we may always find some degree of uncertainty in the location and shape of the objects involved and/or in the accuracy of the manipulator and its tools. The difficulty, though, does not arise only from dealing with the embedded uncertainty of the real world, but also from the fact that the planner has to guarantee the *reliability* of the robot program synthesized in output⁵. Because uncertainty is a drawback always present, the obvious solution to choose in order to cope with it and guarantee reliability of the program is to provide the robot with sensing capabilities (*e.g.* [Lindstedt & Olsson 93], [Borovac *et al.* 94], [Inoue 95]).

Assembly robotics has made a considerable use of sensors, jigs and fixtures in order to reduce, or at least limit, the uncertainty during the execution of the assembly process. However, all of this has resulted in increasing the complexity of reliable plan generation⁶ ([Fleming 85a], [Fleming 85b], [Laugier 88b]). Despite this complexity explosion, though, it is still possible to design and implement planners capable of synthesizing at the end of the conversion process a reliable description of the task to be accomplished in terms of a particular manipulator-level language (*e.g.* [ElMaraghy & Rondeau 92]). Here, we stress once again the point that such a process is not just a mere substitu-

⁵ See subsection 2.2.2 for a brief discussion of *uncertainty*.

⁶ This is discussed in more detail in subsection 2.2.2.

tion of task commands with opportune robot-level ones, but it is a real translation which has to take into account any capabilities for compliant motion, guarded motion⁷ and error checking. The resulting program synthesized by the planner is therefore a sensor-based robot-level program.

A careful analysis of the planning process allows us to identify three separate phases, which are, however, not mutually independent ([Lozano-Pérez 82]).

World Modelling. This stage is necessary to provide the planning engine with an accurate description of the whole work-cell environment, including the objects present in it and their features (*e.g.* [Stobart 87], [Laugier 88a]).

Task Specification. This stage is necessary to define a representation of the task the planner has to accomplish.

Robot Program Synthesis. This is finally the last and crucial planning phase which has actually to synthesize, from the world and task descriptions obtained in the other two stages, the robot motions, actions and controls necessary to carry out the required task (*e.g.* [Rogalinski 94]).

A great deal of the research efforts in task-level programming are nowadays dedicated to these three fields.

In the past years many robot architectures designed following this paradigm have been proposed. In [Kohn 91] an intelligent real-time system with a smart declarative controller and a reactive planner for operating a chemical reactor is presented. In such a system the procedure specifying the controller is not given explicitly but it is inferred from its knowledge base. In [Lefebvre & Saridis 92] a hierarchic organization of the functions of an autonomous agent (*execution*, *coordination*, and *organization* levels) is discussed, and the two lower levels (i.e. *execution* and *coordination*) are implemented. In [Coste-Manière *et al.* 92] the main features of the synchronous task-level language ESTEREL are presented. This language is thought of as an application programming language capable of expressing in a natural way the logical and temporal dependencies of the actions. In [Sato *et al.* 95] a task-level teaching system (ROPSII) with its main

⁷ A guarded motion is a robot motion monitored by one or more sensors.

features is presented. Such a system is characterized first by providing the operator with an object-oriented language (ROPSII) with which to make three-dimensional visual programming, and second by allowing him to describe the failure recovery sequence separately from the main task sequence. By using the knowledge of the task and error recovery sequences, the system develops automatically a collision avoidance trajectory.

As we can see from the few examples reported above, the research field of task-level system architecture is very fertile, however all these systems followed the same paradigm: a centralized control. This transported to mobile robotics translated into the famous three-steps process of *sense-think-act* which consisted in sensing the environment, planning strategies and actions in it, and then performing the plan. The same paradigm transported to assembly robotics translated instead into a different three-steps process which consisted in modeling world uncertainties, planning the use of sensors for removing these uncertainties, and finally compiling it down into a target manipulator language.

Summarizing, the classical knowledge-based approach to robot programming tacitly assumed that however one decides to build one's high-level planners, in the end they produce a robot program in terms of motions of the manipulator in the Cartesian space, in other words in terms of one of today's robot programming languages. Such a paradigm is a sound approach to the problem, but it is not the only possible one. The next subsection will examine a different paradigm: the behaviour-based approach.

2.2.2 Behaviour-Based Approach

Considering the *planner-agent* system as a whole, we may notice that the complexity of the overall system is partly concentrated in the planner and partly in the agent: the simpler the agent, the more complicated the planner, conversely the more sophisticated the agent, the simpler the planner. There are two functions describing the system complexity which we have to consider: one is the complexity of the agent with respect to the percentage of system inside the agent itself, and similarly the other is the complexity of the planner with respect to the percentage of system inside the planner itself. If we assume these functions to be linear and with the same slope, then we would have to conclude that the total complexity, which is the sum of planner and

agent complexities, is constant and that there would be no benefit in creating hybrid systems by shifting part of it from planners to agents. Thus, complexity could be put in planners or agents as convenient with no savings (cf. figure 2.1a). If we assume instead that the aforementioned functions are not linear and the curves associated with them have the shape shown in figure 2.1b, then the optimum solutions would be either an extremely complex planner and a stupid agent or a very smart agent and no planner. In other words, the optimum would be a system which is either totally dominated by a planner or totally reactive: any hybridization would simply cause extra complexity. If we finally assume that the functions above are again not linear but this time the curves associated with them have the shape shown in figure 2.1c, then a hybridization of the system between planner and agent would reduce the total complexity and the optimum solution would lie at the minimum of the system complexity curve. Behaviour-based

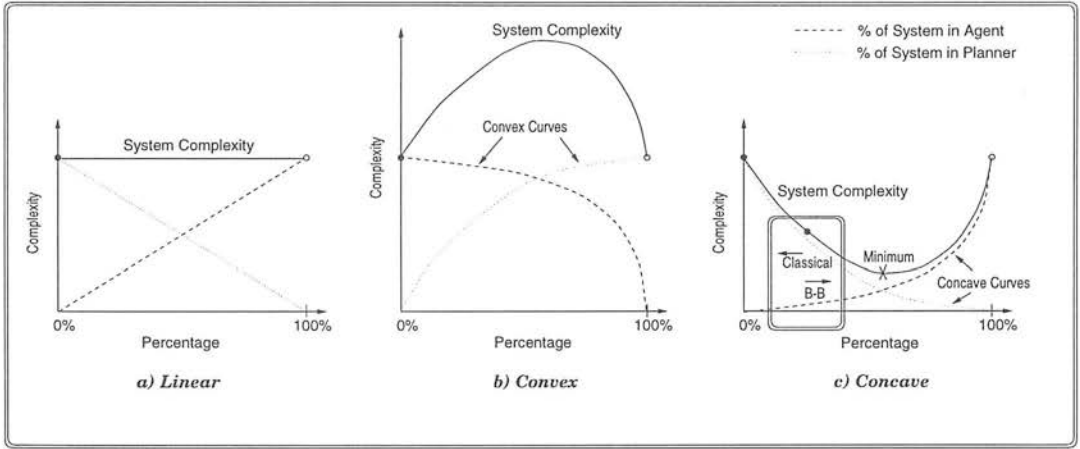


Figure 2.1: Curves describing the Planner-Agent System Complexity.

enthusiasts assert that making agents more capable does reduce the total complexity, so, according to this view, the shape of the curves associated with the two functions introduced above is assumed to be similar to that drawn in figure 2.1c. Thus, there is an optimum solution which has the lowest total complexity, and it is represented by an opportune hybridization between planner and agent. Solutions following the behaviour-based approach move towards this optimum as opposed to those following the classical terms.

As we discussed in subsection 2.2.1, synthesizing a plan in terms of the manipulator

motions in the Cartesian space implies on the one hand to have a considerable knowledge of the work-cell, and on the other to predict the possible error situations and generate the relative recovery procedures accordingly. As discussed above, a planner which deals with all these details makes the overall planner-agent system very complex, so why should an agent programmed in such terms be considered as the optimum one? Considering the knowledge-based assumption of classical artificial intelligence, the answer is Cartesian Geometry itself, because, according to our Newtonian view of the world, our understanding of the assembly process may be more easily expressed in terms of shape, mass, motion, force sensing, and camera views. This is the basis of a sound formal description of the assembly process but is it necessarily the case that using a description of a process as an ingredient in its implementation is the best way of doing it?

Let us suppose we are asked to write a computer program which draws from a given point an equilateral triangle on the computer screen. Using trigonometry we are able to compute the exact Cartesian coordinates of the other two triangle vertices and draw the corresponding triangle. Let us suppose now that we are asked to generalize the solution found before in such a way that we can draw triangles in any orientations. With the help of trigonometry the solution is still possible, but it becomes more and more complex to be written and more and more difficult to be understood by somebody who does not know trigonometry (cf. figure 2.2 here below). If there is an error in

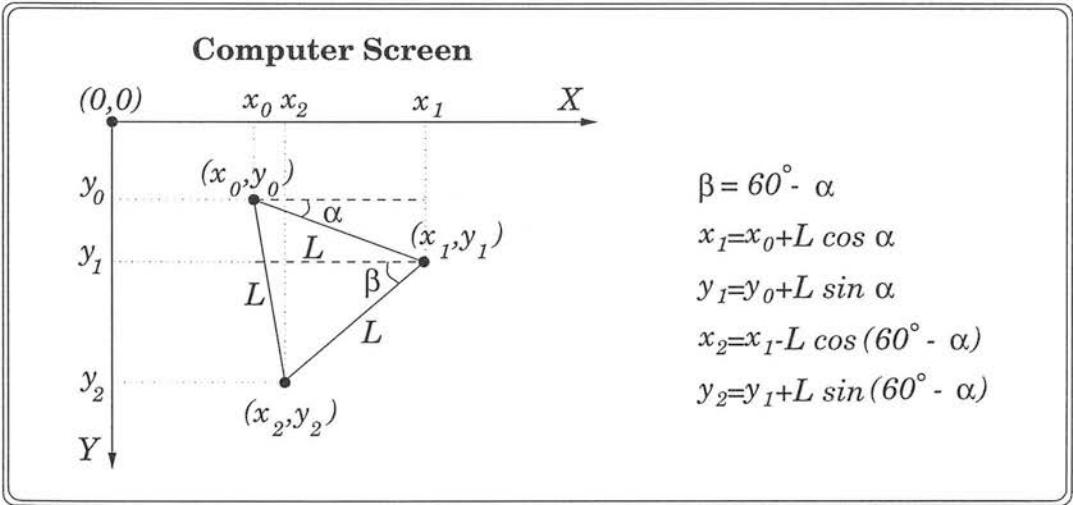


Figure 2.2: General Solution for drawing Triangles on a Computer Screen.

the solution, it is very hard to be noticed unless a complete review of the solution is carried out. But is this the only way to draw triangles on a screen starting from a given point? Well, let us suppose to use a different approach. Let us imagine we can tell the computer to draw a line in a certain direction for a certain length L and then turn about 120° , draw again a line of the same length L as before and turn 120° , and finally draw the last line of length L from here to the initial point. In other words, what we have ended up with is a program written using navigational geometry. Such a program⁸, not only solves the initial problem but also any possible generalization of it in a very elegant and easy to read way. Moreover, the solution in navigational geometry terms can be easily understood by anybody and does not require any particular mathematical knowledge in contrast with the previous one as we can see here below in figure 2.3.

Navigational Geometry	Cartesian Geometry
Procedure Triangle[$(x_0, y_0), \alpha$]	Procedure Triangle[$(x_0, y_0), \alpha$]
Begin	Begin
From (x_0, y_0)	$x_1 = x_0 + L \cos \alpha$
Direction α about the X-axis	$y_1 = y_0 + L \sin \alpha$
Repeat 3 times:	$x_2 = x_1 - L \cos(60^\circ - \alpha)$
Draw line of length L along α	$y_2 = y_1 + L \sin(60^\circ - \alpha)$
Turn 120° from here	Draw line from (x_0, y_0) to (x_1, y_1)
End	Draw line from (x_1, y_1) to (x_2, y_2)
	Draw line from (x_2, y_2) to (x_0, y_0)
	End

Figure 2.3: Solution in Navigational Terms vs. Solution in Cartesian Terms.

The main difference between the two solutions is that the one expressed in terms of absolute Cartesian points needs to know in advance the points between which it has to draw a line, whereas the other needs only to know where it has to go from where it is. In other words the initial solution requires a complete knowledge of the world it has to deal with in absolute terms, whereas the second one needs only to know where to start and in which direction to go, because the rest can be obtained by its local terms.

Following this metaphor, if we regard the program as a plan and the drawing as being performed by an agent, then the capabilities of the agent significantly affect the com-

⁸ We neglect here at this high description level the approximation errors which may lead the third line not to reach the starting point. Nowadays well known lower-level programming techniques can easily fix this problem.

plexity of the planner. In this case a small change in agent complexity leads to greater changes in plan complexity.

This metaphor underlines also another point: all the systems developed following the classical architecture paradigm were viewing the world in absolute terms, but this may well not be the only possible solution. We dedicate this subsection to show that there are other architectural approaches.

As we mentioned at the beginning, the level of details the task planner should reach in the synthesis of a reliable robot plan is a crucial issue. The choice made in all the task level systems (cf. subsection 2.1.5 on page 20) was to generate the plan for accomplishing a task in terms of the instructions available in the particular programming language understood by the robot controller. In general since current manipulator programming languages are based on point-to-point motions, the assembly plan has to be described in such terms. This approach is the simplest and most natural way for explaining to a machine how to accomplish a task, however it takes for granted the justification of being the best approach, or even of being the only possible one.

In this subsection we present three different architectures (Handey, subsumption architecture, behaviour-based assembly) which broke away from the classical paradigm. Basically, they all criticized the idea of a centralized *control* in different ways. The details of their criticism will be discussed later on in this subsection, for the time being let us give a few comments about each. Handey shifts part of the complexities from the planner into the agent, an agent which is still thought of in classical terms. The subsumption architecture instead replaces completely the *sense-think-act* loop, typical of the classic approach applied to mobile robotics, with a set of reactive hardware/software *sense-act* modules capable of promptly interacting with the surrounding environment without any centralized coordination. In other words, it replaces what we may call *centralized intelligence* with a *distributed* one. The behaviour-based assembly finally accepts part of the criticism to the classic approach made by both the subsumption architecture and Handey. In other words, it takes away the complexities from the planner in the same way as Handey, but it implies instead the use of a behaviour-based agent to carry out the plan.

As mentioned earlier, with the presentation of these different approaches we do not want to argue that they are a better solution compared with the classic one, but we want simply to point out that some problems such as robot programming may have more than one solution. In these cases choosing a different way to tackle the problem and giving some justifications for such a choice may lead to a different solution which may be better or worse than the other ones according to the point of view. Now let us go to discuss in more details the three architectures mentioned earlier.

2.2.2.1 Handey

The approach taken by Lozano-Pérez and actually implemented in the programming system Handey ([Lozano-Pérez *et al.* 92]) acknowledges part of the criticisms of complexity explosion in the generation of reliable robot plan within the classical terms, but, as we mentioned above, he just shifts the complexity from the planner to the agent. Handey is designed as a task-level robot programming system which requires just a description of the task to be accomplished in terms of operations on the parts. However, given a goal to achieve within the space of the tasks, it still needs a description of the surrounding environment and of the robot. Nevertheless, once the goal is stated and the knowledge is available, Handey generates a reliable sequence of robot commands which carry out the required task. The system so far assumes to deal just with *pick-and-place* tasks of objects modelled as general polyhedra and placed in carefully modelled environments. It also assumes to be run by robots with up to six joints equipped with parallel-jaw grippers.

A successful execution of a typical *pick-and-place* operation is subjected to several constraints which we may synthesize as follows:

1. there exists a collision-free path kinematically feasible (*i.e.* reachable by the robot) from the robot's starting configuration to the grasp at the object pick-up pose;
2. there exists a stable grasp position in which the object is not free to twist or slip with respect of the gripper once this last has performed the gripping;

3. the grasp must be collision-free, that is no part of the robot is in collision with any obstacle at either the object's pick-up or put-down poses;
4. the grasp must be kinematically feasible at both the object's pick-up and put-down poses;
5. there exists a collision-free path kinematically feasible from the object's pick-up to its put-down pose for the robot holding the object with the chosen grasp;
6. there exists a collision-free path kinematically feasible from the object's put-down pose to the robot final configuration which may be regarded as the initial configuration for the next operation.

The main difficulty in satisfying all of these constraints on a *pick-and-place* operation arises from the interaction among the steps. A grasp, for instance, must be planned so that no collisions arise and the required paths are feasible. Handey tries to deal in detail with most of these interacting constraints when a grasp is planned. However, while the constraints of collision avoidance and kinematic feasibility at both pick-up and put-down pose are guaranteed, some other constraints are handled only approximately, such as, for instance, guaranteeing that a chosen grasp does not interfere with finding collision-free paths for the rest of the operation.

In order to deal with as many of the aforementioned constraints as possible, the architecture of the system has been divided into four nearly independent planners:

- gross motion planner,
- grasp planner,
- regrasp planner, and
- multi-arm coordinator.

Handey breaks down the planning of a *pick-and-place* operation in several steps and each planner is responsible just for a limited subset of them. When a *pick-and-place* is initiated, the robot is in some starting configuration. The gross motion planner plans a motion which takes the robot from such an initial state to an approach configuration

chosen by the grasp planner, which at the same time plans first a grasp configuration reachable by the robot at both pick-up and put-down pose, and second the path between the approach and grasp configurations. Once this planning phase has been completed, the gross motion planner moves the grasped object to the target put-down pose. At that point the grasp planner plans the departure configuration and a path between the put-down pose and the departure configuration. If a grasp which is consistent with both the part's pick-up and put-down poses for some reasons can not be found, then the regrasp planner is called and this generates a sequence of grasps and placements that allows the robot to place the part at its intended destination. These grasps and placements chosen by the regrasp planner may require further planning of gross motions, approach motions, and departure motions.

The planners mentioned above operate on five main data structures:

- a **Part** which is three-dimensional geometric model of an object placed in a particular pose;
- a **World** which is a three-dimensional geometric model of a scene containing one or more robots and multiple parts at a given instant in time;
- a **Robot** which is a description of the kinematic structure of an arm and contains the geometric model of its links;
- a **Goal** which can be either a specification of a set of joint angles for a given robot or the desired final pose of a part;
- a **Plan** which is finally a description of the elementary robot and gripper motions generated by the planner to achieve the goal.

When the planners are invoked, they take these data structures as arguments and in case of success they return a **plan** data structure which can be used to either simulate the plan or execute it on a real robot. In case one of the planners fails to plan the desired operation, it returns a **failure** data structure which contains a reason for the failure and eventually a list of parts in the world which may prevent the planner from achieving its goal. Other planners may use the information returned in such a data structure to generate new goals that may help to achieve the initial task.

The main property of Handey is that, as long as an assigned task remains possible, if the environment changes, the system is still able to generate new motions so as to achieve the desired goal. In other words, if for instance an obstacle is introduced, then, in order to accomplish the task, Handey generates a new grasp and new motions compatible with the new environment. The specification of any task planned by Handey is robot independent, in the sense that even changing the agent the specification does not need any reformulation.

Handey is not intended as a programming system for commercial use, but simply as a research prototype with which to analyze and study the problems involved in the development of task-level systems. Because of this, it emphasizes more the ease of development than its speed of execution. The system takes as input a complete geometric and kinematic description of a robot and its environment and produces as output a program to achieve the user-specified *pick-and-place* operation in the environment.

The programs generated by Handey are made of explicit motion commands for the robot and its gripper. Each of these commands specifies the desired value for each of the robot's joints and the desired gripper displacements. The system assumes the robot controller performs linear interpolation between successive commanded configurations of the robot and makes the agent follow the path produced in this way.

Analyzing the system, the first limitation which appears evident is that it requires as input a complete world model. However, it is possible to get around this problem by using for instance range sensing to model the environment. A more relevant limitation is instead that Handey has no direct way of coping with variations in its environment during the execution of its task. In other words, the system is unable to produce a program which can sense the environment during the execution of the *pick-and-place* operation and make decisions based on the sensing. Moreover, it does not generate compliant motions, nor guarded motions, so as to achieve robustness and reliability. Even more, it does not generate open-loop motions to exploit task mechanics in order to produce the desired result in spite of the initial uncertainty. Nevertheless, Handey as it is implemented now can accommodate one relatively simple uncertainty-reduction operation. To this end, approach, grasping and put down motions may be generated as a kind of guarded move. Future developments of the system, however, are planned

to address the limitations pointed out above.

In conclusion, the architecture of Handey, which is actually a system successfully implemented, may be considered as a possible different solution to the problem of robot task-level programming. Now let us go to examine the criticism of knowledge-based systems from another perspective.

2.2.2.2 Subsumption Architecture

Around the mid-eighties, when Brooks and Lozano-Pérez were defining the project TWAIN ([Lozano-Pérez & Brooks 85]), Brooks decided that the classic knowledge-based paradigm within which the project was being developed was not going to work. He argued that the entire classic knowledge-based approach to artificial intelligence systems was wrong because we were actually missing some basic principle behind it. He maintained that the only way of filling this lack of principles was to abandon temporarily assembly robots, which were too complicated, and start with simple small mobile robots ([Brooks 86a]). He claimed that behaviour-based principles were more suitable to artificial intelligence than knowledge-based ones, and to this end he decided to propose his subsumption architecture ([Brooks 86b]).

He argues that, since autonomous agents have to operate in a world where conditions change rapidly, the old way of solving the problem of building a real time control system for a robot is not adequate. The decomposition of the system in *functional units*, although being a natural and conceptually easy way to think about it, is inherently unable to cope with sudden and fast changes in the environment and to react promptly to them. Brooks maintains that this limitation is not due to the mechanics or the electronics of the agent but instead to the architecture of the agent control system. He proposes to replace the classic functional decomposition of the system with *task achieving behaviours* (cf. figure 2.4 on page 34).

He identifies four requirements for his architecture.

Multiple Goals. The agent has to be capable of dealing with many goals at the same time. These goals sometimes might even be conflicting with each other, like in the case when the agent is supposed to go to a certain location and to avoid an

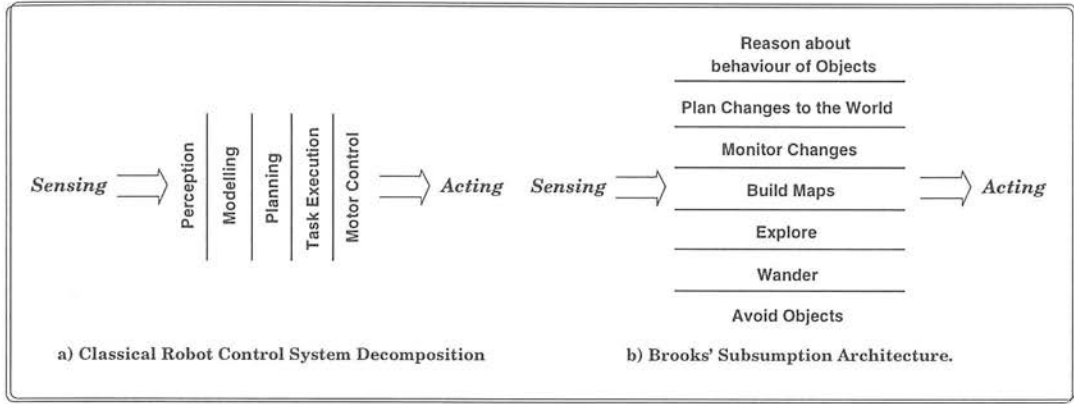


Figure 2.4: Brooks' Approach.

obstacle at the same time.

Multiple Sensors. The agent has to be capable of dealing with many kinds of sensors, whose measurements of a particular physical quantity might sometimes overlap giving in return inconsistent readings.

Robustness. The agent has to be robust in the sense that if any sensor fails, the robot has to be capable of coping with the deficiency and to carry on with its goal relying on the remaining ones still working. Even in the case of a dramatic change in the environment the agent should still be able to perform a sensible behaviour instead of aimlessly wondering around, or worse sitting on the floor without doing anything.

Additivity. The agent has to be capable of increasing its competence as new sensors and capabilities are added to the robot.

In order to satisfy the previous requirements, the behavioural system proposed by Brooks has to be based on several assumptions.

- The complexity of a behaviour is not necessarily produced by a complex control system, but it might rather be simply the reflection of a complex environment.
- Components and relative interfaces should be as simple as possible.
- Agents should be cheap and capable of carrying out useful work in a human inhabited environment with no human help.

- In order to cohabit with humans, agents need three dimensional models of the world and not just two dimensional maps of it.
- Robot maps should be relational in order to avoid large cumulative errors caused by absolute coordinate systems.
- Agents should run in a real environment rather than in a simple world artificially built.
- Agents should use vision for truly intelligent interactions.
- Agents should be robust and capable of quickly recovering from any error situation. This implies that a process of self-calibration should continuously occur inside the robot.
- Finally, agents should be self-sustaining.

As mentioned before, following the classical lines of *functional decomposition* of the control system, the information flowing from the environment to the agent, and *vice versa* from the agent to the environment, needs to pass through all the different pieces composing the system. With such an approach every piece should be completely developed in order to run the robot at all. Furthermore, any subsequent changes to any particular piece would require most of the times major redesigning work of the control system itself. Brooks rejects such a decomposition and argues that so far such an approach has produced just limited robots. He suggests replacing the classical *functional decomposition* of the control system with one in terms of the external manifestations of the agent. To this end, he defines the concept of *level of competence* which is an informal specification of a desired class of behaviours for a robot over all the environments it would encounter. A higher *level of competence* would require more specific desired class of behaviours.

According to Brooks, structuring the agent in terms of *levels of competence* allows the designer to build just one layer of the control system for each *level of competence*. In order to move up to the next higher level, the designer has simply to add a new layer on top of the existing set (cf. figure 2.5 on page 36). In this way once a level is implemented and debugged, it does not require any further future alteration. A layer which identifies

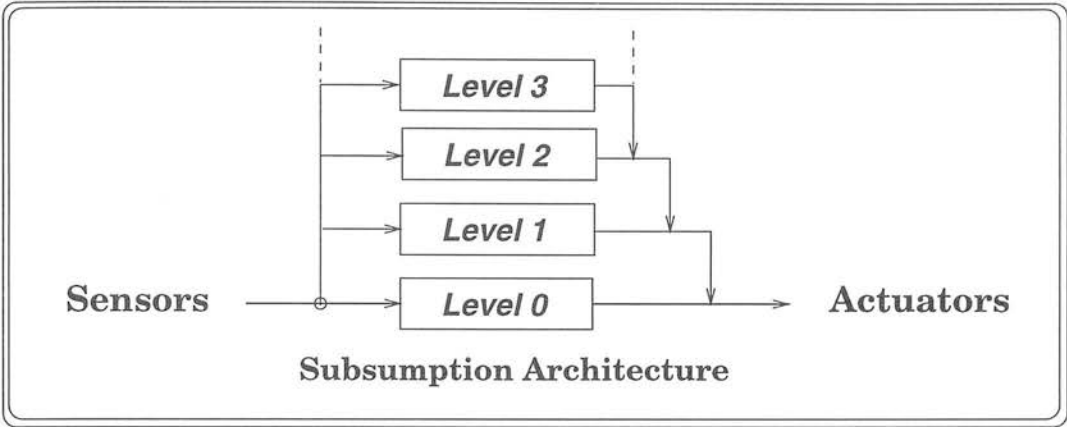


Figure 2.5: Levels of Competence in the subsumption Architecture.

a *level of competence* is made of a set of small processors which send messages *to* each other and receives messages *from* each other. Every processor, which is a finite state machine (FSM), is capable of holding some data structures and of communicating with the others by means of connecting lines. Any communication between processors is asynchronous, so no handshaking or acknowledgement is performed onto the incoming or outgoing messages. The processors within a layer which Brooks calls *modules* are not subject to a central control. This means that each processor has to perform its simple task as best as it can.

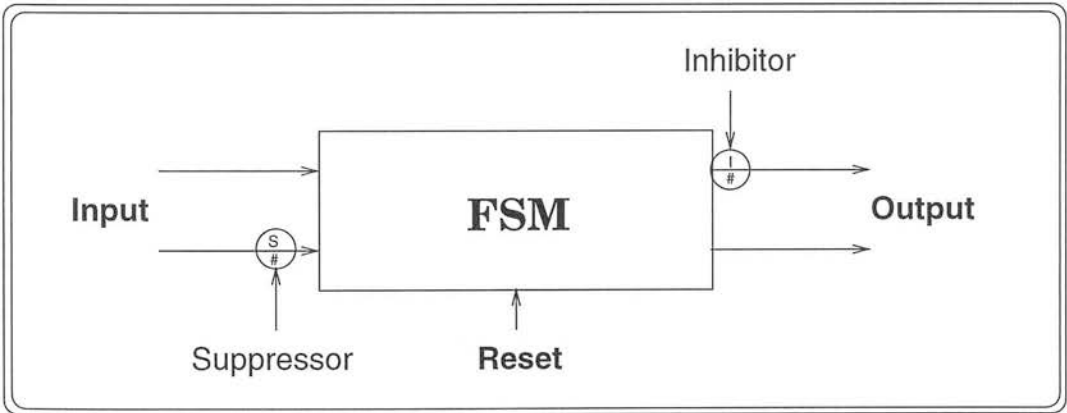


Figure 2.6: Brooks' Finite State Machine.

Each FSM has input lines and output lines (cf. figure 2.6 above). The former are equipped with single element buffers which are capable of holding the most recently

arrived message. In the case the processor is unable to read a message at its input lines before a new one would come, then the old message would be lost. Each output line from a FSM is connected to the corresponding input line of another FSM.

FSMs allows the facility to inhibit outputs or to suppress inputs. This is achieved by connecting the output line of the inhibiting processor to the output of the inhibited FSM and by connecting the output of the suppressing processor to the input of the suppressed FSM. If more suppressing lines are used, they are logically summed by an OR port into a suppressing line.

Such an architecture has been successfully implemented in several mobile robot prototypes. In [Noreils & Chatila 89] a hierarchic control system architecture for a mobile robot based on three types of entities (modules, processes, and functional units) is presented. The control system basically is decomposed in an executive unit which manages the robot resources and in a surveillance unit which manages instead the detection and reaction to asynchronous events. In [Lim *et al.* 91] the operation of a mobile robot in a real environment is decomposed in three modes (*reflexive*, *deliberate action*, and *self-awareness*). The first one is active when the agent has to react quickly to changes in the environment, the second one is active when the robot is carrying out more deliberate actions such as going from one room to another, and the third one is instead always active in order to constantly monitor what the agent can and cannot do given its current internal conditions. In [Nilsson & Nielsen 92] an intermediate software level intended for application programming is introduced between the low-level robot motion control and the high-level task commands.

All these architectures share the characteristic of having a distributed control, though such a key element is not very suitable for the assembly robotics domain, because the kind of problems an assembly robot has to face are inherently very different from those encountered in mobile robotics. A manipulator has to know in advance what it has to assemble and manipulate, in other words it has to follow a plan. The approach which we are going to discuss next faces this issue. On the one hand it criticizes the classic approach from a different perspective, and on the other it proposes a solution to the problem of automatic task-level programming in assembly different from both the classic and Handey architectures.

2.2.2.3 Behaviour-Based Assembly

As mentioned in subsection 2.2.1 on page 21, the approach followed up to the mid-eighties in order to develop automated task-level programming systems was to let the planner synthesize the plan for accomplishing the desired task in terms of the robot motions needed to achieve it (classic approach). Its application to assembly robotics led to many successful parts of task-level systems, mainly in areas such as grasp planning ([Blake & Taylor 93], [Joukhandar *et al.* 94], [Taylor *et al.* 94]), collision free planning ([Latombe 91]), fine motion planning ([Mason 84], [Clegg *et al.* 93]), gross motion planning ([Hwang & Ahuja 92]). The solutions proposed by these systems were most of the time so complex that in many cases the progress consisted merely of showing that a simplified abstraction of the problem was tractable. The limiting key factor was the presence of uncertainty embedded in the robot environment both in the location of objects and in their form ([Laugier 88b], [Whitney & Gilbert 93]). Since the complexity of dealing with uncertainty was so great, the large majority of these systems were tacitly assuming to solve first the assembly problem in an ideal and certain form and then later adapting such a solution to cope with assembly uncertainties (cf. RAPT system on page 18). Unfortunately, incorporating uncertainty handling in a plan which applies only to an ideal world is not so straightforward. Indeed, such an operation increases the complexity of plan generation so much that at the moment it is still unclear whether the problem is fundamentally intractable or just beyond the scope of current technology.

Because of these reasons no automatic robot programming system has been able so far to generate reliable assembly plans and execute them without ever needing any human intervention. The crucial point stressed by Malcolm and Smithers in [Malcolm & Smithers 88] was that it is not possible in an uncertain world to translate deterministically the motions of the objects into the robot motions needed to accomplish them. They claimed that a better way to generate more reliable assembly plans was to incorporate uncertainty from the beginning. This means that an assembly plan would be expressed in terms of part motions instead of robot motions, to be determined by the planner at *run-time* instead of the usual *plan-time* ([Petropoulakis & Malcolm 90]). This sensibly reduces the complexity of plan gener-

ation, because it is inherently simpler to control perturbations at *run-time* than to compile their control in advance at *plan-time*.

The architecture which incorporates these characteristics takes the name of behaviour-based assembly ([Malcolm & Smithers 88] and [Malcolm *et al.* 89]). The term *behaviour* is actually borrowed as a concept from an earlier work of Brooks ([Brooks 86b]) who suggested replacing the *functional* decomposition of the robot control system with a *behavioural* one, in other words to express a robot task in terms of task-achieving activities in the world (cf. subsection 2.2.2.2 on page 33). Following this line, Malcolm and Smithers consider as a primitive entity of their architecture any purposeful activity carried out by the agent.

Malcolm and Smithers' behaviour-based assembly system, although agreeing in principle with Brooks' criticism of the classic knowledge-based approach, applies to a much more constrained and rigid experimental domain than the mobile robot world of Brooks' subsumption architecture. An assembly robot requires to know in advance what to do in order to assemble parts together, besides any assembly task in order to be accomplished is inherently bound to be described at robot level in terms of sequential steps. Thus, a planner is basically still needed in a behaviour-based assembly system. This kind of drawback is not experienced by autonomous agents which do not necessarily need to follow any plan at the low level of robot control. Once an agent is for instance given the goal of reaching a certain point in a room, it does not require to follow a detailed plan to reach the target, in the sense that several modules running in parallel and without any central control may allow the agent to reach the desired goal location. The peculiarity of having a behaviour-based agent and a symbolic planner makes this architecture a hybrid between a task-level programming system and a behaviour-based execution system.

The paradigm introduced by Malcolm and Smithers' architecture allows the simplification of both the definition of robot programming languages and the design of planners which are finally freed from the inconvenience of dealing with an uncertain world. However, their system differs from a classic one because confining uncertainty within low level behaviours allows a symbolic planner to deal just with an ideal world. In this way the task planner is practically freed from the overwhelming work of generating

robot actions for coping with the various uncertainties of the robot environment.

In order to understand this point, let us examine the following example. Let us consider a robot equipped with a vision system (cf. [Chongstitvatana 92]) for locating and grasping objects (cf. figure 2.7 here below). Let us suppose we have an object laying

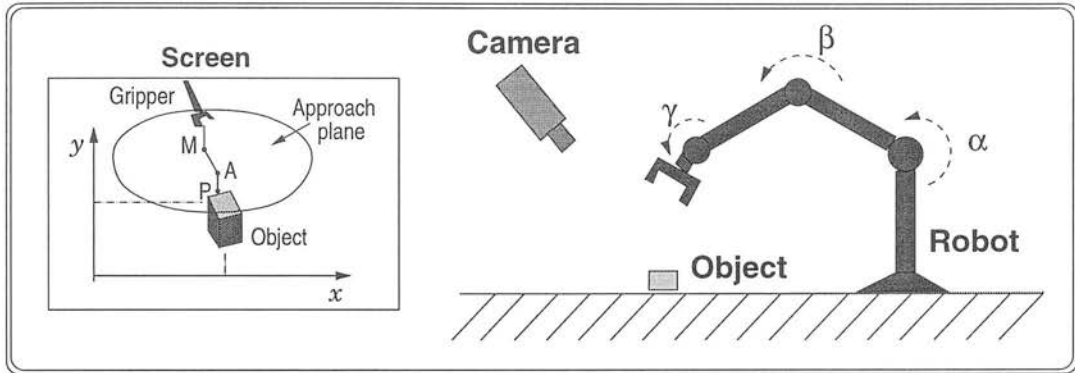


Figure 2.7: Behaviour-Based System example.

on the table and we want the robot to collect and manipulate it. Using the classic approach, the task planner needs to know

- the position of the camera with its relative parameters (*i.e.* focal length, aperture, etc.),
- the robot position,
- the robot kinematic model.

The success in the execution of the task depends heavily on the truth of this knowledge: if the environment changes in such a way as to invalidate some of this knowledge, such as by an accidental movement of the camera, then the execution of the task is very likely to fail.

Using instead the behaviour-based assembly approach, we do not need to know any of the above listed information. The basic idea is to have both the object to be grasped, and the robot gripper, in the camera images. From the object image can be deduced where the gripper image should be in order to make a successful grasp. The difference between this image position and orientation, and the actual position and orientation of

the gripper in the image, provides an error trajectory or spatial transformation (x, y, θ) in terms of image pixels. The system already has approximate linear mappings between joint angle displacements and changes in image position, derived either from observing some initial calibration movements, or from a previous grasp operation. Given these two mappings (from the two images) a robot movement is calculated to take it (for example) $\frac{3}{4}$ of the way to the destination. A new pair of pictures is taken, and the process repeated. Thus, by iterative approximation the gripper will finally arrive at the correct location for grasping, since when both images seem correct, the gripper will be in the correct location.

Given that grasps in this domain require an accuracy of location of about 2 mm, and in a typical image of 512x512 pixels a pixel represents 1 mm, and the mislocation of the part is less than 25 mm, then 4 or 5 iterations usually suffice to bring the gripper to the correct location. As implemented, the detail of the process is as follows ([Chongstitvatana & Conkie 92b]).

The robot gripper is first identified by taking one picture with its fingers opened and one with fingers closed, and then by subtracting the latter from the former so that to locate the gripper fingers' position in the image plane ([Conkie & Chongstitvatana 90]). Once this has been done, an approach plane parallel to the work surface is defined above the tops of the parts at a known height. The mid-point between the two fingers of the gripper (M) and the mid-point of the top edge of the part in the image (P) are projected onto the approach plane which is then mapped 1-to-1 to the image plane (cf. figure 2.7). Indicating with A the projection of P onto such a plane, the robot gripper is then driven vertically down so that its fingertips lie onto such a plane. Once this has been accomplished, the resulting pixel displacement allow to deduce the approach point in the image plane. The gripper is then moved horizontally onto the approach plane by tracking its motion on the image plane. This is achieved by iterative approximations of the transformation \mathbf{T} , which maps the robot coordinate frame to the image plane ($\vec{v} = \mathbf{T}\vec{m}$). Starting with an initial estimate of \mathbf{T} and \vec{v} (\mathbf{T}_i and \vec{v}_i respectively), we can compute an estimate of the motion \vec{m}_i required to drive the manipulator to its target simply by calculating $\vec{m}_i = \mathbf{T}_i^{-1}\vec{v}_i$. Using such an estimate \vec{m}_i , the robot moves to a point which, observed on the screen, is

given by $\vec{v}'_i = \mathbf{T}\vec{m}_i$. At this point, considering the difference between computed and observed movement ($\vec{v}_{i+1} = \vec{v}_i - \vec{v}'_i = (\mathbf{T} - \mathbf{T}_i)\vec{m}_i$), and a new estimate of \mathbf{T} , we can then calculate the next estimate \vec{m}_{i+1} of the required motion as shown before. The iterations, and therefore the robot motions, stop when the difference $\vec{v}_i - \vec{v}'_i$ gets smaller than a certain preset threshold. At that point the gripper can be considered to have reached the target approach point A, thus it rotates in order to get parallel to the edge of the part and then it approaches the part in the same way as it approached A ([Chongstitvatana & Conkie 92a]).

As we can see, the solution to our initial task is mathematically very simple, and besides if the world somehow changes, such as if the camera is accidentally moved or the robot arm is bent, then the task would still be successfully accomplished. From this example we gather how fragile and complex is the solution to the initial problem obtained following the classic knowledge-based paradigm, and at the same time how robust and simple is the one given in terms of the behaviour-based system.

The fragility of the classical system is due to the knowledge used: if it fails to be true, the system fails. The robustness of the behaviour-based approach is due to its avoidance of knowledge: what it does not need to know can vary without affecting the success of the task. Indeed, advantage can be taken of this, for example in moving the cameras under computer control to optimize the view, without having to worry about where the cameras end up.

It is interesting to compare this solution to how Handey performs its pick up operations. Any object in Handey is localized using a depth map which is a two-dimensional array whose indices code for position and whose elements represent the height of a surface at that location. Each depth map is determined by a triangulation-based laser range-finder. This way of sensing gives detailed information about the object to be picked up, but it is good only for static pictures of the world. This is suitable for Handey because the target object is static in the scene and the approaching operation is planned in advance, but it is completely unsuitable for real time driving of the robot gripper as in the behaviour-based assembly example given above. However, after the gross motion planner has planned an approaching path to the target object, the grasp planner detects and maps to a grasp plane any potential collisions between the surfaces coded in the

depth map and the various gripper components. This is accomplished by testing each point of the depth map within the footprint⁹ of a grasp volume of a gripper component. This stage is not required by the behaviour-based example above, because there is no advance planning: the gripper is actively driven towards its goal by its vision system and not by its knowledge about the work-cell. In the end, once potential obstacles have been mapped to the grasp plane, the Handey grasp planner computes an appropriate plan to grasp the target object and depart towards its final destination.

As we can gather from the comparison above, Handey is less bound to the environment layout than a classic system, in fact if new obstacles are introduced but the pick up operation of the target object is still possible, then the pick up can still be performed successfully by a run time grasp replanning. Nevertheless, Handey, as opposed to a behaviour-based design, still relies on the knowledge of work-cell details, and therefore it is still constrained by the truth of this knowledge.

In conclusion, a behaviour-based architecture looks even more promising than Handey in producing robust and reliable systems. For this reason, since our work deals with assembly tasks, it seems more appropriate to develop our research framework within this paradigm.

Besides the work mentioned in the example above, another interesting research project carried out along the line of the behaviour-based assembly paradigm is reported in [Deacon & Malcolm 94]. In such a work the fundamental assumption of having a point-to-point motion as the robot elementary movement is criticized, because it implies any task to be decomposed in an ordered sequence of point-to-point steps in order to be accomplished. Deacon claims that this is not the only way to view an assembly task. A different assumption, which considers a compliant *pushing* motion as the robot elementary movement, is in fact currently being investigated. Adopting such an assumption would mean that we end up with very different robots programmed in a very different way. If we take a task, such as the famous *peg-in-hole*, and we employ a point-to-point robot to carry it out, we would need to write a large amount of code with lots of sensory processing. If we take the same task and we employ instead a compliant *pushing* robot, the code and the sensory processing needed would be greatly

⁹ A footprint of a grasp volume is its projection onto the horizontal plane of the depth map.

reduced, because the robot would naturally adapt itself to the environment. In other words, if we consider for instance once again the example of the *peg-in-hole* task and we suppose that there is a tiny error in the information about the position of the hole, then a point-to-point robot would try to deform the hole in the attempt to follow the direction it was ordered to go. Under the same assumptions a compliant *pushing* robot would not behave the same way, because it would follow the constraints dictated by the environment, that is it would just push in a certain direction until a constraint would force it to slide along the surface of the obstacle. We know that with such a basic assumption about the robot elementary movement the *peg-in-hole* task works successfully (cf. [Deacon 95]). What we do not know as yet is if an assembly robot programming system can be erected on this foundation¹⁰: the only thing we may say is that human behaviour would seem to be closer to this assumption rather than to the point-to-point one.

At this point before carrying on with the rest of the discussion, it is worth pointing out a few arguments which criticize the behavioural approach as a whole. To start with, we have to stress that it is still very hard to find explicit published criticisms of the behavioural approach, especially in assembly, but there are three points which are commonly raised verbally at presentations, workshops and conferences. First of all, the lack of underlying principles. In this regard, we have to say that several projects are currently investigating the issue, and there is much general agreement about the guidelines to be followed in implementing behaviour-based systems, but no sound basis for these has emerged so far. A second point commonly raised is the fact that the behavioural approach will fail to scale up and tackle the complexities of big problems, such as those involving large robot systems with many sensors. In this respect, we have to remark that a few research projects have acknowledged the weakness and are currently investigating the issue ([Brooks & Stein 93], [Steels 94a], [Kim 96]). Another point often raised is the fact that behavioural implementations are sometimes very similar to those done by workers who do not follow behavioral principles. In this regard, we have to observe that this criticism is very short-sighted because it misses the point being made by the behavioral approach. When the first

¹⁰ A research project sponsored by the ACME-SERC grant n° GR/J94372 and conducted by Deacon himself is currently investigating this.

graphic operating systems like the MacintoshTM appeared on the market, they too were largely criticized as being just graphic gadgets stealing part of the computational power of the computer. However, the main point of the philosophy proposed by the Macintosh was the fact that it was possible to hide the complexities of the underlying operating system simply by manipulating graphic icons with the aid of a mouse, allowing in this way even inexperienced programmers to work readily on a computer. Thus, in analogy, the fact that sometimes the solutions developed following the behavioural approach are similar to the classic ones should not be viewed as a weakness, because behavioural solutions are simply providing the operator with a simpler way to program robots.

Now, there is an important issue regarding the behaviour-based assembly approach which requires to be discussed: how many *behaviours* do we need in order to have a usefully programmable system? This is a crucial point because only the existence of a limited number of them would mean that a general purpose robot capable of doing anything programmable exists. The next section will examine this issue in depth.

2.3 Behavioural Decomposition

The interesting question which we were left with at the end of subsection 2.2.2.3 is one of the key problems in the development of behaviour-based assembly architectures. Basically, if a system has to work relying just on behaviours, then how many of them do we need in order to have a working system? If the answer is, for instance, thousands of them, then it would be very difficult to develop and build a general-purpose task-level robot programming system following this paradigm which would be able to deal with at least 95% of the manufacturing assembly tasks (cf. subsection 2.3.1 on page 52 for a detailed discussion). If instead the answer is just a few dozen then it would be easier to develop a behavioural system which is practical to learn and use by somebody who is not computer literate, such as a factory floor worker.

An issue such as the investigation of the existence of basic elements in a certain domain is not new to science and certainly is not confined to the behaviour-based assembly paradigm. For instance, we can draw an analogy with two other fields: chemistry and computer science. In the former this issue consisted in determining the basic

elements which all matter in nature was made up of, whereas in the latter it consisted in individuating the basic operations with which to solve any computable problem. In the case of chemistry, scientists isolated one by one many elements and found out that beyond a certain atomic weight atoms became unstable. The result of their research was that the number of chemical elements, which are stable in nature, was limited and small. In the case of computer science, instead, the notion of *Turing machine* was introduced ([Turing 37]). Such a machine is an idealized computing device characterized by having:

- an *input/output* device, usually a tape of unlimited length;
- a *read/write* head which can read symbols from the tape and write them onto it;
- the possibility of moving the *read/write* head along the tape in either directions;
- a set of instructions, called *machine table* which completely determine the movement of the *read/write* head in conjunction with the symbols on the tape;
- a finite number of *logical states* into which the machine can pass;
- a finite alphabet of *input/output* symbols.

Thanks to such a device, Church and Turing hypothesized that any computable procedure can be modeled by a *Turing machine* table, and proved that there exists a class of *universal Turing machines* which can imitate any other *Turing machine*. From such a hypothesis and their result it follows that it is possible to describe devices which are computationally powerful enough to perform any formal operations whatsoever. The practical result which follows is that any programming language needs just a limited set of commands to process whatever procedure is computable.

By analogy, we aim to reach a similar practical result in the behaviour-based assembly paradigm. The following three subsections will provide first the grounds onto which the research presented in this thesis is based (cf. subsections 2.3.1 and 2.3.2), and then will explore how this problem has been tackled in the relevant literature (cf. subsection 2.3.3).

2.3.1 Assembly Operations

We know that the kind of tasks encountered in manufacturing industry are basically fabrication and shaping of materials, and joining and assembly of parts into final products (cf. [Owen 85]). The process of assembly may be regarded as the fixing or forming of discrete items to and around each other. There are very few sectors in industry which do not require at least one joining process, and these are arc and spot welding, mechanical fasteners, and adhesive bonding.

One of the first systematic investigations of the most common manufacturing tasks was carried out in [Kondoleon 76] where, by taking apart and reassembling a set of six products (a refrigerator compressor, an electric jigsaw, an induction electric motor, a toaster oven, a bicycle brake, and an automobile alternator), it was found that the items examined could be assembled with various combinations of just 12 operations (cf. figure 2.8 here below):

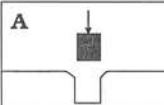
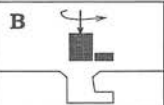
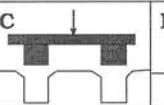

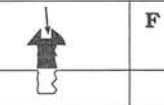
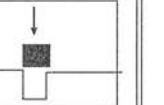
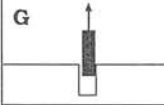
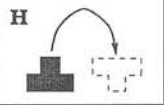
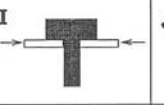
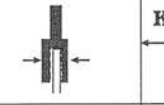
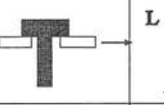
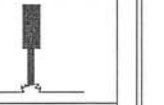
					
Simple Peg in Hole	Push and Twist	Multiple Peg in Hole	Insert Peg and Retainer	Screw	Force Fit
					
Remove Location Peg	Flip Part Over	Provide Temporary Support	Crimp Metal Sheet	Remove Temporary Support	Weld or Solder

Figure 2.8: Basic Manufacturing Assembly Operations Identified by Kondoleon in 1976.

- A simple peg-in-hole which consists in inserting a part with a shaft (peg) into a hole;
- B push-and-twist which consists first in mating a peg into a hole like A and then in securing it to the assembly by twisting the peg usually by one quarter;
- C multiple peg-in-hole which consists in matching a male part with several pegs with a female part with at least an equivalent number of holes,

as for instance a plug into a socket;

D insert peg and retainer which consists first in inserting a peg like in **A** and then in securing it by inserting a retainer;

E screw which consists in fastening a threaded peg to a matching threaded hole by contemporaneously rotating and advancing the peg along the hole axis;

F force fit which consists in inserting a peg into a slightly smaller hole;

G remove location peg which consists in extracting a peg out of its accommodating hole (reverse of **A**);

H flip part over which consists in reversing the orientation of a hole;

I provide temporary support which consists in temporary holding a part during the assembly;

J crimp metal sheet which consists in joining two metal sheets by pressing with a huge force their edges;

K remove temporary support which consists in releasing a part previously held (reverse of **I**);

L weld or solder which finally consists in joining two metal parts by fusing some soldering material along one or more of their borders.

Four of these operations (**G**, **H**, **I**, and **K**) were meant just to support the assembly process, whereas all the others were used in a variety of formats as assembly processes.

Besides identifying the basic operations mentioned above, Kondoleon analyzed in his survey also the frequency of occurrence of each of them (cf. figure 2.9 on page 49). It emerged from his statistics first that the most common operations in the manufacturing processes examined were performed vertically down, and second that the *peg-in-hole* and *screwing* operations outnumbered all the others by a large factor.

In conclusion the main result which Kondoleon achieved in his survey was to show that a group of completely different industrial products could be assembled by resorting to a limited number of basic operations.

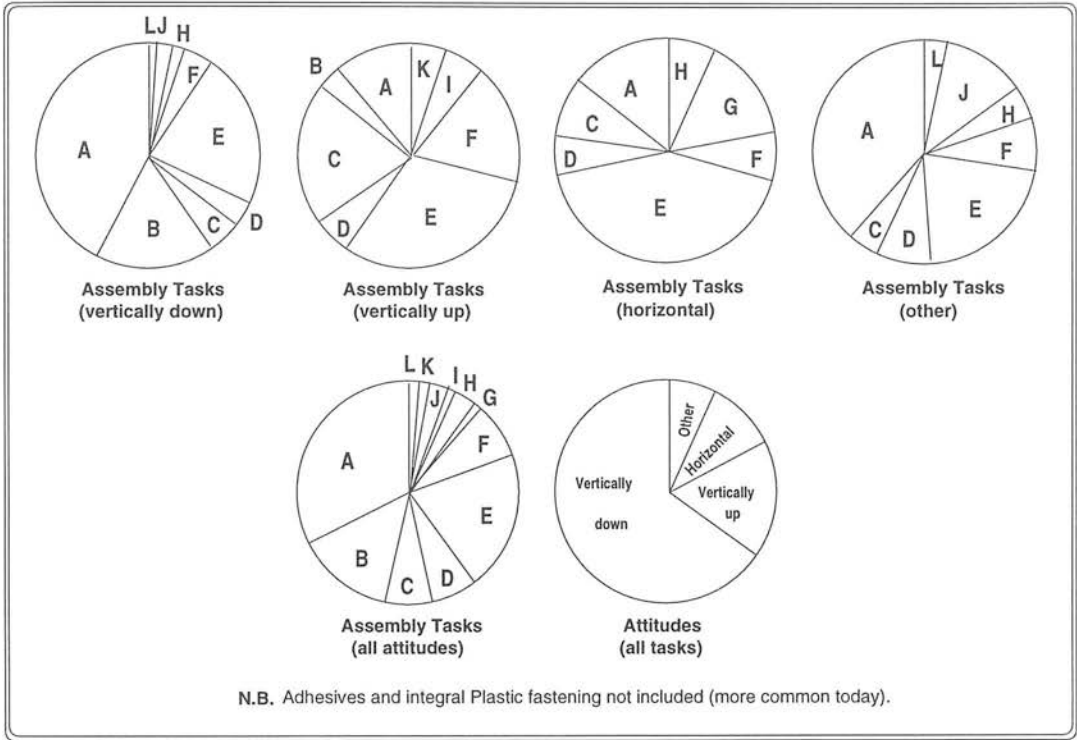


Figure 2.9: Qualitative Results of the Statistics emerged from Kondoleon’s Survey.

Examining now other surveys on assembly processes carried out in the relevant literature during the past two decades ([Nevins & Whitney 78], [Owen 85], [Byrne 87], [Hodges 92]), we notice that some of the operations have changed over the years following the changes in manufacturing technology, whereas some others are still today widely used. The use of plastic, for instance, was not considered a common task in the seventies, whereas today *crimping* has become an obsolete technology seldom used. At present we can divide assembly applications in two main categories ([Groover *et al.* 86]): parts mating in which two or more parts are brought in contact with each other, and parts joining in which two or more parts are first mated and then joined permanently or semipermanently so that to preserve their mating conditions.

Parts mating includes four kinds of operations:

Peg-in-Hole This task, which is the same found in Kondoleon’s survey, implies the insertion of one part (the peg) into the hole of a second part. There are two kinds of peg-in-hole tasks: round peg-in-hole and

orientation-dependent peg-in-hole. In the former the orientation of the peg with respect to the hole axis is irrelevant for the successful insertion, whereas in the latter it is very important (*e.g.* square peg in a square hole).

Hole-on-Peg This task was not contemplated by Kondoleon in his set because it was a mere variation of peg-in-hole. Nevertheless, there is a subtle but very important difference between the two of them: in order to mate the parts, hole-on-peg implies the part with the hole to be actively driven towards the peg which lies still at a known location in the work-cell, whereas peg-in-hole instead implies the peg to be actively driven towards the hole. A typical example of hole-on-peg is the placement of a bearing or gear onto a shaft.

Multiple Peg-in-Hole This task, which is reported in Kondoleon's set, is another variation of peg-in-hole and consists in matching a part with multiple pegs to a part with a corresponding number of holes. The insertions are supposed to be performed all at the same time. In order to align pegs with their matching holes, this operation requires the assembly system to be able to orient the part in all directions. A typical example of this multiple peg-in-hole is the assembly of microelectronic chips on a circuit board.

Stacking This task, which was not considered by Kondoleon, consists in laying down the different components of the assembly one on top of the other with no pins or other devices for locating the parts relative to each other.

The second category (parts joining) implies, as we pointed out earlier, not only mating two or more parts together but also joining them in order to hold the components together. This category includes the following eight tasks:

Screw Fastening This task, included in Kondoleon's set, consists in joining a part with a threaded hole to one with a matching threaded shaft (screw) by tightening the screw of the first part to the threaded hole

of the second part. Basically, it is like putting a peg in a hole by means of a helical rotation of the peg along the hole axis. As already pointed out by Kondoleon, this is still today one of the most common assembly tasks (cf. figure 2.9 on page 49).

Retaining This task is also included in Kondoleon's set and consists in holding the parts together by means of a retaining device, which may even be made of simple pins inserted in several locations of the parts involved. However, among the various retaining devices, the most common one is a ring, like a snap ring or a C-ring, that clamps onto one part to establish its relationship with another part.

Press Fitting This task, included too in Kondoleon's set, is another variation of peg-in-hole but with the difference that this time the peg is slightly larger than the hole in which it has to fit in. This kind of joining process can result in a very strong coupling of the parts involved, but it requires a substantial force in order to be achieved. Usually only hydraulic power press machines can provide such a force.

Snap Fitting This task was not considered by Kondoleon and may be viewed as a hybrid between retaining and press fitting. It still consists in mating a peg to a slightly smaller hole, but, differently from press fitting, the opposing force to the insertion is only temporary and it occurs just during the mating process. During the pressing of the parts towards each other, in fact, one or both of the parts elastically deforms to accommodate the catching elements of the parts.

Adhesive Fastening This task consists in spreading glue or another form of adhesive material onto a specific region of one part of the assembly and then mating the second part to that region. This task was not contemplated in Kondoleon's task because there were few adhesive materials in the seventies reliable enough to be considered a viable manufacturing joining task. The process of spreading the adhesive can be done by laying it down along a defined path, like in the arc welding, or at specific points, as in spot welding.

Welding This task, also included in Kondoleon's set, consists in joining



two parts by means of a welding process, which may be achieved in different ways, such as continuous arc or spot welding, soldering, brazing, or ultrasonic welding. It is still a very common operation especially in the automotive industry, however with the development of very reliable adhesive material and the overtaking of plastics it is starting to show signs of age.

Crimping This task in the context of assembly consists in deforming a portion of one part, usually a metal sheet, to fasten it to another part. A typical example is when an electrical connector is crimped (*i.e.* squeezed) to a wire. This operation was a very common joining process in the seventies, and therefore it was included in Kondoleon's set, however nowadays the overtaking of plastic in the manufacturing industry, and the use of adhesives, has made it obsolete.

Sewing This task finally is very common for joining soft or flexible parts like cloth or leather. Such a task was not included in Kondoleon's set because the assembly of rigid parts only was considered.

Notice that the above listed classification does not include remove location peg, flip part over, and provide and remove temporary support because, as Kondoleon himself pointed out, these are not assembly operations. Moreover, push and twist is also not included, because, although being a proper part mating operation, it may be viewed as another form of peg-in-hole.

From the statistical data shown in [Byrne 87] and reconfirmed in [Byrne & Hopkins 91] and [Hodges 92] we notice that 95% of the assembly tasks are nowadays made of peg-in-hole and its variations (35%), multiple peg-in-hole (8%), stacking (8%), screw fastening (27%), press fitting (7%), snap fitting (5%), and retaining (5%). Because these tasks represent the majority of the manufacturing assembly processes (cf. figure 2.10 on page 53), it would be very useful to have a robotic system capable with an appropriate programming of reliably performing all of them. In the rest of this thesis we will concentrate our attention specifically on them, thus whenever we talk about the majority of assembly tasks, we intend to refer to them.

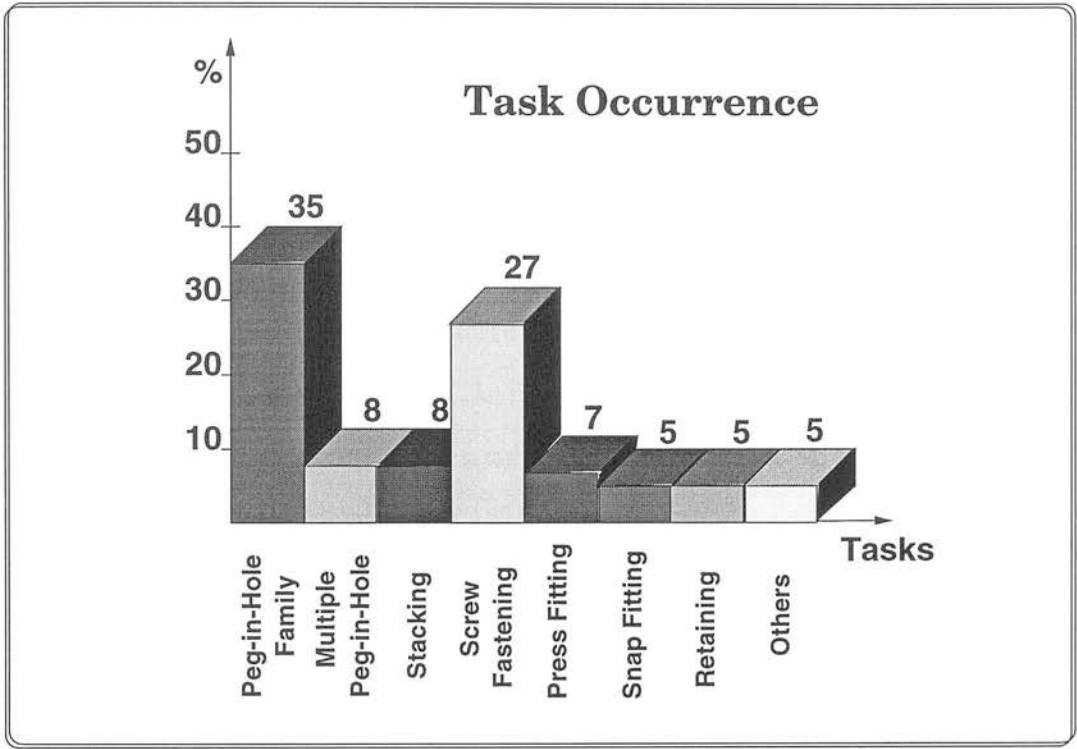


Figure 2.10: Assembly Tasks Percentage.

2.3.2 Balch’s General-Purpose Modules

A result in a certain way similar to the one achieved by Kondoleon was serendipitously reached by Balch and Malcolm during an early project conducted here at the assembly robotics laboratory of the Department of Artificial Intelligence of this University, which consisted in developing software routines general enough to handle large spatial misalignments and easy enough to be adapted to other similar assembly tasks (cf. [Balch 92b] and [Balch 92a]).

The equipment used to carry out such a research was basically made of a UMI RTX robot as the physical manipulator, a Sun3/160 workstation as its controller, and an expensive Lord 15/50 unit as a force sensor.

During the assembly of his test benchmark (an electric contactor) Balch found that the robot program required for carrying it out was largely based on a small group of 23 parameterized modules which we briefly list and comment here below.

Move-Absolute moves the robot to a fixed location.

Move-Relative moves the robot to a fixed distance from its current position.

Move-Gripper opens and closes the robot gripper.

Get-Force fetches six axes of force reading from the force sensor.

Get-Force-Vector fetches a force along a vector (x,y,z,Rx,Ry,Rz) .

Check-Force checks whether the force along a vector exceeds a threshold.

Reset-Sensor resets the force sensor by subtracting a *tare force*¹¹ from the values transmitted by the sensor.

Get-Position returns the current vector position (x,y,z,Rx,Ry,Rz) of the RTX robot gripper.

Get-Position-Vector fetches the position along vector (x,y,z,Rx,Ry,Rz) .

Guarded-Move performs a guarded absolute move.

Guarded-Move-Relative performs a guarded relative move.

Lift moves the RTX robot until the force or torque in the opposite direction to the movement are less than a threshold.

Dab performs a guarded relative move followed by a lift.

Follow drives the RTX robot in one direction while maintaining a force-torque in another one.

Hopping-Follow drives the RTX robot in one direction while *hopping* up and down in another one.

Press keeps a force-torque in one direction.

Spiral-Search drives the RTX robot along a spiral path while maintaining a force-torque in another direction.

Hopping-Spiral Search makes the RTX robot perform a spiral search by *hopping* a part along a surface rather than sliding the part on this last.

¹¹ A *tare force* is represented by the current forces and torques.

Relieve-Stress drives the RTX robot backwards and forwards until the force along a specified axis is smaller than a certain threshold.

Peg-in-Hole drives a part in one direction while relieving stresses in another one. If the part jams, it drive the robot backwards by a small step.

Contact-Position finds the intersection of the current force with a specified plane.

Put-Flat places a part flat against a surface.

Check-Lock attempts to make small movements of a part and measure the resulting forces and torques. If these exceed a threshold, then the part is locked in its socket.

Each of these modules share the same two characteristics: they all exit in just one of a small number of fixed states, and they are instantiated by means of parameters, such as the direction of movement or the maximum force. They are not intended to succeed or fail in the normal sense, rather each one is thought of as a function which returns a value according to its inputs. This idea is very important because it holds the key to generality, however Balch reaches a different conclusion. According to him, in fact, behaviours can not be as general as these modules because they have a specific purpose by definition (cf. definition 4.1 on page 85). As we will discuss in section 4.3 on page 98, this is not necessarily true, because a behaviour such as peg-in-hole, for instance, may be used not only for inserting a peg into a hole, but also for checking the presence of a step wall upon a certain surface of the assembly.

He argues that since any behaviour can be described by an opportune parameterized combination of these modules, the purpose of a behaviour would be simply acquired from the logic of how these modules are composed together. He also maintains that they are general enough to be employed as off-the-shelf software packages for describing any assembly, as if they are predefined general-purpose procedures.

He speculates that it would be possible to build on top of these modules a graphic interface to be run in a windows environment. With such an interface it would be possible to program a robot simply by composing with the aid of a mouse the icons

representing the modules into a window. The simple clicking on an icon would then enable one to see the current settings of the module parameters involved, and eventually to edit their values again.

The good point of the work developed by Balch is the fact that he managed to develop a small group of low-level software modules having disjunctive multiple purposes. However, the claim that they can be used to program assembly tasks was only partially tested on a unique testbed. Moreover, as argued in section 4.3 on page 98, the claim that behavioural modules cannot be as general as his modules is not necessarily true, because it is possible to define a behavioural module which, despite being designed with a specific purpose, can be used to achieve also a different one. For instance, a peg-in-hole module may be used both to mate a peg with a matching hole and to search for a hole with a hopping search. Balch, however, developed his routines from the bottom up, and was not specifically trying to construct behavioural modules. Our top down approach, trying to construct behavioural modules, resulted in a similar coverage of general capability with only half the number of modules.

2.3.3 Related Research

There are some other systems which share the concerns about decomposition and elementary operations in assembly without being necessarily behaviour-based systems. In this regard, an interesting work carried out in [Krogh & Sanderson 86] develops a modelling framework for representing assembly tasks as a collection of discrete operations with precedence relations reflecting the physical constraints. From such a framework it is possible to define an operation precedence graph with which to analyze alternative system configurations and supervisory control structures. By means of this graph a product¹² may be decomposed into its appropriate assembly sequence.

The interesting point of this work is the acknowledgement of the need of a limited set of descriptive primitives with which to program assemblies. However, the need of introducing into the primitives probabilistic models with which to estimate uncertainty makes the approach liable to complexity explosion.

¹² The work reports the example of an electric torch.

In [Noreils & Chatila 89] a control system architecture for a mobile robot and the structure of a behaviour-based language for accomplishing missions are developed. This architecture is organized in four modules: supervisor, executive module, surveillance manager, diagnostic module. The general frame for the monitor units composing the surveillance manager is proposed to be of the form

$$\mathbf{MNTR} \langle \text{conditions} \rangle \Rightarrow \langle \text{actions} \rangle$$

where $\langle \text{conditions} \rangle$ contains the conditions on the sensors or on the robot state variable, and $\langle \text{actions} \rangle$ contains the set of reactive actions to be carried out when the left-hand side conditions are true. Each command is defined as:

$$\begin{aligned} \langle \text{command} \rangle &::= (\langle \text{operator} \rangle \{ \langle \text{surveillance} \rangle \}^*) \\ \langle \text{surveillance} \rangle &::= \mathbf{MNTR} [\mathbf{STATIC}] \langle \text{condition} \rangle \\ &\Rightarrow \langle \text{reflex-action} \rangle \{ \langle \text{control-action} \rangle \} \end{aligned}$$

The reflex-action represents the immediate reaction to the event launched by the surveillance manager, and $\{ \langle \text{control-action} \rangle \}$ specifies the reaction through the executive module after a surveillance is triggered and the reflex-action executed. A mission is finally defined as

$$\begin{aligned} \langle \text{mission} \rangle &::= (\mathbf{MISSION} \langle \text{mission-name} \rangle (\\ &\quad [(\mathbf{PRE}: \langle \text{precondition-list} \rangle)] \\ &\quad [(\mathbf{ENV}: \langle \text{environment-list} \rangle)] \\ &\quad [(\mathbf{LSURV}: \langle \text{surveillance-list} \rangle)] \\ &\quad (\mathbf{MAIN}: \langle \text{body} \rangle))) \\ \langle \text{body} \rangle &::= \langle \text{mission} \rangle \\ &\quad | \langle \text{command} \rangle \end{aligned}$$

where $\langle \text{precondition-list} \rangle$ is the set of conditions which must be true for executing the mission, $\langle \text{environment-list} \rangle$ sets the execution environment in order to provide the controller with enough information for error recovery and local decisions, $\langle \text{surveillance-list} \rangle$ is the set of monitors which have to be active during the mission, and finally $\langle \text{body} \rangle$ is the sequence of commands to be executed.

The interesting point of this work is the definition of software units with which to program specific missions for mobile agents. However, the possibility of having more than one unit running in the system in a way similar to Brooks' subsumption architecture makes the approach not very suitable to assembly robots.

In [Nilsson & Nielsen 92] an intermediate software level for application programming between the high-level user commands and the low-level motion control of industrial manipulators is developed. Such a level is composed of two elements: an executive layer and an application layer. The former implements both the robot programming language and the executive for the execution of the robot programs. The latter contains instead the application specific control features, and it is compiled and linked together with the rest of the system.

The architecture proposed in this work, although being defined as a hierarchy of layers, still views robot application programming at a very low level. Thus, the activity of robot programming is not actually simplified, because an experienced application engineer would still be required to program or debug the software.

A new kind of architecture for programming manufacturing robots is proposed in [Archibald & Petriu 93]. The system described is based on well defined modules (robot skills) which may be viewed as a well defined object template, or as an exact coded representation of the robot behaviour. Each template is graphically represented as an icon which can be moved around the terminal screen with the mouse and can be composed in a sequence in order to program a particular behaviour. These ideas are very similar to those that led Balch to develop his general-purpose modules, which in this case are seen as iconic object templates. According to the paradigm suggested by this architecture, there are three different possible users for programming the agent: the operator who uses simple English descriptions, the robot programmer who uses the predefined iconic object templates (skills) discussed above, and the systems programmer who actually defines and prepares the icons of the various templates.

The interesting point of this work is the definition of parameterized software units capable of accomplishing specific robot actions. However, these units are viewed simply as software macros to be instantiated each time with the appropriate parameters.

Moreover, some of these units may still require models of the environment. As of today (1996), the architecture is still only partially implemented.

An interesting work, described in [Nnaji 93], presented the structure of a new task-level programming system for industrial robots named RALPH (Robot Assembly Language Planner in Hierarchy). The system, which is developed following the classic approach for controlling the agent, still requires a world model obtained by means of a CAD system. However, by fusing the knowledge of the world model and its sensor data into a dynamic world database, RALPH is able to generate world spatial relationships to be used by the planner to define sensor-based plans. The system planner is decomposed in a hierarchy of four layers:

- a task-level planner,
- a mid-level planner,
- a general robot-level planner, and
- generic robot-level planner.

The first layer accepts task-level commands as input and for each of them it produces as output the corresponding refinement in terms of mid-level commands. The second layer, in turn, takes these mid-level commands as input and generates a robot independent plan as output. The third layer adapts the plan synthesized by the upper levels to the particular robot configuration (*e.g.* SCARA, Cartesian, anthropomorphic, etc.) employed in the work-cell. The last layer, finally, takes this generic plan and instantiates it to the particular robot manipulator which has to accomplish the plan.

The interesting point of this programming system is the definition of its language. At the highest level the system provides just four commands: assemble (with), reorient, hold, and release (cf. table 2.1 on page 60). The first three of them are refined and expressed again by the task-planner in terms of a small group of nine commands: place (onto), place (over), place (beside), insert (into), screw (into), flip, turn, hold, and hold (against). At the following lower level the mid-level planner refines further the command insert (into) into one of three possible options: drop (into), drop_insert (into), and push_insert (into). Similarly, the command screw (into) is refined into either screw

Task Commands	Task-Level Planner	Mid-Level Planner
assemble (with)	place (onto)	<i>idem</i>
	place (over)	<i>idem</i>
	place (beside)	<i>idem</i>
	insert (into)	drop (into) drop_insert (into) push_insert (into)
	screw (into)	screw _r (into) screw _l (into)
reorient	flip	<i>idem</i>
	turn	turn <i>num_of_turns</i> turn <i>num_of_deg</i>
hold	hold	<i>idem</i>
	hold (against)	<i>idem</i>
release	<i>idem</i>	<i>idem</i>

Table 2.1: RALPH Task-Level Commands.

(into) (which stands for screw right) or screw_l (into) (which stands for screw left). In conclusion the total number of different commands available at the mid-level layer, which still holds task-level programming characteristics, is fourteen (cf. table 2.2 here below).

Mid-Level Planner
place (<i>feature, orientation</i>) (onto) (<i>feature</i>)
place (<i>feature, orientation</i>) (over) (<i>feature</i>)
place (<i>feature, orientation</i>) (beside) (<i>feature</i>)
drop (<i>feature, orientation</i>) (into) (<i>feature</i>)
drop_insert (<i>feature, orientation</i>) (into) (<i>feature</i>)
push_insert (<i>feature, orientation</i>) (into) (<i>feature</i>)
screw _r (<i>feature</i>) (into) (<i>feature</i>)
screw _l (<i>feature</i>) (into) (<i>feature</i>)
flip (<i>feature, direction</i>)
turn (<i>direction</i> <i>intermsof turns</i>)
turn (<i>direction</i> <i>intermsof angles</i>)
hold (<i>feature, orientation</i>)
hold (<i>feature, orientation</i>) (against) (<i>feature</i>)
release (<i>motion_flag, feature, orientation</i>)

Table 2.2: RALPH Mid-Level Planner Commands.

Since RALPH ultimately produces a plan in terms of instantiated manipulator-level commands of a specific robot system, the claim is that the task-level language employed

is capable of describing any manipulator assembly task. This is an important point which, although being achieved with a classic knowledge-based architecture, shows that the number of task-level commands required for a general-purpose robot language is limited and small (fourteen in this case).

The good point of the architecture implemented by Nnaji is the effort dedicated to defining a programming system based on a limited number of high-level commands. However, the system still heavily relies on the knowledge of models whose truth is crucial for the generation of a reliable plan.

At this point, having reviewed some interesting works which are related to our research, we are ready to define the project which addresses the problem introduced in section 2.3 on page 46. However, before doing so, it is worth summarizing what we have discussed so far in this chapter.

2.4 Summary

This chapter aimed to present and discuss three topics which are relevant to our research: the evolution of robot programming (section 2.1), robot architectures (section 2.2), and behavioural decomposition (section 2.3).

The first topic disclosed an important point: the role played by uncertainty in a typical assembly world. As we showed in detail, sensorless robots were unable to cope with all assembly uncertainties, hence sensors and programming languages incorporating their use were introduced. However, not only uncertainty was not eliminated, but robot programming did not get any easier either. This was soon realized when, by raising the level of abstraction, we passed from simple guiding and playback methods (joint-level programming) to textual languages (manipulator-level programming) which were capable of taking advantage of the eventual presence of sensory data. An interesting project discussed in subsection 2.1.4 on page 18 (RAPT) took the abstraction a step further: towards object-level programming. A program was described at such a level in terms of symbolic specifications of geometric goals. However, the incorporation of uncertainty handling in RAPT turned out to be computationally too expensive, thus the system was not acceptable to industry for commercial development. The further devel-

opment of systems at the highest level of abstraction (task-level programming) made evident that the problem of complexity explosion lay in the fundamental assumption that a task had to be described in terms of point-to-point motions. This led us to focus our attention on the issue of system design which represented the second topic of this chapter.

We dedicated section 2.2 to discuss the two main paradigms in robot architectures: the classic approach and the behavioural approach. A system designed according to the former converts a task specification into an appropriate sequence of robot motions and actions. Such a conversion, which is based on a complete knowledge of the work-cell, relies on the assumption that world uncertainty can be modeled. However, as we saw in the case of RAPT, this assumption weakens the generation of reliable plans and requires the work-cell environment to be carefully engineered with jigs and fixtures so that to limit the overall influence of uncertainty. A system designed instead according to the behavioural paradigm shifts part of the complexity from the planner to the agent. This, according to behaviour-based enthusiasts, does reduce the complexity of the overall *planner-agent* system. To this regard we presented three examples of systems which criticized in different ways the classic knowledge-based approach adopted by the great majority of the task-level programming systems: Handey (subsubsection 2.2.2.1), the subsumption architecture (subsubsection 2.2.2.2), and behaviour-based assembly (subsubsection 2.2.2.3). The first one is not actually designed following the directives of the behavioural paradigm, although it shows a clear move towards that direction. It shifts part of the complexities from the planner to the agent, but the agent is still thought of in classical terms. The subsumption architecture instead takes an extreme attitude rejecting completely the use of a planner and substituting it with a completely reactive system. Such an attitude is suitable in mobile robotics, where the environment is not so well defined and where goals can be achieved without any plan (wall following, obstacle avoidance, wandering, etc.), but it is not very appropriate for an assembly world where environment and goals are rigorously defined. The last example (behaviour-based assembly) takes a hybrid approach by shifting part of the system complexities from the planner to an agent thought of in behavioural terms, that is in terms of task achieving entities called behaviours (cf. page 39). Bearing in mind that our work deals with an assembly world, we compared the aforementioned

three architectures and noticed that the behaviour-based assembly approach produced more robust and reliable assembly systems than the other two examples (cf. page 43), thus we concluded that it was sensible to develop our research within the framework of that approach.

The third and last topic discussed in this chapter concerns the behavioural decomposition of assembly processes (section 2.3). We pointed out that by choosing to work with a behaviour-based assembly architecture we face an important issue: how many behaviours do we need in order to have a usefully programmable assembly system? The answer to this question holds the key to the success of this architecture. If it is in terms of thousands, then a behaviour-based assembly system would *a priori* fail to be accepted by industry for commercial use. If instead the answer is limited within a few dozen, then it would be possible to develop on top of these behaviours a task-level system easy to learn and use even by the computer illiterate such as a factory labour. In order to give an answer to this issue, we examined several surveys on assembly task classification (cf. subsection 2.3.1). The first and most interesting of them was due to Kondoleon (cf. [Kondoleon 76]) who found out that a sample of very diverse assemblies required a small set of just twelve operations in order to be accomplished (cf. page 47). By looking at more recent surveys (cf. [Byrne 87] and [Byrne & Hopkins 91]), we noticed that 95% of assembly tasks cluster in a set of basically seven tasks (cf. page 52). Examining then an interesting work carried out within the behaviour-based assembly paradigm by Balch (cf. [Balch 92b]) showed that a complex assembly (an electrical contactor) was accomplished by resorting to just 23 highly parameterized modules which had a limited number of exit states (cf. subsection 2.3.2). These modules do not succeed or fail as in the normal sense, but they merely return a value which codes their exit state. As a consequence they have no purpose on their own, but they acquire one from the way in which they are composed and used together. It is this characteristic which gave Balch's modules a great generality: we will expand this discussion in chapter 4. Reviewing then other related works (subsection 2.3.3), we noticed that the most interesting to us is represented by RALPH (cf. page 2.3.3). Its main feature is the decomposition of its planner in a hierarchy of four layers: task-level, mid-level, general robot-level, and generic robot-level. The system, although being designed within the classic approach to robot control, shows another important feature: the presence of a

task-level language based on a small number of four commands which are expanded at the mid-level planner to fourteen task-level commands. By resorting to such a set RALPH is capable of planning an assembly for different kinds of robot configurations.

Summarizing what we achieved in this chapter was to show that, because the design of robust and reliable task-level systems was strongly influenced by the presence of uncertainty in the world, any attempt of coping with it resulted in a complexity explosion. This motivated a step back and a reconsideration of how different architectural design could reduce such a complexity. Systems built within the behaviour-based assembly approach showed appealing characteristics of reliability and robustness, and therefore we chose such a paradigm as a framework for our research. This though raises the important issue of how many behaviours to provide it with in order to have a usefully programmable system, which in turn is linked to how many different assembly tasks we have to deal with. However, we gathered from several surveys that such a number is limited to just a few classes. Further evidence (cf. RALPH in [Nnaji 93]) also pointed out that a set of task-level commands with which to describe assembly tasks does exist. This, although achieved within the classic terms, is an important point because it shows that general-purpose task-level assembly commands do exist. What we want to investigate is if the same result holds for a behaviour-based assembly system where each command is a task-achieving entity in the assembly world.

At this point we are ready to turn our attention to define the project which will allow us to address such an investigation.

Chapter 3

Definition of the Project

As pointed out in the previous chapter, the main problem which has so far limited the exploitation of robots in manufacturing industry is how to deal with uncertainty. The behaviour-based assembly paradigm (cf. subsection 2.2.2.3 on page 38) is a possible solution to this problem, however a system based on such an architecture has to rely on behavioural modules and we do not know as yet how many of them we need in order to have a usefully programmable system. As said earlier in this thesis, if the answer to this issue is in terms of thousands of modules, then it would be very difficult both to develop and build such a system and to have it accepted by normal end-users on the factory floor, who would certainly request simplicity and ease of learning of the different commands, that in this case would be the different behaviours available. If instead the answer is just two/three dozens of commands, then it would be easy to erect on top of them a task-level robot assembly language capable, even in the presence of uncertainty, of programming reliably and robustly the great majority of assembly tasks.

In this chapter we aim on the one hand to discuss and formalize in details the issue raised above (section 3.1), and on the other to define clearly the project to address it (section 3.2).

3.1 Discussion of the Problem

As mentioned above in the introduction to this chapter, a behaviour-based assembly system requires the definition of appropriate behavioural modules. However, *how many* modules and *which* of them are vitally necessary are still open issues. These are very important points to discuss, and are linked to the problem of defining a set of primitive behaviours.

We know that, in order to program a general-purpose computer, we need only a set of a few elementary commands. However, it is so inconvenient to program in these terms that most computers provide machine codes with many more commands, let us say from 30 to 200 or so. The availability of these extra commands greatly facilitates the implementation of high-level programming languages, such as Pascal or Prolog, which are easier to learn and use for the average programmer. Similarly, it is possible that all assemblies may be accomplished by guarded move commands alone (cf. footnote on page 23), but it will surely be very tedious and error-prone to do so by using this single command alone. What we would like to find is instead a set of task-level assembly commands which are convenient for people to learn and use and with which we can reliably program the great majority of the assembly tasks (cf. page 52).

Various studies have been carried out for describing in a few commands the diverse manufacturing assembly tasks. One of the first methodic surveys showed that very different industrial products might be assembled by resorting to various combinations of just 12 basic general assembly operations (cf. subsection 2.3.1 on page 47).

In an interesting work published in [Hopkins *et al.* 88] it was shown that, although in assembly it is always possible to have a large variety of geometrically different tasks, the range of tasks that occurs in practice is considerably less. So, if these generic operations can be programmed for robots as generic modules, then the provision of a toolbox of assembly operations that encompasses the majority of industrial assembly tasks is a viable proposition which can greatly simplify the programming of assembly robot systems. A similar conclusion was also suggested in [Balch 92b] (cf. subsection 2.3.2 on page 53) where a set of highly parameterized assembly routines was introduced. However, the generality of such a set was only speculated since they were extrapolated

from just one test-bed (an electrical contactor). Balch's modules, which he called behavioural modules, could be used for a variety of purposes, depending upon which exit was taken to mean success and which failure. In other words, the specific purpose was not inherent to the routine, but was assigned by the higher level module using it. It has been argued (cf. page 55) that, since inherent purpose is a feature of a behavioural module, this kind of disjunctive purpose means that these routines are not behavioural modules, but some kind of lower routines. This is a moot point which we argue against in chapter 4. For the time being, let us just say that Balch's set is capable of describing a complex assembly.

As pointed out in section 2.2.2.3, behaviour-based assembly architectures should be able to reliably perform assemblies, because they should be robust enough to cope with uncertainty (cf. page 43). A system designed in accord with such a paradigm is based on goal-achieving entities called *behaviours*, each of which is hierarchically expressed in terms of simpler ones (cf. section 4.2 on page 95). At the beginning of this section we mentioned that we do not know as yet how many behaviours are vitally important. However, only the existence of a limited set of them capable of expressing the majority of manufacturing assembly tasks (cf. page 52) holds the key to the success of this architecture.

We know from section 2.2.2.2 on page 33 that the accomplishment of a task may be regarded as the state in which the world is left after the robot has performed an appropriate behaviour. We also know that every behaviour is hierarchically built on top of simpler ones. Every behaviour in a broad sense may be regarded as a task, thus since each of them, when performed by the agent, achieves a goal in the world, and since every task can be formulated in terms of at least one behaviour, we may say that the law associating behaviours to tasks is a surjection from the domain of the behaviours \mathbf{B} onto the codomain of the tasks $\mathbf{T}_{\mathbf{B}}$, that is in formal terms:

$$\forall \beta \in \mathbf{B} \exists t_{\beta} \in \mathbf{T}_{\mathbf{B}} : \beta \xrightarrow{\Phi_{\beta}} t_{\beta} \quad (3.1)$$

where β is a behaviour belonging to the domain of the behaviours \mathbf{B} , t_{β} is the task accomplished by β , $\mathbf{T}_{\mathbf{B}}$ is the domain of all the tasks accomplished by the behaviours in \mathbf{B} , and Φ_{β} is the function which associates β with t_{β} .

Considering this, the topic problem we want to tackle can be summarized by wondering if there exists a set of elementary behaviours with which to program the great majority of the assembly tasks. In this regard the expression *great majority* may appear rather vague, but, as already pointed out in subsection 2.3.1 on page 52, we know that 95% of the assembly tasks clusters in just seven classes. However, for the purpose of this thesis it is unnecessarily ambitious to aim for such a percentage since the remaining minority are just variations of the major tasks, or exploitations of specialized tools, and in fact we argue later that, considering the details of the chosen tasks, an 80% figure is adequate (cf. page 73). Bearing this in mind, the problem introduced above may be formulated again as:

Does there exist a finite set \mathcal{B} of N basic behaviours in terms of which it is possible to program 80% of the assembly tasks and is it possible to implement behavioural modules that realize some cases of the desired behaviour?

Giving an answer to this issue represents the goal of the research reported in this thesis.

3.2 Project

As argued at the end of the above section, the problem we are going to tackle consists of investigating if a limited set of general-purpose behaviours capable of accomplishing 80% of the assembly tasks exists. By examining this problem in detail, we notice that the key to the solution lies in two very basic questions: do there exist any behaviours which are common to different assemblies? And if they do exist, can they be generalized so that they can be used in different situations for achieving different purposes, and hence different behaviours?

These two questions are very important, and only by giving an answer to them first, we can address the problem emphasized above.

In the following subsections we will present first the project which tackles the aforementioned questions, then the aims we want to achieve with it, then again the experiments which enables to pursue such an investigation, and finally the extra hardware we require.

3.2.1 Project Description

In order to answer to the first of the two questions introduced at the beginning of the section, we need:

- to examine several assemblies,
- to analyze which behaviours are required to express them, and finally
- to refine the behaviours into their minimal terms within the behaviour-based assembly paradigm.

Once this stage is complete, we need to synthesize and group into a set the common minimal terms found above, if there are any. If the number of elements of such a set increases every time a different assembly is considered, then it would be very likely that the set is unlimited, or too big to be of any practical use, *i.e.* as a base of a behavioural language. If instead, once it has reached a certain size, the number of elements remains the same despite changing assemblies, then it would mean that the behaviours in the set are general enough to be used in many different situations.

In order to carry out the research, it would be unfeasible to consider one by one all the possible assemblies. What we need to do instead is to select a few significant assemblies and to build the set of elementary behaviours out of them. Once such a set is found, we can then extrapolate the general set of basic behaviours. To such a purpose we take advantage of the statistic about the most common assembly tasks gathered by Byrne's surveys (cf. figure 2.10 on page 53) which shows that 95% of the assemblies can be accomplished with just seven kinds of basic operations.

The definition of what is a minimal unit which may still be called behaviour is crucial because it affects the process of refining the assemblies, and hence both the size and the generality of the set of basic behaviours which we end up with. We will analyze this point in detail in chapter 4 on page 84. For the time being, let us describe the details of the work-cell which we assume to work with.

3.2.1.1 Basic Default work-cell Description

The features of a work-cell characterize the kind of jobs we can deal with. Thus, in order to be as general as possible, we have to make very plain assumptions about the work-cell itself. This guarantees a broader applicability of the results eventually gathered from our investigations.

Going in details, we can assume to work with a rather common basic set up: a robot manipulator, its relative controller, a terminal, and an end-effector (cf. figure 3.1 here below). As regards the manipulator, we assume it to be Cartesian with a SCARA

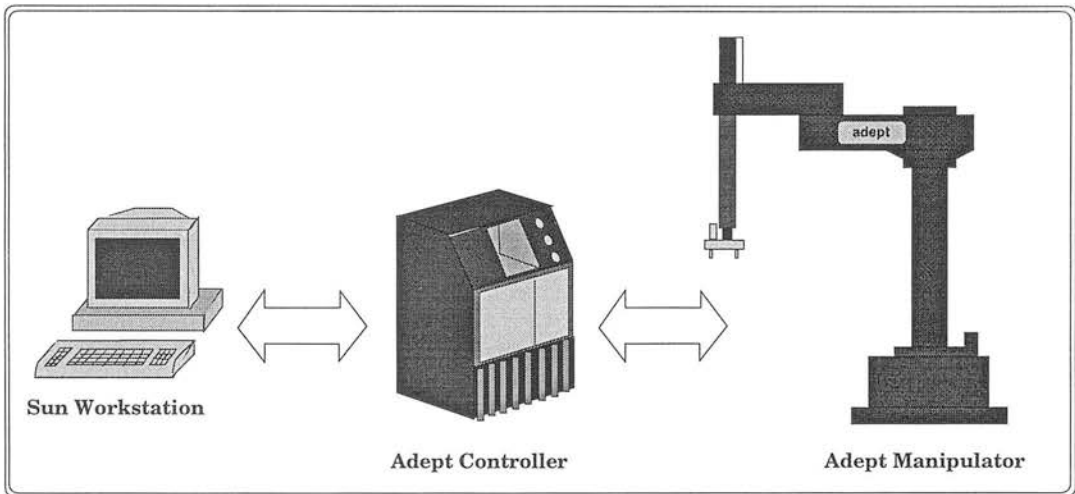


Figure 3.1: Robot System used for the project.

configuration (Adept 1) and with 5 degrees-of-freedom: the arm can any time be moved anywhere in the work-cell envelope (*i.e.* working volume) by simply specifying the Cartesian coordinates of the target point with respect to the robot frame system (cf. appendix A on page 228).

As regards the controller, we assume it to be programmed with a textual programming language (VAL II) as well as with a teach-pendant. In this way the manipulator can be moved to a target location either manually or with textual commands. In any case the position of the manipulator is defined as a 6-dimensional vector: three-dimensional space coordinates, and yaw, pitch and roll angles.

As regards the controller terminal, we assume it to be a Sun3/160 workstation, though,

in order to maintain generality, we use the workstation as a mere terminal with nothing more than what we may find on a usual robot video terminal.

As finally regards the end-effector, we have to say that a general-purpose multi-fingered gripper given its versatility would be a perfect choice, but unfortunately there are none commercially available, and indeed such an end-effector is still today the subject of extensive research ([Grupe & Henderson 86], [Mehdian & Rahnejat 94], [Reynaerts & vanBrussel 95]). Assuming to restrict the kind of objects we can deal with to prisms with parallel faces and to cylinders, we realize that a two-fingered gripper with parallel jaws would suffice our needs. In this regard, we developed an electric gripper to be mounted onto the robot wrist (cf. appendix B) directly driven by the robot controller via parallel binary output lines. Such an end-effector is capable of gripping objects by opening or closing its jaw-fingers according if the object is gripped from inside, as it may be in the case of a hollow cylinder, or from outside, as in the case of a prism. The gripper is operated by an electric motor connected to the power supply through a switchboard which allows the power to go through or to be reversed according to the digital state of two driving parallel lines coming from the controller. In this way a motor is activated only when the robot actually needs to operate it. As regards the reading of the gap between the fingers, it is accomplished by resorting to a sliding linear potentiometer mounted on one side of the gripper and attached to one of the fingers. When this last moves along the gripper, it forces the slider of the potentiometer to follow the motion. Since the voltage measured changes according to the position of the slider, and since within a certain range the potentiometer keeps a linear proportionality with the slide, it is possible to convert the voltage into a linear distance between the current position of the finger and the position corresponding to finger closed (cf. [Pettinaro & Malcolm 94] for more details).

3.2.2 Aims and Objectives

As mentioned at the beginning of section 3.2 our final goal is to determine if there exists a set of elementary behaviours with which to program assembly robots. In order to carry out such an investigation, we need to ascertain first if there are behaviours which are common to different assemblies, and then if these common behaviours may

be made general-purpose. In this regard, we described in subsection 3.2.1 the project addressing these particular points. Here, we want just to state in clear terms what results we aim to achieve at the end.

As said before, first of all we aim to give an answer to the problem of the existence of a set of elementary behaviours with which to program the great majority of assembly tasks. What we exactly mean with *great majority* is 80% of the total assembly applications (cf. statistic in subsection 2.3.1 on page 52). The reason for such a figure will become clear in subsection 3.2.3 on page 73.

The second result which we aim to reach is to identify which behaviours are comprehensive enough, versatile and convenient to be used, and easy to be composed into larger entities, in other words, which behaviours can be included in the aforementioned basic set.

Summarizing, the two aims which we want to pursue are:

- establishing the existence of a limited set of primitive behaviours with which to program most of the assembly tasks, and
- identifying the components of the above set.

In order to carry out our project, we divide it in two steps: first, analyzing the behavioural decomposition of several assemblies within the behaviour-based assembly paradigm, and then synthesizing out of them a library of convenient general-purpose behaviours. Thus, after having chosen significant assemblies and having decomposed them in terms of behavioural units, our first objective is to build up a library of behaviours. Once such a library is available, we redefine its components in more general terms discarding those which can be expressed in terms of the others. At the end of the process we have a set of general-purpose behavioural units which can be used to express different assemblies. Our second and last objective is then to determine the limits of applicability of a program expressed in terms of the elementary behaviours of the set synthesized earlier.

In conclusion we set the following objectives:

- creating a library of general-purpose behaviours out of the behavioural decompositions of a few significant assemblies, and
- determining within which limits similar assemblies may be accomplished by the same behavioural program.

At this point, before going to discuss which assemblies we want to experiment with, a remark has to be drawn about the library mentioned above: its components are hierarchically erected on top of simpler behavioural units belonging to more primitive libraries (cf. chapter 4 on page 84). The innermost core of this hierarchy of libraries *is* the set of general-purpose behaviours which we are looking for. Thus, bringing to light such a core is the goal of our research.

3.2.3 Choice of the Experiments

Having described the project and stated the aims of our research, what we are now left with is to select the assemblies which allow us to investigate the existence of a set of elementary behaviours.

As shown by Byrne's survey 95% of the assembly tasks clusters in just seven basic operations (cf. subsection 2.3.1 on page 52): peg-in-hole and its variation (35%), multiple peg-in-hole (8%), stacking (8%), screw fastening (27%), press fitting (7%), snap fitting (5%), and retaining (5%). As regards press fitting, we have to point out that it is very technology dependent, besides it always requires special equipment, such as an hydraulic press, in order to be performed. Thus, the robot acting as a mere feeder does not play a relevant role in this kind of application. As regards screw fastening, it usually requires automatic screwdriver tools in order to be accomplished, however, we assume in our investigation that the robot does not make use of any such tools. The reason for this lies in our assumption of modeling this kind of operation on the human screw fastening, which consists in firmly holding one part with the left hand and twisting the other part with the right hand. As regards then multiple peg-in-hole operations, we have to say that they are common in special applications only, like electronic microchips assembly, and they are most of the time accomplished by specialized machinery. Since we want to investigate assembly operations in which a

robot plays a more active role, we disregard press fitting and multiple peg-in-hole and concentrate our efforts on the other five (cf. list above) which account to 80% of all the assembly tasks. In this regard it is difficult to find a single assembly which requires all of them. Thus, in order to cover these five operations, we have to select different assembly experiments, each containing some of them.

To start with, let us look back at both Kondoleon and Byrne’s surveys (cf. page 47 and page 52 respectively). There we notice that one of the most frequent tasks occurring in the manufacturing assembly industry is peg-in-hole (cf. figure 2.10 on page 53). Since one of our objectives is to investigate the limits within which a behavioural program can be applied to a similar assembly task (cf. page 72), we need not just one assembly but an entire family¹ of them. To such a purpose we choose as our first experiment to assemble the two sets of pieces shown here below in figure 3.2 ([Pettinaro & Malcolm 95a]). This assembly consists basically in stacking the L-shape

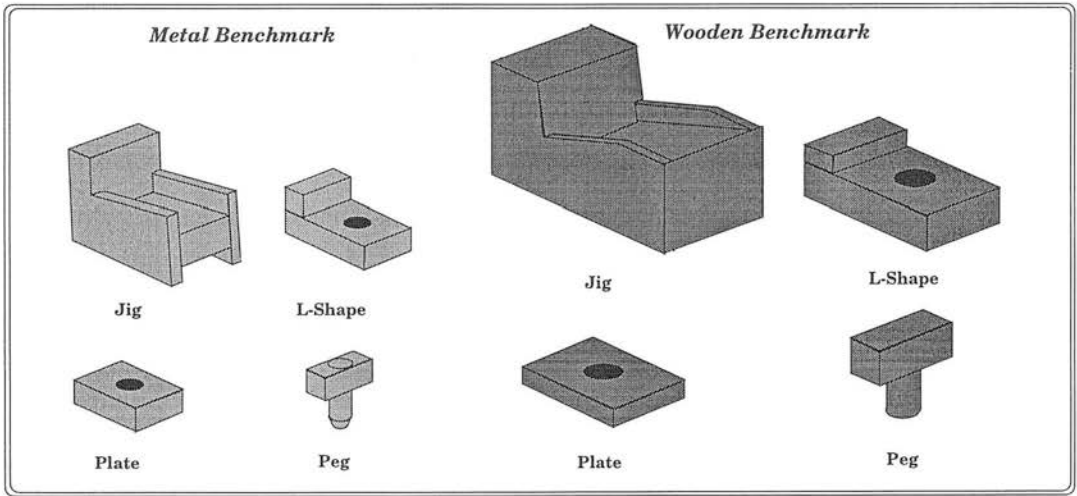


Figure 3.2: Benchmark Family Assembly.

part onto the jig, the plate onto the L-shape, and finally to insert the peg into the coaxial hole through the plate and the L-shape part (cf. figure 3.3 on page 75). Basically just two kinds of operations are involved during this assembly: stacking and peg-in-hole. According to the survey of manufacturing tasks recalled at the beginning of this subsection, they are both very common: the former occurs 8% of the times

¹ We consider two assemblies to belong to the same family if they require the same sequence of operations to be assembled.

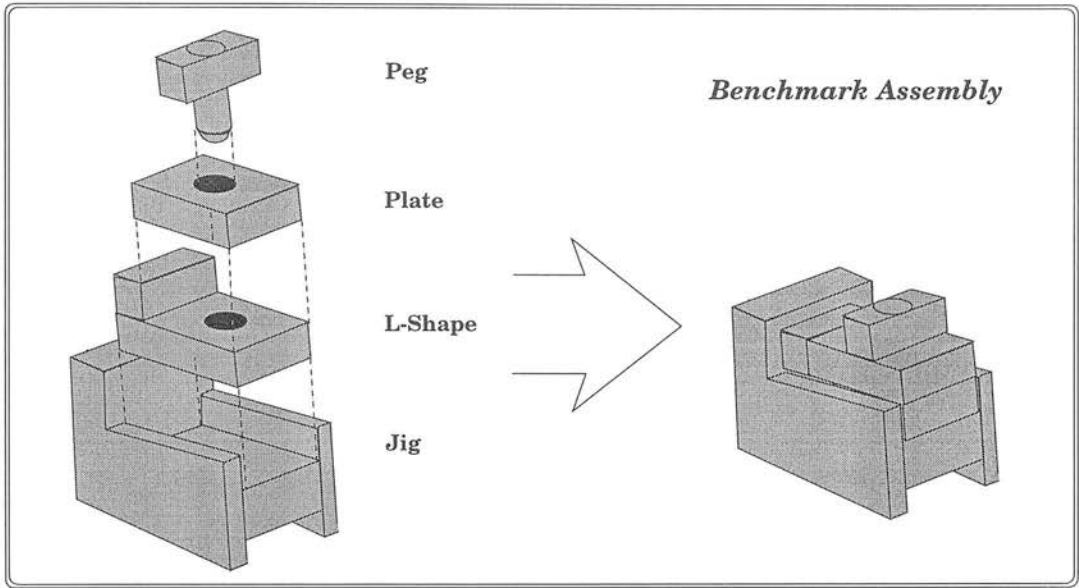


Figure 3.3: Benchmark Assembly.

and the latter 35% (cf. figure 2.9 on page 49). Thus, developing behaviours capable of performing these operations is very important.

Considering the assemblies of the two sets of parts, we notice that they clearly belong to the same family, however despite evident similarities they are different. This difference does not merely consists in different geometrical size of the parts but also in some basic features, such as the slope of the jigs, or the presence of a chamfer on the tip of the metal peg and on the edges of the wooden jig. What we want to achieve by experimenting with this family of assemblies is to study what behaviours we need to develop in order to accomplish both of them (cf. second objective of our project on page 72). A reliable and robust behavioural program capable of performing both benchmarks represents in brief the final goal of this experiment.

Another assembly operation which we are interested in is what Kondoleon calls insert-peg-with-retainer. We showed in subsection 2.3.1 on page 52 that such a task is still today quite common. It is closely related to the snapfits' family and consists in mating a male part (a peg) with a female counterpart by overcoming the resistive force of a retainer located into the female part. This is an interesting task because it requires a compliant insertion of a peg into a hole in the presence of a temporary obstacle

resisting the insertion. Since retaining is still today a common assembly operation (5%), it is worth studying it by choosing an assembly which requires such an operation. The closest one available to us was provided by the GEC MARCONI Research Centre (STRASS). Such an assembly is made of two simple parts: a jig and a peg named Strass A and Strass B, respectively (cf. figure 3.4 here below). The former, which is

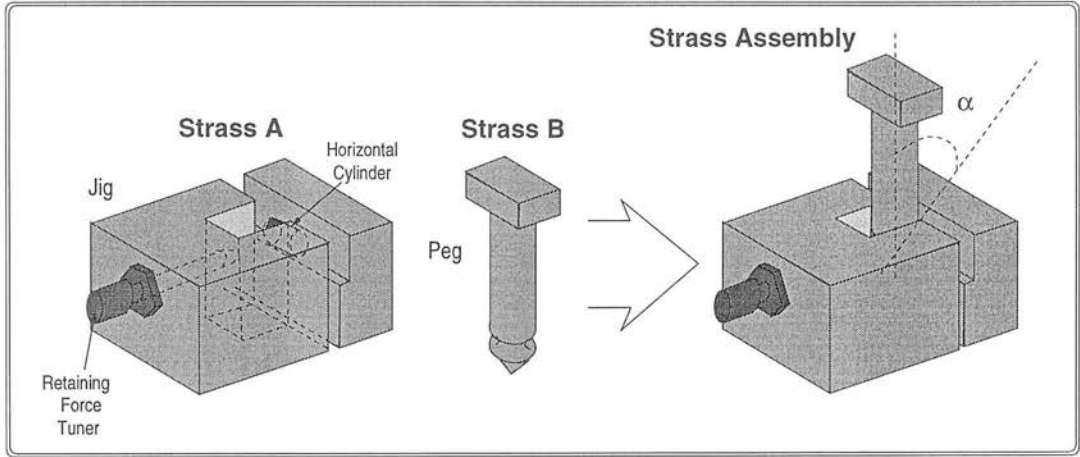


Figure 3.4: Strass Assembly.

the female part, has a spring retainer directly linked to a screw on one of its sides: the fastening of such a screw sets the amount of the retainer resistive force. The jig has also got on the other edge of the hole in a fixed location opposite the retainer a horizontal cylinder which acts as a passive retainer. Strass B is instead a chamfered peg whose chamfer matches the aforementioned cylinder. The assembly consists in mating Strass B with Strass A by overcoming the temporary resistive force of the spring retainer. Experimenting with this kind of assembly allows us to study both snapfits and retainings, each of which, according to the survey of the most common assembly operations discussed on page 52, occur 5% of the times. Because of this we choose the STRASS assembly as our second experimental test-bed.

The final task we want to investigate concerns screw fastening, which is with an occurrence rate of 27% the most common manufacturing assembly operation after peg-in-hole (cf. figure 2.10 on page 53). In order to investigate such an operation, we decided to run as our final experiment the assembly of real industrial objects: a family of three torches (cf. figure 3.5 on page 77). This is a rather special form of screwing in which

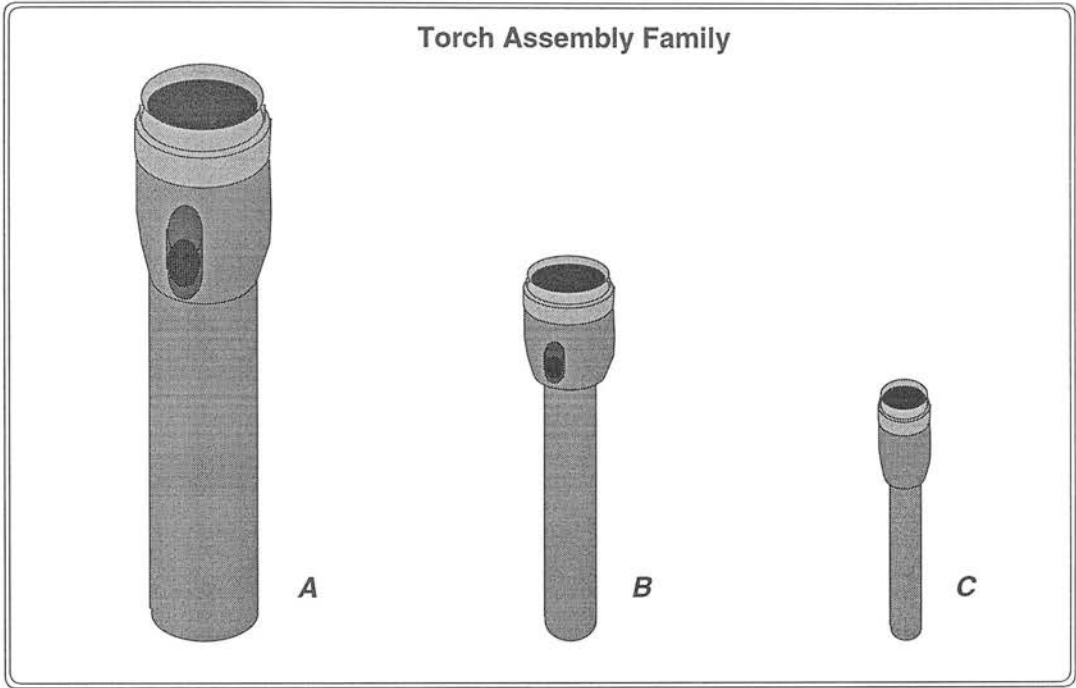


Figure 3.5: Torch Family Assembly.

the parts are joined not by screwing bolts into nuts but instead by fastening the two joining parts together. In other words, it implies male and female screwthreads to be part of the assembly.

The goal which we aim with this experiment is to perform the assembly of the entire family of torches using the same behavioural program which is developed in a bottom-up fashion by getting the best generality out of the behavioural modules needed for the accomplishment of one torch assembly. Incidentally, we have to point out that the side result we want to achieve with this experiment is to show that the assembly of a family of similar objects is not simply a matter of writing a parameterized robot program which can be scaled up or down according to the set of parts involved² but it is instead a matter of creating a description of the assembly task independent from the actual physical size of one particular set of parts.

Summarizing, the three assemblies selected in this subsection allow us to study 80% of the most common assembly operations: stacking (8%), peg-in-hole (35%), snap

² This alone in fact would not guarantee in general neither robustness nor reliability.

fitting (5%), retaining (5%), screw fastening (27%). Thus, developing a limited set of behaviours capable of performing all of them would allow to give an answer to the problem of the existence of general-purpose behaviours with which to program assemblies.

3.2.4 Extra Hardware Required

We know from subsection 2.3.1 on page 52 that the great majority of assembly tasks (95%) are nowadays made of just seven tasks. Two of these in particular (peg-in-hole and screw fastening) are by far the most common. A robot system provided with at least three degrees of freedom, with a magazine of exchangeable tool end-effectors to be loaded as required (*e.g.* a gripper, a screwdriver, etc.), and with suitable fixtures is capable of performing all of them. In this regard, assuming screw fastening to be modeled in a human-like fashion (*cf.* page 73), a simple two-fingered jaw gripper as the one we assumed to have in our work-cell (*cf.* subsubsection 3.2.1.1 on page 70) is more than enough.

Considering now the assemblies we chose in order to carry out our research, we realize that, in the case of the torch family, we are dealing with a very peculiar form of screwing, which involves the fastening of large awkward parts (*cf.* page 76). The robot of our work-cell provided with the above gripper and a suitable jig is perfectly capable of accomplishing it just with the basic hardware we assumed to have in our work-cell. However, by assuming to have a table gripper acting as a programmable vice and capable of rotating about a central vertical axis, we would not require any jig to aid the accomplishment of the screw fastening. In this regard, we have to observe that making this assumption does not limit the generality of our research. In fact, the only behaviour relying on this extra hardware would be screw fastening, and such a behaviour could always be redefined so that to make use of just the basic work-cell hardware. In conclusion, the assumption of a second table gripper (*cf.* appendix C) is very useful to have because it simplifies the implementation of screwing, but it is not essential, because what may be done with two grippers can always be done more tediously with a plain jig-gripper system. Thus, the decision of developing a second gripper as an adjustable fixture should be intended simply as an aid to perform human-

like screw fastenings (cf. subsection 3.2.3 on page 76).

Thus, without any loss of generality, we can develop a two-handed robot system (cf. figure 3.6 here below). Going in details, such a system is made of one two-fingered

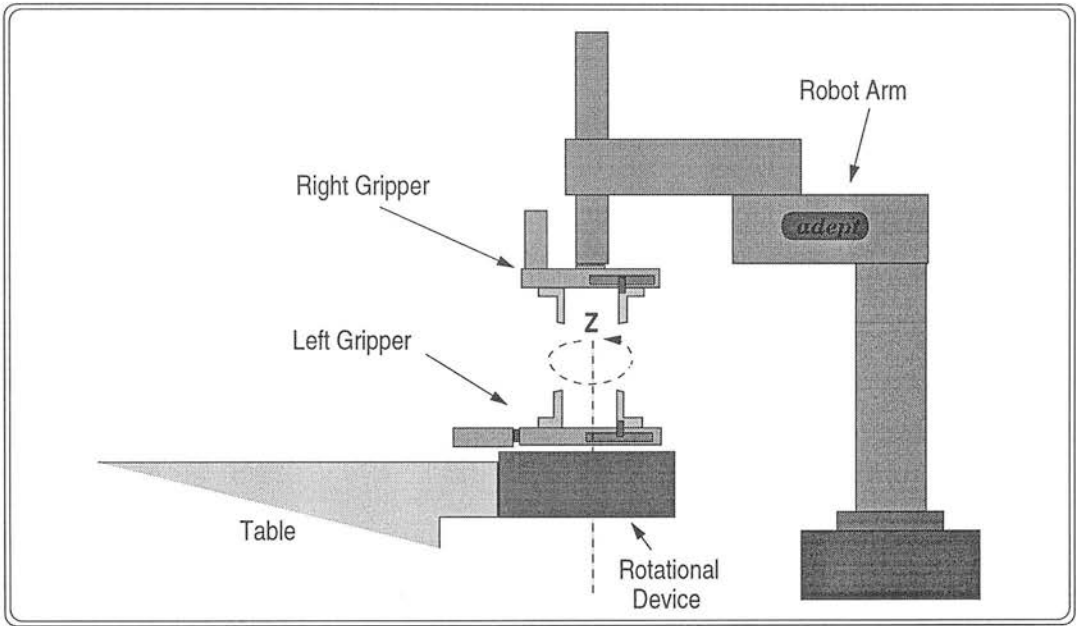


Figure 3.6: Two-Handed Robot System Diagram.

electric jaw gripper mounted onto the robot wrist, and another similar two-fingered electric jaw gripper mounted onto a turntable capable of rotating about its central vertical axis in a specific location of the work-cell. The first gripper is used as the robot end-effector (right gripper) and the second one as the robot left hand (left gripper). The two grippers and the turntable are operated each by a DC electric motor directly driven by the robot controller through a switchboard, which selects one of the above devices by means of two output binary lines. In case one of the grippers is selected, the activation of its motor allow to open or close its two fingers according to the logical state of two extra output lines. In case the turntable is selected, the activation of its motor allows to rotate clockwise or anti-clockwise the gripper mounted onto its top according to the state of the same two extra output lines. The reading of the gap between the fingers is the same for the two grippers (cf. page 71), and indeed a similar argument applies for the turntable, too. In this case, the angle rotated about the vertical axis is measured with respect to a specific initial orientation. Further details

about the two-handed robot system architecture are reported in appendix C (cf. also [Pettinaro & Malcolm 95b]).

Now, let us look back again at the operations involved by the assemblies we selected earlier (cf. subsection 3.2.3 on page 73). Considering them and all the hardware which we assumed to make use of (two-handed manipulator system), we realize that we do not need to provide the robot with any sensing capability in order to accomplish them. However, since in two cases (benchmarks and torches) our robot has to perform an entire family of assemblies, it would be of great help to provide it with a sensor which would allow to adapt its behaviour according to the particular part it is currently dealing with. This leads to the problem of choosing an appropriate sensor. Roughly speaking the simpler it is, the faster the data gained from it can be processed. Analyzing the kind of tasks which our robot has to deal with, we notice that it needs to pick up objects of different sizes from a work-cell location, move them about, and lay them down at another location. Thus, in order to enable more general positions and sizes to be handled, the environment information we require to get from a sensor should consist basically in detecting contact events. Among all the senses possessed by human beings vision is surely the most powerful one, but also the most complicated and expensive to reproduce in the artificial. Besides, many high precision cameras³ have not sufficient resolution to cope with very tiny robot moves⁴. In order to overcome these problems force-torque sensors may be employed, however, as pointed out above, these sensing devices provide far more information than we actually require out of a sensor. Considering then their relatively high cost, we conclude that they do not represent a perfect choice for us. An interesting solution to the problem of finding a suitable sensor is proposed in [Kim *et al.* 93]. It suggests to use an event signature sensor, that is a device capable of notifying events' occurrences. Physically it is made of a piezo-film wrapped around the fingers of the robot (cf. figure 3.7 on page 81). Every time a bending⁵ of the film occurs, a signal can be sent to the controller warning that a contact has taken place. In this way the two fingers may be used to detect

³ A camera may be regarded as a rough approximation of an eye.

⁴ For example, part mating typically involve motion with accuracy less than 0.1 mm, whereas cameras in typical assembly work have a resolution greater than 1 mm.

⁵ Every bending produces a voltage.

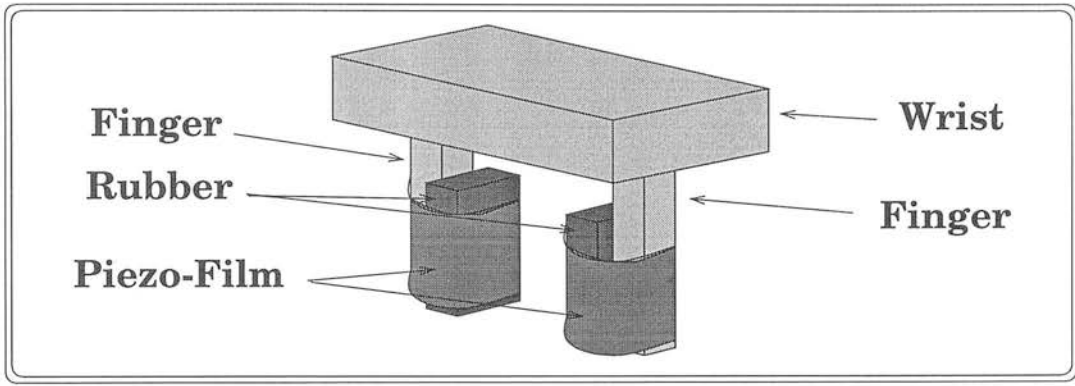


Figure 3.7: Gripper equipped with Kim's Piezo-Film Sensor.

contacts with other surfaces (cf. [Kim 96]).

Taking into account what information we need to get out of a sensor, we notice that Kim's sensor, although being very simple and crude, provides us reliably and robustly with the minimum information we require (contact detections) in a very inexpensive way, and this characteristic makes it very appealing to us. Of course, it cannot be the ultimate solution for every assembly task, but it can greatly and cheaply help in many situations. For these reasons we decided to equip the fingers of both grippers in our two-handed robot system with this kind of sensing device.

Summarizing, we can draw two important conclusions from this subsection. First, the extra hardware we assumed to make use of (two-gripped system) is very useful but not essential, because we could develop our research without it. Such a set up would affect only low-level details of the implementation of just a few software modules, thus it does not restrict the generality of our investigation. Second, the presence of two grippers introduces another set of sensors and other two active devices (left gripper and turntable) acting in parallel with the robot manipulator. This is important because it usefully extends the range of control options which behavioural modules must handle.

3.3 Summary

At this point, before carrying on reporting the rest of the thesis, it is worth briefly summarizing the main points analyzed in this chapter.

We aimed here to achieve two results:

- formalizing the topic problem to be investigated, and
- defining the appropriate project for addressing it.

As regards the former, we pointed out that a behaviour-based architecture requires several behavioural modules in order to run an assembly agent (cf. section 3.1 on page 66). The questions of how many of them we actually need and which ones are vitally necessary are linked to the fundamental issue: if there exist a set of elementary general-purpose behaviours. In this regard we drew an interesting analogy with the widely accepted Turing's thesis. According to this last a general-purpose computer can be programmed by a set of a few commands. However, it is very inconvenient to do so using them alone, thus more sophisticated and higher-level commands have been erected on top of them. In pretty much the same way any assembly robot application may be regarded as a sequence of some combinations of guarded-motions (cf. footnote on page 23) and end-effector operations. Programming robots in these terms, though, is as inconvenient as it is programming computers by using Turing's commands alone. The behaviour-based assembly approach may represent a solution to the problem of robot programming, but it may be so only if we do not need to develop a new behaviour for every new assembly the robot has to accomplish. In other words the behaviour-based assembly paradigm may be a viable solution only if there exists a set of elementary general-purpose behaviours in terms of which we can formulate more complex behaviours and therefore deal with at least 80% of the assembly tasks. Investigating this issue is what we aim to tackle in our research (cf. page 68).

As regards the second result, we raised the fundamental issue of general-purpose behaviours (cf. section 3.2 on page 68). The solution to this issue is very important in order to define a set of basic behaviours with which to perform industrially useful assembly work. Thus, we defined a project which consisted in selecting significant assemblies, developing behavioural programs to accomplish them, and refining the behaviours involved into their elementary behavioural terms. In order to pursue such a project we stated the basic hardware assumptions about the work-cell, and subsequently we discussed three topics: the clear definition of the aims of the research, the experiments to

carry it out, and the assumption about extra hardware to be employed in the work-cell. As regards the aims of our research, we agreed to achieve two results: finding out if a limited set of primitive general-purpose behaviours exists, and finding out the elements which it is made of. As regards the choice of the experiments, we decided on the grounds of the statistic of the most common assembly operations discussed on page 52 to experiment with three assemblies (cf. subsection 3.2.3): a family of benchmarks (cf. page 74), the STRASS assembly (cf. page 75), and a family of torches (cf. page 76). These three assemblies allow us to study 80% of the most common assembly operations (cf. page 73). As regards finally the extra hardware in the work-cell, we took two important decisions: first, we decided, based on the kinds of tasks which we want to deal with (cf. page 52), to provide the work-cell with a two-handed robot system made of two similar two-fingered jaw grippers (cf. page 3.2.1.1), second, we decided to equip each finger of the two grippers with Kim's touch sensor (cf. page 80) which is the simplest and cheapest sensor providing all the information we require (contact events). Incidentally, we have to point out that the assumption of having a two-handed robot system does not affect the generality of our research because the second gripper is treated simply as an adjustable fixture directly driven by the robot controller (cf. page 78).

Chapter 4

Behaviours

We know from the previous chapter that the aim of our research is to investigate the existence of a limited set of elementary behaviours with which to program reliably the great majority of assembly operations (cf. section 3.1 on page 66). In order to start such an investigation, though, we have first to finalize a few important concepts on which we will base our research: *behaviours* and *behavioural modules*, and *hierarchy of behavioural modules*. We have also to discuss another rather fundamental question about behaviours: can they be general-purpose? The answer to this issue is crucial because if general-purpose behaviours do not exist, we would not be able to find any set of elementary behavioural units with which to program assembly tasks.

This chapter aims to formalize the concepts above and to give an answer to this basic question. The definitions and comments discussed here will become useful in chapter 6 on page 189 for analyzing our experimental data. For convenience we divide the discussion in three sections: the formalization of the concepts of *behaviour* and *behavioural module* (cf. section 4.1), the definition of *hierarchy* of behavioural modules (cf. section 4.2), and the discussion of the level of generality that *behaviours* can reach (cf. section 4.3).

4.1 Behaviours as Behavioural Modules

A behaviour-based assembly system, as introduced in subsubsection 2.2.2.3 on page 38, is an architecture proposed to solve the problem of reliably programming assemblies

in the presence of uncertainty. The fundamental unit of such a system is a *behaviour* which can be thought of as an operation, or a sequence of operations, for accomplishing a purpose (cf. page 39). This definition gives just an intuitive idea of what a behaviour is. By examining the relevant literature, we notice that several other definitions have been proposed. Steels, following Powers ideas ([Powers 73]), thinks of it as a regularity in the interaction dynamics between the agent and the environment ([Steels 94b]). For instance, if we suppose to observe an agent that maintains a certain distance to a wall, then, as long as this regularity holds, we may say that the agent is performing an obstacle avoidance behaviour. Using more formal terms, we can view a behaviour as a mapping from a stimulus to a response ([Mukerjee & Mali 94]), or, in similar terms, as a relationship between enabling conditions and the goal to be achieved ([Bonarini 94]). By taking into account all these different views, we can follow behaviourists and model the basic unit of our architecture in terms of an opportune tuple:

Definition 4.1 (Behaviour) *It is an ordered 3-tuple $\beta = \langle S, A, G \rangle$ in which a set of stimuli S , and a set of actions A are mapped to a goal G .*

Considering behaviours as goal-based activities, we can discriminate three kinds of them: a goal-achieving activity which recognizes a goal only once it reaches it but does not know at the beginning how to reach it; a goal-seeking activity which is designed to seek a goal that is not explicitly represented; and a goal-directed activity which seeks the achievement of a goal by following an explicit representation of it ([McFarland & Bösner 93]). In this respect we can regard our behaviours as goal-directed activities.

Behaviours have a scope out of which they do not apply any more. For instance, a behaviour such as wall following would apply as long as there is wall to follow, or obstacle avoidance would apply as long as there are obstacles to bypass. In other words behaviours apply as long as they have a *purpose*.

As mentioned at the beginning, we know that an agent performing a behaviour accomplishes a purpose, thus the two concepts are closely related to each other, however defining a purpose is very complex. Nevertheless, we can intuitively think of it as what an agent aims to achieve ([Malcolm 93]). According to this view, a purpose is

a property intrinsic of a behaviour and it does not depend on the state of the world, nor on the stimuli triggering the behaviour, finally nor on the way in which the goal is achieved. For instance, the purpose of obstacle avoidance would be avoiding collisions regardless of the position of the agent with respect to the obstacle encountered on the way or of any particular action undertaken by the agent in order to bypass an obstacle. Based on these observations, we can finally define the purpose of a behaviour as follows:

Definition 4.2 (Purpose of Behaviour) *The purpose \mathcal{P} of a behaviour β is given by the goal \mathcal{G} which an agent performing β aims to achieve.*

Any behaviour, which can be regarded as the external manifestation of an interaction between an agent and its environment, has to be implemented in a physical entity, in order to be carried out by an agent. Such an entity, which takes the name of *behavioural module*, can be thought of as an opportune combination of hardware and software.

Definition 4.3 (Behavioural Module) *A behavioural module is a unit encapsulating a combination of hardware and software in order to accomplish a purpose in the real world.*

In this regard, it is worth noting here the divergence from Brooks and Steels according to whom behaviours may emerge from the interactions among the different modules of the system. Emergent behaviours in our case are not permitted, because we want each behaviour to be implemented as a behavioural module.

Behaviours are modular in the sense that they can be composed together so that to accomplish more complex tasks ([Malcolm 90]). In this respect an entire assembly may be hierarchically viewed as a single grand behaviour made of several simpler units (sub-behaviours). Since behavioural modules accomplish behaviours, the same observation drawn here can be transferred to behavioural modules, too. Thus, a module β_m implementing a complex behaviour β may be decomposed in terms of modules implementing simpler behavioural components of β ([Mataric 92a]).

Besides modularity a behavioural module should also satisfy other characteristics:

- it handles the uncertainty typical of the task,
- it integrates sensing and action at a low-level,
- it is computationally minimalist, and
- it knows as little as necessary.

In this regard we have to point out that in a behaviour-based assembly architecture it would also be preferable (but not necessary) if behavioural modules could run in parallel with others and sense their triggering condition instead of being activated by procedural delegation.

In general a system in which tasks may be expressed in terms of a collection of behavioural modules constitutes a behaviour-based system ([Steels 94b]). Such an observation transferred to an assembly world yields the following definition:

Definition 4.4 (Behaviour-Based Assembly System) *It is a system made entirely of behavioural modules with a minimum of central control (ideally none) and a minimum of general representation of the world (ideally none).*

We have to point out that the four definitions¹ given above (from def. 4.1 to 4.4), are very important to our research: they will be in fact extensively referred to in the rest of this thesis.

As mentioned in section 3.1 on page 66, we do not know how many behaviours, and hence behavioural modules, a system such as this would consider vitally important in order to be able to program useful assembly work. However, we do know that 95% of assembly tasks are clustered in just seven basic classes (cf. page 52), and that more complex tasks may be defined in terms of them. Thus, given the property of modularity, an architecture such as the one defined above requires first to develop basic modules accomplishing the aforementioned elementary assembly tasks, and then to program complex assemblies in terms of them, which are supposed to be easier and simpler than, for instance, VAL II commands. In this regard, a planner built following a

¹ The complete list of definitions is reported in [Malcolm 90].

behaviour-based assembly design may take advantage of modules previously developed in the same way as, for example, we may take advantage of off-the-shelf mathematical functions for solving particular mathematical problems.

Each behavioural module should be kept as ignorant as necessary about general knowledge of the environment, but nonetheless it should anyway achieve the purpose it is meant to in the best way it can. A behavioural module, whose execution by an agent produces the desired behaviour, is therefore forced to cope with the problems which may arise at *run time* by devising an appropriate solution. In this way a behavioural module may be interpreted as an interface which masks the underlying work-cell details from the planner. The advantage which clearly emerges from the architectural choice of keeping modules ignorant is twofold. On the one hand, because each module requires little information to run, and therefore few software parameters as inputs, it can be less often mistaken, hence it gets more robust. On the other hand, because modules can be combined together to accomplish different tasks, they extend the generality of their applicability.

From these two remarks emerges an important issue: how can a behavioural module communicate with its external world (*i.e.* other behavioural modules)? We know that it may need as input some external knowledge, eventually none, but what about its output? The one thing we know about is that the execution of a module may manifest different external behaviours according to how the results gathered by the execution are interpreted by an observer. For instance, let us assume to use the binary sensor introduced on page 80 which allows to detect event contacts, and let us suppose that the behavioural module implementing *peg-in-hole* can terminate in three possible states: 1) *peg mated with the hole*, 2) *hole searched but not found*, and 3) *obstacle found before the end of the search for the hole*. With these assumptions in mind, we may use *peg-in-hole* not only to achieve, as its name suggests, the insertion of a peg into a hole, but also to test the presence of a step wall on a surface by testing the occurrence of the third outcome (cf. *hopping spiral search* in section 5.2 on page 115).

Looking back at the above example, it is natural to wonder how we can make this information available to other behavioural modules. Since as mentioned above the execution of a task may in general yield many different final states, we can code the

particular situation in which the world is left at the end of the execution with a number, which is then given as output to the behavioural module: we call such an output code *exit state* ([Wilson 92]). This allows to carry in just one parameter all the information regarding the state which an agent finds itself in at the end of the execution of the module. However, such a parameter may in general not be the only output, as it may happen that also results of some computations or measurements are returned as outputs of the module.

The possibility of having an exit state to code the situation in which agent and environment are left at the end of the module is very useful at the stage of designing a behavioural program. In this regard, though, there are two other points which we have to discuss: the graphic representation of behavioural modules (cf. subsection 4.1.1) and their composition (cf. subsection 4.1.2).

4.1.1 Graphic Representation of Behavioural Modules

We know from definition 4.3 on page 86 that a behavioural module is an opportune combination of hardware and software. As observed above, they may require some parameters as inputs and may optionally return results of computations or measurements as outputs, but they always return an exit state coding the particular situation which agent and environment are left when the execution of the module terminates.

As we know, an agent running a behavioural module performs a behaviour, and hence achieves a purpose in the domain of the tasks in the best way it can. In order to do so, though, first it has to gather from its sensors as much information about the world as it can, and then it has to select the way in which it can achieve its goal by processing optional input parameters to the behavioural module. It is clear that, in order to run hardware and software components, some sort of local control is needed. Such a control would be responsible for local computations and for directing the execution of lower modules. Moreover, some kind of local knowledge may also be required in order to perform some particular operations of a task. Such a knowledge may be used to set optionally some of its hardware and/or software components. These remarks are all summarized in figure 4.1 on page 90.

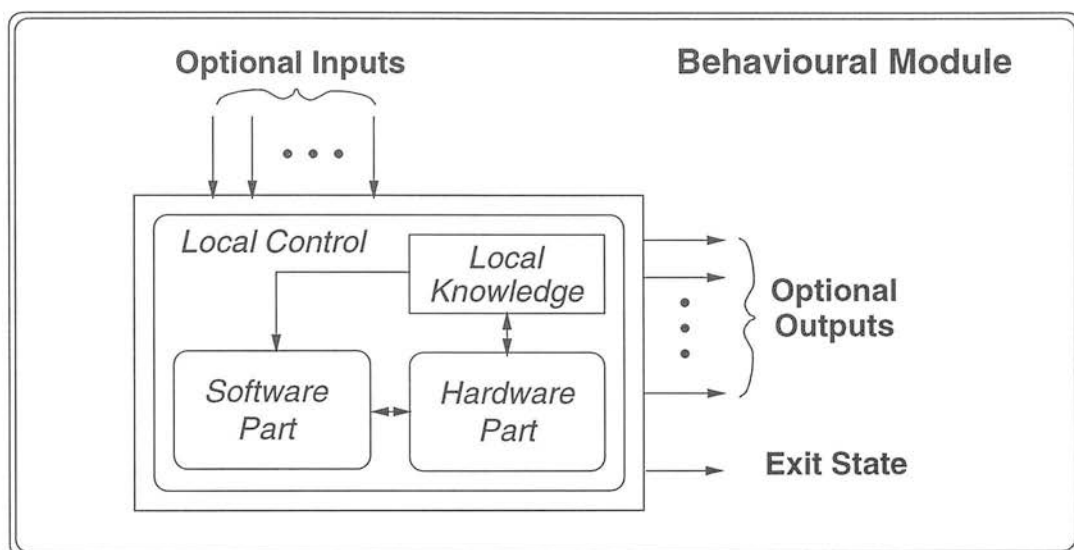


Figure 4.1: General Diagram of a Behavioural Module.

As regards the local control, we have to point out that the role it plays in a behavioural module is twofold: on the one hand it monitors and runs optional hardware units, and on the other it processes the data optionally obtained by the hardware components together with the local software computations. Such a control has to be performed somehow relying on commands belonging to a low level robot language such as VAL II. However, we do not have to think that both hardware and software components run physically within the same system. In general, different unit components may reside in different places. For instance, if we think of a situation where a robot has to perform a task by driving an end-effector and by monitoring the execution with some sensors, then the behavioural module driving the agent would have to deal with the end-effector driving, the sensor monitoring, and the strategy to achieve the task. In such a case driving unit of the end-effector could be located in a servo-mechanism, the sensor monitoring in the robot controller, and the strategy computation in a workstation. All of these units would interact with each other to achieve a final goal, but they would not be inside the same physical system (cf. figure 4.2 on page 91).

At this point let us return to talk about the other characteristic of behavioural modules mentioned at the beginning of this section: modularity.

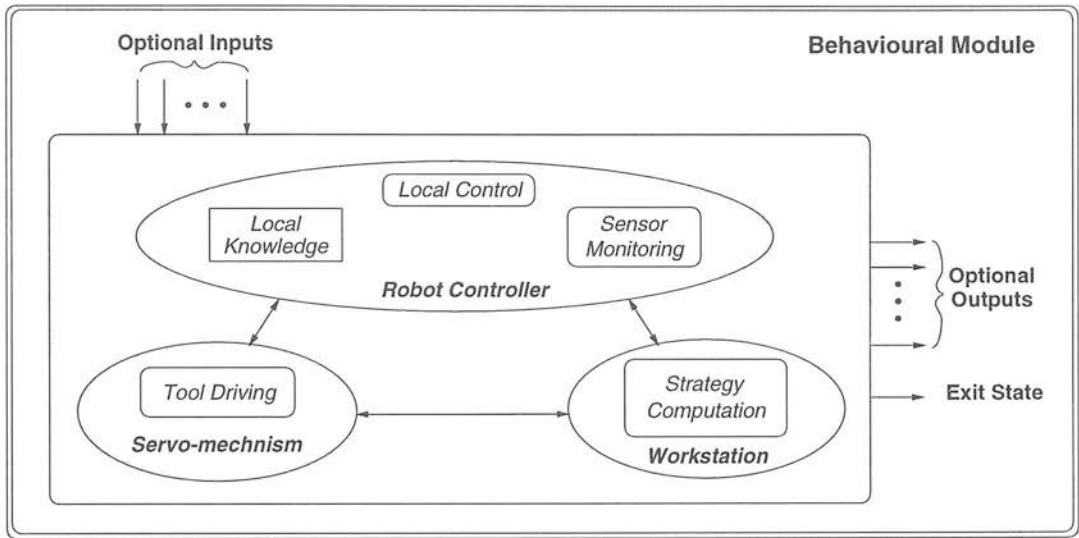


Figure 4.2: Example of a Behavioural Module Units Location.

4.1.2 Composition of Behavioural Modules

As discussed in the previous subsection, behavioural modules may be seen as black boxes with some optional parameters as inputs, with an exit state to code the situation at the end of the execution of the module as output and optionally with some results or measurements as other outputs. Here in this subsection we are not interested, as we did earlier, in analyzing what the content of these black boxes should be, but we want instead to discuss how they could be composed together so that their external interaction with each other drives the agent to achieve a certain goal.

Each behavioural module executed by an agent and considered by itself performs a behaviour, and hence achieves a goal in the domain of the tasks. If we find a way to join together somehow a few of these modules, we would accomplish more complex tasks. To this end we established in the previous subsection that a behavioural module exercises a sort of local supervision on the execution of a module. Such a control defines how a goal is achieved in response to some input parameters (stimuli), and therefore it defines the purpose of the entire module. However, we have not so far explained how this control is exercised on the different modules. In other words we do not know as yet how to compose modules together. To this end we characterized the output of a behavioural module to be a parameter (exit state) which codes the situation at the

end of the execution of the module itself (cf. page 89).

In order to make use of the information provided by an exit state, we have to define opportune behavioural module operators. To start with, let us observe that any assembly may be accomplished as a sequence of steps. Thus, we reckon that one way to compose modules should be the sequence which we can define as:

Definition 4.5 (Sequence) *A behavioural module β_m is the sequence of the modules β_{m_1} and β_{m_2} , and we indicate it with $\beta_m = \beta_{m_1} \otimes \beta_{m_2}$, when it allows an agent to perform strictly in order first the behaviour associated with β_{m_1} and then the one associated with β_{m_2} .*

Another interesting remark drawn in the previous subsection pointed out that different modules may in general run independently, possibly even in different physical systems. Such a remark leads to the conclusion that a certain goal may be achieved by running more than one module at the same time. For instance, picking up an object in an area clustered with obstacles would require an agent to perform picking up and obstacle avoidance at the same time. A situation such as this is not rare, thus we reckon that a second operator which we should define is fork.

Definition 4.6 (Fork) *We define a behavioural module β_m to be a fork of the behavioural modules β_{m_1} and β_{m_2} , and we indicate it with $\beta_m = \beta_{m_1} \odot \beta_{m_2}$, when an agent performs it by contemporaneously running β_{m_1} and β_{m_2} .*

Another situation which may occur, and therefore we need to deal with, is represented by the possibility of selecting and running specific behavioural modules and disregarding any other. An example would be a task such as putting a peg into a hole which would select a round peg-in-hole module or a square peg-in-hole one according to what peg is currently being held. This represents another form of composition which we call selection and which can be defined as follows:

Definition 4.7 (Selection) *We say that a behavioural module β_m is the selection of the modules β_{m_1} and β_{m_2} according to a condition predicate*

Cond, and we indicate it with $\beta_m = \beta_{m_1} \overset{Cond}{\oplus} \beta_{m_2}$, when an agent performs, depending on the logical outcome of the condition *Cond*, either β_{m_1} or β_{m_2} .

Another interesting common situation occurs when a task requires repetitions of the same pattern of actions. An example would be, for instance, palletizing² parts collected from an input chute. In such a case an agent has repeatedly to pick up a part from the chute and place it on a pallet. This way of composing modules is what we can label as repetition. In this regard we have to observe that there are actually three forms of repetition: repeating while a condition holds true (while-repetition), repeating until a condition is satisfied (repetition-until), and repeating a certain number of times (for-to-repetition). The first two of them are equivalent, however, it is better for a question of convenience to retain them as separate operators.

Definition 4.8 (While-Repetition) *We define the composition operation of while-repetition of a behavioural module β_{m_1} with respect to a predicate condition *Cond*, and we indicate it with $\beta_m = (\beta_{m_1})_{Cond}^*$, the module which allows an agent to repeat β_{m_1} while *Cond* holds true.*

Definition 4.9 (Repetition-Until) *We define the composition operation of repetition-until of a behavioural module β_{m_1} with respect to a predicate condition *Cond*, and we indicate it with $\beta_m = (\beta_{m_1})_{Cond}^*$, the module which allows an agent to repeat β_{m_1} until *Cond* is satisfied.*

Definition 4.10 (For-To-Repetition) *We define the composition operation of for-to-repetition of a behavioural module β_{m_1} for *N* times with respect to a counter *i*, and we indicate it with $\beta_m = (\beta_{m_1})_{i=1}^N$, the module which allows an agent to perform β_{m_1} from *i* = 1 to *i* = *N*.*

At this point we observe that these six operators (cf. from def. 4.5 to def. 4.10) are enough to define by composition any programs (cf. [Böhm & Jacopini 66] and [Dijkstra 69]). Thus, given the isomorphism between the space of the behavioural modules \mathcal{B}_M and the space of the programs \mathcal{P} , we can say that they are enough to

² A pallet is a common type of container used in a factory to hold and move parts.

represent any behavioural module β_m in \mathcal{B}_M . This result is exactly what we were looking for at the beginning of this subsection, because, by using these operators³ (sequence, fork, select, while-repeat, repeat-until, and for-to-repeat), we are now in the condition of composing simple modules to create composite ones.

Now, in order to give a graphical representation of each of them, let us summarize some points. Every behavioural module has an exit state as output for coding the situation which both agent and environment are left in at the end of its execution. A module may in general have also some other optional outputs, such as results of computations or measurements (cf. page 89). In this regard, for the sake of simplicity, we indicate all the outputs from one module component with just one line. Every behavioural module has a sort of local supervision (cf. *local control* on page 89) which allows it to control the execution of its components.

Based on the above remarks, we can talk more specifically about each operator. As regards sequence, we have to observe that the local control of a module has to evaluate the outputs of each component in cascade. According to the result of this evaluation, the control allows to follow the stream of the behavioural sequence, *i.e.* the next module, or to quit the sequence with an opportune exit code. As regards fork, we have to observe that the local control still plays an important role because it allows both to recognize the achievement of a goal and to inhibit the execution of the other module once the target goal has been reached. As regards select, the local control has simply first to process the output of the selected module component and then to encode it so that to represent all the possible execution outcomes. Finally, as regards repetition, we have to distinguish the two cases: while-repeat and repeat-until. The former subordinates the execution of a module component to the truth of a certain condition, whereas the latter to the falsity. Thus, the former requires a control to be exercised before the execution of the behavioural component, whereas the latter requires such a control to be held after the component execution. As regards for-to-repeat, the repetition is controlled by a counter which is incremented each time the module is performed up to the N -th time. Figure 4.3 on page 95 summarizes graphically the observations drawn above. Notice that, by omitting the inputs to each

³ As regards the repetition operators, only one is actually necessary, the other two are just convenient re-expressions.

module, we have simplified the diagrams, because we wanted to draw attention mainly on the composition of the modules.

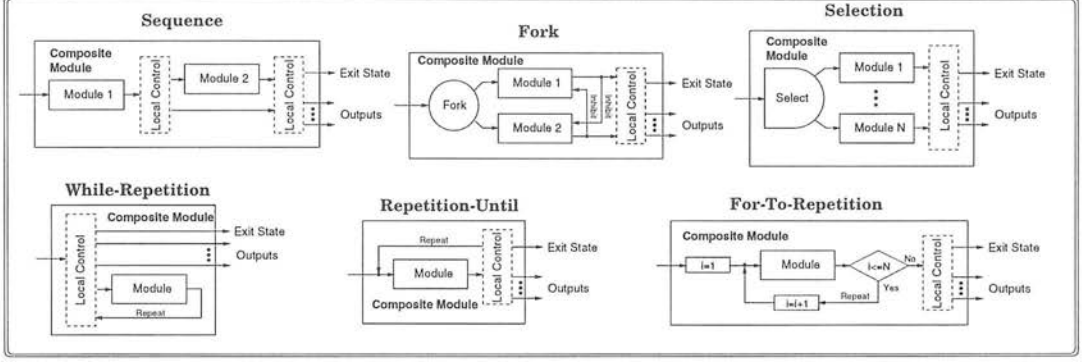


Figure 4.3: Behavioural Module Basic Operations.

At this point, let us turn our attention to the second concept we mentioned at the beginning of this chapter: the hierarchy of behavioural modules.

4.2 Hierarchy of Behavioural Modules

As pointed out earlier (cf. section 4.1 on page 86), any behavioural module can be expressed in terms of more primitive behavioural units. This means that, by using the operators introduced in subsection 4.1.2 on page 92, simple modules can be composed together so that to achieve complex goals. This remark yields the conclusion that some sort of hierarchy holds among behavioural assembly modules (cf. [Wilson 94]). Conclusion which in turn leads to the concept of behavioural module complexity.

In order to rank behavioural modules, we have to establish when a certain module is more complex than another. It is useful in this respect defining what we mean with most primitive behavioural module:

Definition 4.11 (Most Primitive Behavioural Module) *A module β is one of the most primitive when it is expressed directly in low-level terms of the robotic system, such as hardware signals, robot joints' specifications, or low-level robot language commands, but it is not expressed solely in hardware terms.*

Any complex behavioural module is defined by opportunely composing together these particular primitive units. Thus, we can by convention assume them to be the lowest level (level zero) of the hierarchy mentioned at the beginning. Modules expressed directly in terms of them will belong to the next level of the hierarchy (level one). Level two of the hierarchy will, in turn, be made of those expressed in terms of modules at the level zero and/or one, and so on.

Based on the above observations, we can now give the definition of hierarchical level of a behavioural module.

Definition 4.12 (Behavioural Module Hierarchical Level) *Given a behavioural module β , we say that it belongs to hierarchical level zero if it is one of the most primitive, and that it belongs to hierarchical level \mathcal{N} if it is expressed in terms of modules up to hierarchical level $\mathcal{N} - 1$.*

In this regard, since the level occupied by a module in the hierarchy may be viewed as a form of complexity⁴, we can actually use it as a measure of it. Thus, we can formalize the complexity of a behavioural module as follows:

Definition 4.13 (Behavioural Module Complexity) *We define complexity \mathcal{C} of a behavioural module β its hierarchical level.*

By virtue of such a definition we can say in the end that a behavioural module β_1 is more complex than a behavioural module β_2 when $\mathcal{C}(\beta_1) > \mathcal{C}(\beta_2)$, that is when β_1 occupies a higher hierarchical level than β_2 .

At this point we may wonder what level of complexity the modules implementing the basic set of elementary behaviours we are looking for should have. In order to give an answer to this issue, we dedicate the next subsection to discuss this subject.

4.2.1 Hierarchical Level of the Basic Behavioural Modules

As claimed earlier, the level zero of the behavioural modules' hierarchy is made of what we called most primitive behavioural modules (cf. def. 4.12 on page 96), which are

⁴ The term *complexity* of a behavioural module is intended here as the complexity of implementing both hardware and software of the module itself.

those modules defined directly in terms of the robot hardware and low-level software, such as manipulator-level commands (cf. definition 4.11). In this regard, we ought to observe that VAL II commands fit such a definition. For instance, commands like `move` and `signal` in a broad sense have a purpose and, when executed, may produce an external manifestation of the interaction between an agent and its environment, therefore we can claim that they are very simple form of behavioural modules, indeed the most primitive in the sense given in definition 4.11. Using these commands, we are actually able to program any assembly task executable by the robot manipulator, however, as explained in subsection 2.1.3 on page 17, robot programs expressed in these terms are not easy to be written and debugged. They basically fit the definition mentioned above, but they are not what we are looking for (cf. section 3.1 on page 66). In this regard, an interesting example is provided in mobile robotics by Matarić, who suggests that a small group of five elementary behavioural modules (safe-wandering, following, dispersion, aggregation, and homing) is capable, by opportune combinations of its items, of describing any complex behavioural module related to motions on a plane (cf. [Matarić 94]).

By virtue of what said above, the set of basic behavioural modules with which we can express most of the assemblies, and which is ergonomic for the operator to use, cannot be the set of the most primitive ones. We need therefore to modify definition 4.11 so that the level zero of the behavioural modules' hierarchy is set to those which, despite their relative simplicity, still achieve a goal in the domain of the task assemblies when they are executed by an agent. We call them basic behavioural modules, and the behaviours, which result from their execution, basic behaviours. We can formalize these observations in the following definition:

Definition 4.14 (Basic Behaviour) *A behaviour β accomplishing a task \mathcal{T} and achieving a goal \mathcal{G} in the domain of the assemblies \mathcal{A} is said to be a basic behaviour when it cannot be further decomposed into simpler behavioural units which still achieve a goal in \mathcal{A} .*

Definition 4.15 (Basic Behavioural Module) *Any module β_m implementing a basic behaviour β takes the name of basic behavioural module.*

A set of modules fulfilling definition 4.15 is the basic set of elementary behavioural modules which we are looking for.

4.3 Generality of Behaviours

Following a behaviourist approach, we defined earlier a behaviour β as a mapping of stimuli and actions to a goal (cf. def. 4.1 in section 4.1 on page 85). According to this view, an agent performing β aims to achieve a certain goal \mathcal{G} , thus this last defines the purpose of β . In this regard, the work carried out by Balch raised an important issue: generality (cf. subsection 2.3.2 on page 55).

He argued that if we assume behaviours to have a specific purpose, then they become basically *ad hoc* solutions to a particular assembly problem and they cannot be used again in other similar situations. His claim was that generality had to be searched for at a lower-level where the programming units were a form of generalized procedures⁵ designed without any particular purpose. According to this view, a behaviour built using his modules acquired a purpose only from the particular way in which they were composed together. In this regard, although we agree in principle that a purpose of a behaviour is given by the particular way in which its components are linked together and interact with each other ([Mataric 92b]), we do not subscribe the idea that such a purpose is necessarily the only one ([Young 94]). In other words, we think that a behaviour may have within a certain scope more than just one particular purpose. Hammering a nail, for instance, may be viewed as a behaviour which aims at hanging something on a wall, like a painting, or at joining two wooden parts together. Thus, the characteristic of disjunctive purpose introduced by Balch's modules does not lower them to the level of mere generalized routines, because some of them may actually be regarded in a broad sense as primitive forms of behavioural units.

A behavioural module is defined as having an exit state which codes the particular situation which agent and environment are left in at the end of its execution (cf page 89). Since a module always terminates in one of a few possible exits, it does not succeed or fail: this would be simply derived from the use we make of it. For instance, if we

⁵ A generalized procedure has to be intended here as a routine which integrates both hardware and software.

assume that *peg-in-hole* has three exits (*peg inserted in hole*, *hole searched but not found*, and *obstacle detected during search for the hole*), then each would be a success or a failure according to which purpose such a behaviour is used for. We argue that this is exactly what makes behaviours general. Thus, the claim that they can not be used in many situations because of their fixed purpose does not hold. As argued above, we can actually define within a certain scope multiple purpose behaviours, which can be composed together so that to bring to light new composite ones ([Steels 93]) whose purposes are given by their components.

4.4 Summary

At this point we can summarize what we have discussed and achieved in this chapter. The main points which we wanted to cover here were the definition of the concepts of *behaviours* and *behavioural modules*, the definition of *hierarchy of behavioural modules*, and the discussion about what generality behaviours have.

As regards the first of them (cf. section 4.1 on page 84), we followed the behaviourists' approach and we defined it as a 3-tuple of stimuli, actions, and goals (cf. def. 4.1). Associated with such a concept, we discussed two other important concepts: the purpose of a behaviour and a behavioural module. The former was defined as the goal which an agent performing it aims to achieve (cf. def. 4.1), and the latter as one of its possible implementations (cf. def. 4.3). Behaviours know as little as necessary about the general knowledge of the world, and this gives them the advantage to be more robust and reliable, because the less they know about it, the less they can be mistaken by it. Regarding behavioural modules, we pointed out that they have two basic characteristics: they are modular and they always terminate in just a few possible exit states which is then returned as an output. By coding them with a number, we can actually use just one parameter to carry information about the result of the execution of the module. Taking advantage of the characteristics of modularity, we defined 6 composition operations (sequence, fork, selection, while-repetition, repetition-until, and for-to-repeat) which, given the isomorphism between the space of the behavioural modules and the space of the programs, enable us to generate any other behavioural module starting from basic ones (cf. from def. 4.5 to def. 4.10).

As regards the concept of hierarchy of behavioural modules (cf. section 4.2 on page 95), we maintained that since modules may be expressed in terms of more primitive ones, then some sort of hierarchy holds among them. To the end of defining such a hierarchy, we introduced the concept of most primitive behavioural module (cf. def. 4.11). We showed that any module expressed in terms of hardware and low-level software commands, but not solely in terms of hardware, has to be considered most primitive. We pointed out that a module such as this belongs to the level zero of the hierarchy, and that any other belongs to the level N if it is defined solely in terms of modules up to $N - 1$ (cf. def. 4.12). By virtue of this, we claimed that the hierarchical level of a behavioural module may be considered as a measure of its complexity (cf. def. 4.13). We concluded the section stressing the fact that we are looking for a set of basic behavioural modules in the domain of the task assemblies, that is a set of modules which cannot be decomposed in sub-tasks with a purpose in such a domain.

The last and final discussion of this chapter (cf. section 4.3) was dedicated to a very crucial issue: the generality of a behaviour. We pointed out that an earlier work of Balch suggested that in order to have full generality we cannot associate a specific purpose to a behaviour, because in this way its applicability would be constrained. We argued that behaviours may actually be defined with a disjunctive purpose characteristic. Thus, looking for general-purpose behaviours may be considered a viable approach.

Chapter 5

Project Experiments

As defined in section 3.1 on page 68, the problem which we want to tackle consists of investigating the existence of a basic set of ergonomic¹ elementary behavioural modules with which programming 80% of the assembly tasks. To this end we pointed out two important issues: existence of multiple purpose assembly behavioural modules, and their generalization. In order to address these issues, we laid out the framework of our research as follows (cf. section 3.2 on page 68):

- 1) selecting significant assemblies,
- 2) decomposing them in terms of the necessary behaviours,
- 3) refining each behaviour up to its minimal behavioural terms within the behaviour-based assembly paradigm,
- 4) synthesizing and generalizing the minimal terms found.

In this respect, we have already accomplished the first step listed above (cf. subsection 3.2.3 on page 73) by selecting the benchmark assembly family (cf. figure 3.2 on page 74), the STRASS assembly (cf. figure 3.4 on page 76), and the torch assembly family (cf. figure 3.5 on page 77).

What we are going to examine in this chapter is the second and third point of the aforementioned framework, leaving the discussion of the last one to chapter 6 on page 189. However, before doing so, we have to precede the description of our experiments with

¹ In the sense of being convenient and useful elements for programming.

the discussion of an important point: guarded motion. As will be realized throughout the chapter, this is an important building block for describing both the selected assemblies and many other ones.

For a question of convenience we divide our discussion in four parts: the guarded motion (cf. section 5.1 on this page), the benchmark assembly family (cf. section 5.2 on page 115), the STRASS assembly (cf. section 5.3 on page 156) and the torch assembly family (cf. section 5.4 on page 165).

At this point, before starting our discussions, we want to draw the attention to two important hardware assumptions took back in subsection 3.2.1.1 on page 3.2.1.1: the use of a point-to-point robot with a SCARA configuration, and the use of the VAL II programming language. We also assumed our robot to be equipped just with a simple binary touch sensor (cf. subsection 3.2.4 on page 78 for further details).

5.1 Guarded Motion

One of the most important functions of textual robot languages, and the main feature which distinguishes them from computer programming ones (cf. subsection 2.1.3 on page 17), is manipulator motion control. A motion command can be defined as a mapping from valid starting and ending robot configurations to nominal workspace trajectories ([Latombe 91]). As recalled above, we assumed to work with a point-to-point robot. Thus, it is plain that any language designed to program such an agent has to incorporate at least a few dedicated commands to make the manipulator move from a starting spatial point A to a terminating spatial point B. The syntax in which motions would be specified greatly depends on the particular language involved.

Robot motions are usually distinguished between two categories: gross and fine motions ([Whitney 89]). The former aim simply at approaching the vicinity of a target point within the robot envelope while avoiding obstacles ([Strenn *et al.* 94]), whereas the latter aim at carefully attaining such a target. Often during a gross motion there is little need to follow a precise path, and the emphasis is on speed. On the contrary, speed is less dominant in fine motions, where the emphasis is put more on adjustment and reaction ([Lozano-Pérez *et al.* 83]). Textual robot languages provide in general

specific commands for both gross and fine motions.

There are two most popular ways of implementing robot motions: by straight line or by joint interpolated ones ([Groover *et al.* 86] and [Craig 89]). Whatever the case is, a motion is most of the time accomplished without any external sensor (open loop motion). However, if we want our robot to cope with world uncertainties and to correct and adapt its behaviour according to its environment, a kind of interruptible sensor-based *move* command needs to be developed in some way (*guarded move*²). To this end we took the decision to equip our robot with a differential touch sensor (cf. subsection 3.2.4 on page 80) capable not only of detecting any contacts with obstacles along a motion path but also of immediately halting the manipulator whenever such an event occurs.

Developing a general guarded motion command capable of performing both gross and fine motions is what we aim to achieve in this section. The execution of such a command, as we pointed out in the example reported in subsection 4.2.1 on page 96, achieves a purpose in the domain of the tasks, thus it may be regarded as a very simple form of behaviour, indeed a guarded motion would fit what in definition 4.11 on page 95 we labelled as most primitive behavioural module. Because of its broad generality it is used throughout this chapter as the basic unit in terms of which to express any actions for putting together parts of the assemblies discussed in sections 5.2, 5.3, and 5.4 on page 115, 156, and 165 respectively. For convenience we divide the discussion about guarded motions in two subsections: analysis of the task (subsection 5.1.1 here below), and description of a possible implementation (subsection 5.1.2 on page 107). The section is concluded with a summary of the results obtained.

5.1.1 Guarded Motion Analysis

As mentioned at the beginning, a guarded motion is basically a sensor-based move command, whose purpose is simply to detect or avoid any possible contacts of the manipulator with other objects during the motion from the current location to a specified point within the work-cell. Thus, as recalled above, it can be regarded as one of the most primitive behavioural module.

² Confer with footnote on page 23.

A great deal of research in guarded motion has focussed attention on obstacle avoidance. In this respect, the concept of reflexive behaviour presented in [Wikman *et al.* 93] and [Wikman & Newman 92] which allows the movement of a manipulator in a cluttered environment along a collision-free path is quite interesting to us because it implements the feature of reflexive reaction of an agent in response to external stimuli (obstacles). Another relevant research project in this topic was the development of a segmentation algorithm for providing a robot with model-based, real-time, whole-arm collision avoidance ([Strenn *et al.* 94]). Other research topic areas in robot motion concentrated on control. In this regard, a reciprocity-based task decomposition was proposed as a general and unified theory for hybrid position/force control of manipulators ([Sinha & Goldenberg 93]). As regards control in particular, it was also established that an asymptotically stable scheme for motion control of rigid robots can be developed with induction motor drives ([deWit *et al.* 93]). A project for space robotic applications proposed the definition of a method to estimate and predict general three-dimensional motions of an unknown rigid body under no external forces and moments using vision information ([Masutani *et al.* 94]). However, perhaps the most intriguing research area regarding guarded motions is the development of external sensing capabilities for directly controlling the performance of a motion. In this respect, a collision avoidance robot system using whole arm proximity sensors (WHAP) was developed on a PUMA 560 arm ([Feddema & Novak 94]). The sensing was achieved by employing capacitance-based sensors which, by generating electric fields completely encompassing the arm, enable the robot to detect obstacles as they approach from any directions. The information gathered by the WHAP sensors together with their geometry and robot configuration is used to scale the commanded joint velocities of the robot, and hence to control the motion of the manipulator. Much research is being carried out in the development of reliable sensors for robotics applications ([Lindstedt & Olsson 93], [Joseph & Rowland 95], [Williams 95], [Kim 96]). Sensors are most of the time handled by specific low-level routines which are often not portable from one system to another. In this respect, the concept of a virtual sensor is interesting, which is thought of as an abstraction offering a uniform way of dealing with sensors, and a structured approach simplifying sensory data acquisition ([Armstrong 95]).

As recalled at the beginning, we decided to develop a guarded motion employing a very

simple form of binary touch sensing. Thus, we have to think of it as a module which corrects and adapts the motion of the agent to the constraints of the environment. In this context such a module would be unable to avoid obstacles without prior touching them, and also it would not be able to perform any kind of whole arm collision avoidance.

In order to define a possible implementation of the task, we need to think of it as being carried out by concurrently performing two sub-tasks: the movement itself, and the sensor monitoring (cf. figure 5.1 here below). As soon as the module is entered, an

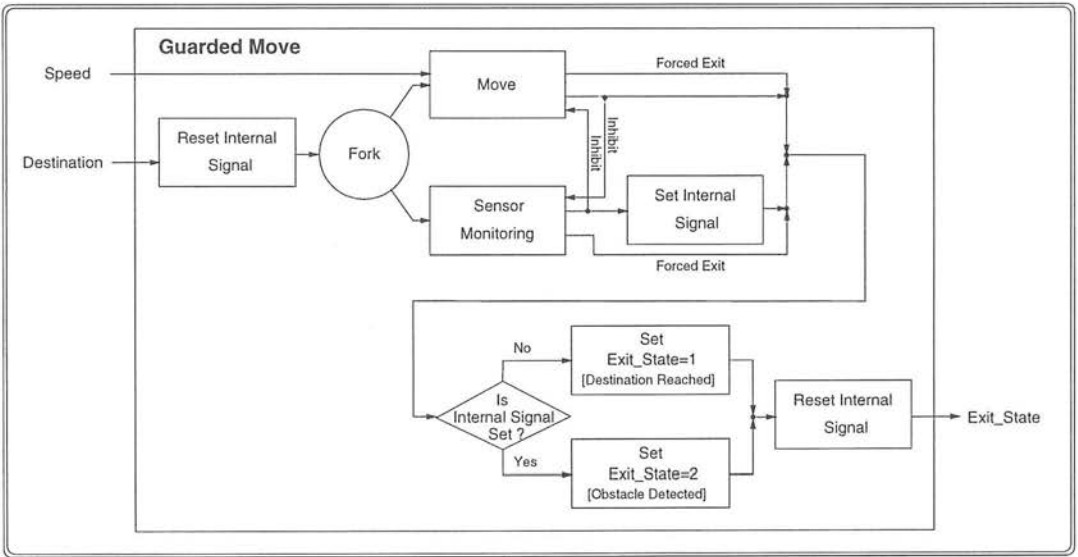


Figure 5.1: Guarded Move Diagram.

internal signal is reset in order to prepare the sensor monitoring sub-module to be run. Once this is done, a fork operation splits the execution control into two modules: first, a motion sub-module responsible for driving the agent, and second, a sensor monitoring sub-module responsible both for checking the occurrence of any signal issued by the sensory device and for reflexively stopping the motion and the sensor polling when such an event occurs. The two sub-modules are executed in different systems: the motion itself is performed by a robot controller, and the sensor monitoring by an external device, in our case a PC, which, by continuously polling a binary touch sensor (piezo-film), is able to interrupt our manipulator agent (Adept) when a certain threshold is exceeded (cf. subsection 4.1.1 on page 90). Incidentally, we observe that this second

sub-module is capable of detecting both direct and indirect collisions³ of obstacles with the sensor surface.

Each sub-module can force a premature termination of the other by issuing an inhibition signal as soon as it completes its execution. Thus, both of them have two possible termination exits, which for the motion sub-module are forced exit and agent reached target location, whereas for the sensor monitoring are again forced exit and sensor triggered signal. In case the sensor monitoring sub-module terminates first, after having stopped the manipulator motion and the sensor polling, our guarded motion module has to set the same internal signal reset at the beginning to warn that an event contact has occurred. In case instead the motion sub-module terminates first, the sensor monitoring has to be prematurely stopped and the sub-module has to quit through the forced exit which bypasses the internal signal setting.

At the end of our guarded motion execution, the internal signal is checked, and, according to its setting, an exit state parameter is set to 1 if the destination is reached without collisions, or to 2 if a collision has been detected.

Now, let us turn our attention to another important point regarding this primitive task: the way in which the motion itself is carried out.

We mentioned earlier that we are employing a point-to-point robot. This means that the kind of motion performed are always from a starting spatial point **A** to a final spatial point **B**. This kind of motion can be performed by following either a joint interpolated path, or a straight line ([Groover *et al.* 86] and [Craig 89]). Choosing one solution instead of the other affects greatly the implementation of the behaviours based on it. As a general rule, we may say that since joint interpolated motions can always be performed, they become very useful when very large spatial motions within the work-cell are required ([Fu *et al.* 87]). For instance, if an assembly requires at a certain point the robot to turn 180° for collecting the parts it needs, a joint interpolated motion would always allow the robot to accomplish such an assembly subtask. Motions of this kind, though, end up requiring more computations when we

³ An indirect collision happens when the object held by the gripper hits an obstacle. When such a case occurs, the force vibrations spread from the point of contact throughout the object held by the gripper until they reach the surface of the the sensor whereupon the collision is detected and notified.

need for some reasons to make the robot end-effector approach a target destination along a straight path because a slightly curved one would cause problems like in a peg-in-hole operation. On the other hand, from the user point of view, straight line motions are instead much simpler to deal with: their only drawback is that some motions are forbidden ([Canny 87]). The robot is for example unable to move its end-effector to a certain location along a path which forces it to pass through itself (cf. figure 5.2 here below).

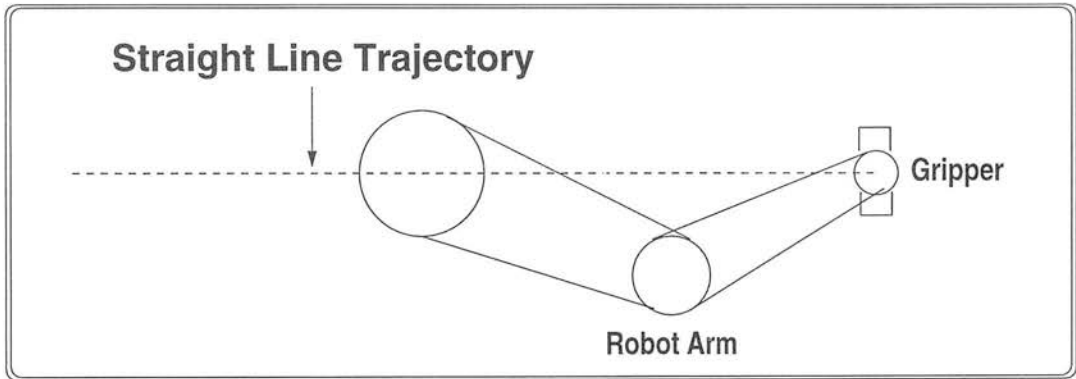


Figure 5.2: Example of Straight Line Motion impossible to be carried out.

Taking into account that most of the assembly tasks may be entirely carried out by a collection of simple straight line motions and assuming to work with valid trajectories⁴ only, we decide to sacrifice the power of the joint interpolated motion, which in any case may always be approximated by opportune sequences of straight lines, in favour of the simplicity from the user point of view of the straight motion⁵.

At this point let us turn the discussion to the issues raised by the implementation of guarded motion.

5.1.2 Guarded Motion Implementation

The particular implementation of guarded motion presented here, which we address from now on with the name of `guardedmove`, follows the discussion we had in the

⁴ A valid trajectory is any path which can be performed by the manipulator without causing joint errors in the inverse kinematics.

⁵ This kind of motion is from the robot point of view far more complex than the joint interpolated one.

previous subsection. To start with, let us point out that, since the manipulator agent employed in our research is programmed in VAL II, we need to express the control part of the module in terms of such a language (cf. subsection 4.1.1 on page 89).

There are two points which need to be defined precisely: first, what external knowledge such a module requires as input, and second, how to output the information about the state in which the robot is left at the end of the execution of the module.

As regards the former, we have to remark that the main characteristic we require for a suitable guarded move is to be able to correct and adapt the manipulator motions to the constraints of the environment which it is in. As said earlier, the simplest sensor which allows us to achieve such a characteristic is a binary touch sensor (cf. page 80). Thus, although a guarded move may in general be implemented with collision avoidance features, we are not going to use any other proximity sensor in order to achieve them. As a consequence, we do not provide any knowledge of scene understanding or navigational maps within the work-cell. What we require as input concerns the motion itself. In this regard, we already know from the previous subsection that our guarded move has to make our manipulator agent physically move its end-effector from its current location to a destination point within the work-cell. Assuming the robot to be capable of reading its current gripper position, we need to feed as input just two parameters: the target point and the speed with which to move there. This second input parameter is important because we want to use `guardedmove` both for gross and fine motions. In this regard, although we may determine an optimal speed limit for the latter, the former should not be constrained by any preset limit. Thus, we decided to retain it as an input parameter to be instantiated according to the kind of motion.

As regards the output of our guarded motion module, we need to point out that, since we are employing a simple differential touch sensor (cf. page 80), we assume an obstacle to be sensed during the motion path each time the touch sensor detecting an event triggers a signal to the robot controller, whose duty in such a case is to inhibit immediately any further movement of the manipulator arm (cf. figure 5.1 on page 105). If the motion is completely carried out without any event detection, we assume the robot end-effector to have reached its target destination. By virtue of these observations, we reckon that the execution of `guardedmove` may have just two possible

outcomes:

- 1) Target location reached,
- 2) Obstacle detected along the motion path before reaching the target destination.

Notice that we did not consider *trajectory impossible* as an outcome of the module, because we assumed to work with valid trajectories only. Thus, coding the two situations listed above with **1** and **2**, respectively, we may use them to represent all the exit states of the module, and hence its outputs.

At this point, given the importance of the guarded motion primitive behaviour, it is worth showing the details of the VAL II implementation which we are going to use throughout the rest of this thesis.

5.1.2.1 VAL II Implementation of Guarded Move

Before carrying on with the description of the VAL II implementation we ought to say that, as mentioned on page 80, the piezo-film composing the sensor and wrapped around both fingers may be subjected to positive and negative bendings whose different signals could be theoretically used to distinguish between making and breaking contacts (cf. figure 5.3 here below). However, there are considerable practical difficulties in

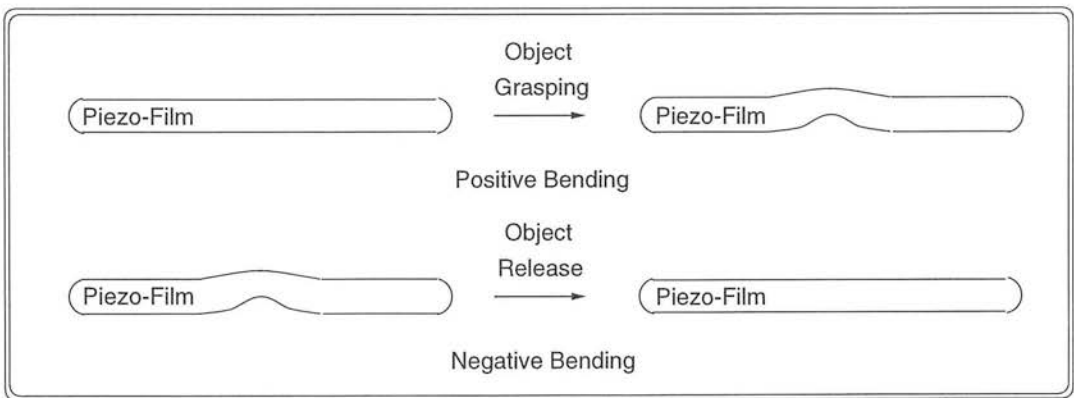


Figure 5.3: Piezo-film bendings and correspondent operations.

reliably realizing this distinction, since most curvatures of one sign are accompanied

at their edges by curvatures of the opposite sign, and the output is the algebraic sum of these ([Kim 96]). However, we do not need to make such a distinction. Thus, we decided to use only one parallel digital input line for every finger in order to detect an impact. This means that we require to employ two parallel input lines of the robot controller for the right gripper and other two lines for the left one (cf. appendix A, B, and C on page 228, 230, and 233, respectively).

When one of these lines triggers, an inhibition signal preventing further movements is issued to the robot controller and a specified subroutine is promptly executed. In this regard we have to point out that, although the movement inhibition is promptly issued as soon as an event is detected, the manipulator physically stops dead only after a certain delay caused partly by the PC software loop time for monitoring the four sensor lines, partly by our agent (Adept) interrupt time delay, and finally partly by the deceleration time.

In order to measure how much these delays affect our guarded motion, we put a weight of 1773 *g* at a well known location and we drove the robot manipulator 198.41 *mm* away along a straight line. Then, keeping the acceleration and deceleration parameters set to 100% and varying the percentage of the robot full speed (9 *m/sec*) from 1% to 10%, we repeatedly drove the manipulator against the weight, subject of course to a relocation of this last in the same position after each collision, and we measured the distance covered by the robot after having been interrupted by the sensor (cf. figure 5.4 on page 111 for the experimental readings).

As expected, the result gathered by this experiment was that, when the sensor triggers because of a contact, the robot stops moving only after having travelled a certain distance which increases with the speed. As far as our guarded motion is concerned, such a result means that if the robot holds a part and a collision is detected by the sensor, then, because of the software and hardware delays, the part may eventually be forced to slip through the manipulator fingers by the collision. Since the Adept robot which we are using is very stiff, the main compliance we may rely on is given by the finger rubber pads onto which the sensor is mounted. Such a compliance is limited to just 2-3 *mm*, thus we conclude that the behavioural modules based on this kind of guarded motion which exploits sensor readings, such as pick-up or stack (cf.

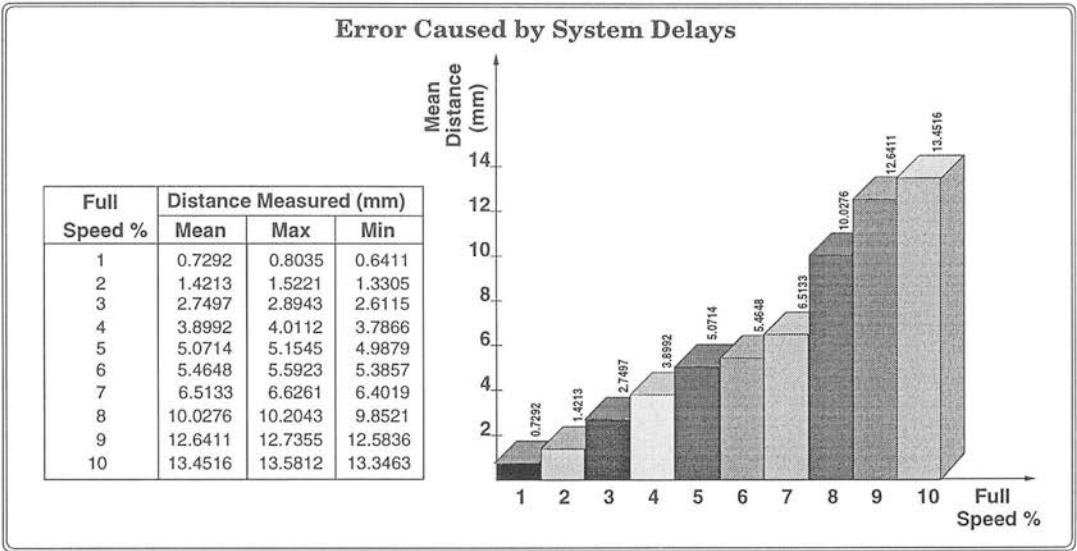


Figure 5.4: Mean Distance covered because of Hardware and Software Delays.

subsubsection 5.2.2.1 and 5.2.2.2 on page 139 and 143, respectively), should set the speed at no more than 2-3% of the full rate because at that level the travel can still be absorbed by the compliance of the finger rubber pads. It is plain that employing less stiff robots may allow the use of greater velocities. The speed limitation in behaviours involving contact exploitation (fine motions) is very important.

Returning now to the inhibition signal generated by the piezo-film bendings, we have to observe that, assuming that we handle the two kinds of bendings in the same way, the operations of grasping and releasing objects will always trigger at least one of the four lines. The reacting subroutine mentioned above has the purpose to set an internal signal to notify that a contact has taken place⁶. Such a flag, which needs always to be reset every time a new guarded motion is started, allows to discriminate between the two possible outcomes listed above.

At this point, before describing the VAL II implementation of our guarded motion, let us observe that the process which performs the sensor polling is run continuously in an external system (PC). Thus, the fork operation mentioned in the analysis of guarded move (cf. subsection 5.1.1 on page 105) consists basically in activating the monitoring of

⁶ This is necessary because of the limited parameter handling of VAL II.

the sensor lines. VAL II provides in this respect a useful reacting command (REACTI⁷) whose syntax is

REACTI < *Signal Line* > , < *Triggering Procedure* > , < *Priority* > .

Such a command warns the manipulator controller to perform a poll of the specified < *Signal Line* > before executing any other following VAL II instruction until a command of the form

IGNORE < *Signal Line* >

is encountered. Every poll checks if the logical value of < *Signal Line* >⁸ has changed from 0 to 1, and in this case it first interrupts both any motions of the robot arm and whatever the controller is currently executing, and then, if the < *Priority* > of the < *Triggering Procedure* > is higher than that of the main program, it triggers the execution of the < *Triggering Procedure* > .

At this point, following the diagram reported in figure 5.1 on page 105, we can describe the code of our guardedmove. To start with, the module receives as input the speed rate with which it has to drive the robot arm, and the final location of the work-cell where it has to move it, and it outputs the exit state parameter coding the outcome of the motion. The first operation performed is to reset an internal signal which is used as a flag⁹ in case an event is detected. Then, after having issued four reacting commands (cf. above), one for each sensor line involved, a subroutine performing the actual motion is called (uncondmove). Such a subroutine first sets the speed of motion (SPEED), and then activates the motion itself by issuing a MOVES command which forces the controller to drive the arm along a straight line¹⁰. Before quitting the subroutine, a BREAK command is issued so that the controller is forced to wait for the motion to be

⁷ The command REACTI, which stands for *react immediately*, is provided by VAL II together with the command REACT. This latter differs from the former because it does not stop the robot arm automatically when the line monitored is triggered.

⁸ If < *Signal Line* > is specified with a minus sign in front, then an interrupt signal would be triggered only when the logical value of < *Signal Line* > changes from 1 to 0.

⁹ This is caused by the lack in VAL II of any parameter passing between triggered procedure and main program.

¹⁰ The equivalent VAL II command driving the arm along joint-interpolated motions is MOVE.

completed or interrupted. Such a command is important because the robot controller always delegates the execution of the motion to its arm servomechanisms, thus, if such a command is not issued, the controller would proceed to execute the rest of the module without waiting for the move to be completed. The instructions following the subroutine are four IGNORE commands which disable the monitoring of the sensor lines. The penultimate instruction sets the exit state parameter to 1 or 2 according to the logical value of the internal signal (flag), which is set to 1 by the triggering procedure in case a contact event occurred (a piezo-film bending). At the end, before terminating the module, the internal flag is reset.

We report here below the VAL II code described above.

```
.PROGRAM guardedmove(WithSpeed,FinalLoc,Exit_State)
  SIGNAL -2001                      ; Reset Internal Flag
  REACTI 1001,setsig,127             ; Right Gripper: Finger 1
  REACTI 1002,setsig,127             ;           Finger 2
  REACTI 1003,setsig,127             ; Left Gripper : Finger 1
  REACTI 1004,setsig,127             ;           Finger 2
  CALL uncondmove(WithSpeed,FinalLoc); Motion Subroutine
  IGNORE 1001                       ; Right Gripper: Finger 1
  IGNORE 1002                       ;           Finger 2
  IGNORE 1003                       ; Left Gripper : Finger 1
  IGNORE 1004                       ;           Finger 2
  Exit_State = BITS(2001,1)+1        ; Set Exit According to Flag
  SIGNAL -2001                      ; Reset Internal Flag
  RETURN
.END

.PROGRAM uncondmove(WithSpeed,ToWhere); Motion Subroutine
  SPEED WithSpeed                   ; Declaration of Local Variable
  MOVES ToWhere                     ; Move Arm Along Straight Line
  BREAK                             ; Wait For The Move To Be Ended
  RETURN
.END

.PROGRAM setsig()                   ; Triggering Procedure
  SIGNAL 2001                       ; Set Internal Flag
  RETURN
.END
```

As final remarks, we point out that the internal signal used as a flag is associated with the Adept internal line 2001, and the four binary input lines polled during the execution of the module are associated with the four binary input lines 1001, ..., 1004. Each line is linked to the corresponding touch sensor wrapped around one of the fingers. Every time one of the touch sensors triggers one of these lines during the poll, the robot

controller immediately interrupts our `guardedmove` module and it starts executing the triggering procedure `setsig` with the highest priority (127).

As mentioned earlier, the logical value of the internal flag 2001 is set to 1 or 0 according to if a contact has been detected or not. Therefore, in order to set `Exit_State` to 1 or 2, it is sufficient arithmetically adding 1 to the value of the flag. The result of this sum codes the two possible outcomes of the module execution:

1. No contact event detected and motion carried out successfully.
2. Contact event detected before the end of the motion.

As last remark, we observe that the resetting of the internal flag 2001 at the end of the module is performed purely for a question of neatness.

5.1.3 Guarded Motion Summary

At this point let us briefly summarize what we have discussed so far. The guarded motion implementation presented above aims simply at allowing an agent to correct and adapt its motion to the environment constraints (cf. page 104). Thus, it is not intended to perform any whole arm collision avoidance. We observed that its purpose is to move from one work-cell spatial point to another. In this respect, our `guardedmove` has to be regarded as one of the most primitive behavioural modules in the sense given by definition 4.11 on page 95. We also observed that the motion itself can be carried out by either a joint interpolated path or a straight line. Because it is easier for the end user to reason in terms of straight lines, we decided to implement our `guardedmove` with this kind of motion, even though this requires more computations for the robot controller.

By experimentally testing our `guardedmove` with different speed rates, we observed that, since the robot is very stiff, the only compliance we can rely on is given by the finger rubber pads and it is constrained within the range of 2-3 *mm*. This means that, in order to avoid part slipping, all the behavioural modules exploiting sensor readings for fine motions have to limit their speed rate to at most 3% of the full speed (cf. figure 5.4 on page 111). However, modules using this implementation of `guardedmove`

for gross motions should not be constrained by any speed limitation. Thus, we decided to retain speed as an input parameter.

Regarding the motion itself, we also pointed out that, assuming the robot to be capable of reading its current position, all of what the module requires to know is the final destination. Therefore, we decided to use it as the second and last input parameter.

As regards the module output, we observed that the two possible execution outcomes are either target location reached (1), or obstacle detected along the motion path before reaching destination (2). Thus, we decided to use the numbers 1 and 2 to code the two possible exit states.

5.2 Benchmark Assembly Family

The assembly which we are going to discuss in this section is a family of two similar benchmarks (cf. figure 3.2 on page 74): one is made of aluminium (small benchmark) and the other of wood (big benchmark). Each member of the family is composed of four pieces (a jig, an L-shape, a plate and a peg) and consists simply in mating these four parts together (cf. [Pettinaro & Malcolm 95a]). For convenience we report their two-dimensional diagram in figure 5.5 here below.

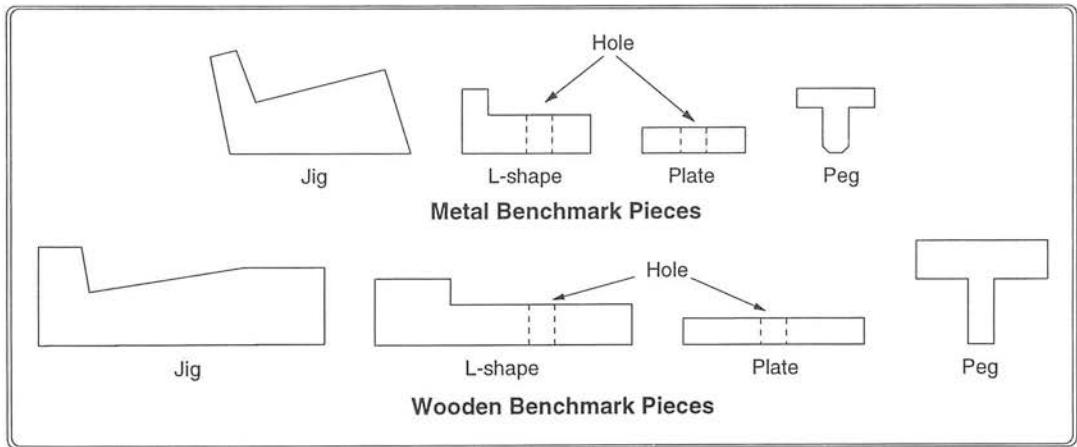


Figure 5.5: Benchmark Family Parts.

Such an assembly task can be accomplished in different ways, however each way depends on the particular strategy adopted for part mating. In this regard, since we

analyze the behavioural components of this assembly task with a bottom up approach (cf. page 77), we divide its discussion in two parts: the partial mating of L-shape, plate and peg (partial benchmark assembly), and the full mating of the four parts (full benchmark assembly). The goal we aim to achieve in both cases is to show that a program for assembling one kit expressed in terms of behavioural modules can be used without any alteration for assembling a similar kit belonging to the same family. In the end, by generalizing the modules required for the partial assembly to perform the full one, we aim to develop ergonomic general-purpose behavioural modules versatile and convenient to be used by common end-user.

Another important reason why we prefer to work with this kind of assembly (cf. subsection 3.2.3 on page 73) is that even in its simplicity it requires one of the most common manufacturing tasks in industry: peg in hole (cf. figure 2.10 on page 53). Such a task is so important in our quest for the basic set of elementary behavioural modules with which to program most of the assemblies¹¹ that we dedicate the following subsection to discuss it. Thus, we divide the rest of this section in three separate discussions: peg-in-hole (subsection 5.2.1 here below), the partial benchmark (subsection 5.2.2 on page 136), and the full benchmark (subsection 5.2.3 on page 147). The results achieved and discussed in these subsections will then be summarized at the end (subsection 5.2.4 on page 154).

5.2.1 Peg in Hole

As soon as we talk about putting a peg in a hole, we think of an extremely simple and trivial task which any normal human being can perform without particular problems. When we try to have the same task carried out by a robot, surprisingly we incur so many difficulties that it makes us realize how complex apparently simple tasks such as this are. Our interest for peg in hole, though, is not confined just to this realization. As showed in an earlier chapter, this task is one of the most common in manufacturing assembly industry (cf. subsection 2.3.1 on page 47), and at the same time one which hides in its simplicity most of the difficulties of assembling parts. But why is it so hard for a robot to carry it out? One of the reasons lies in its limitations in sensing

¹¹ In the sense given in subsection 2.3.1 on page 52.

the surrounding environment ([Tung & Kak 94]), but yet it is not the main one. Any robot is not conscious of what it is doing ([Norman 94]): the only thing that matters is moving its end-effector to a well specified position and once there performing any eventual action like screwing, grasping, spraying, or whatever. In the specific case of peg-in-hole, if a peg is for some accidental reasons wrongly gripped, or if a hole is not exactly in the location expected, a manipulator would be most of the time unable to mate successfully the parts without any human help. What is really missing is a description of the task in terms which enables the robot to cope with the uncertainty embedded in the world ([Torrance 94]). Behaviour-based assembly may be a possible answer, and here we show that we can actually accomplish this task reliably within the terms of such a paradigm.

As we can easily gather by its name, peg-in-hole consists basically in inserting a peg into a hole ([Bland 86]). In this regard, we assume, as most of the researchers tacitly do, to deal just with rigid parts. Nevertheless, there are some cases which involve flexible parts mating ([Zheng *et al.* 91], [Nakagaki *et al.* 95]), however, given its low occurrence in manufacturing industry, we will disregard it in the rest of this discussion.

The great majority of the relevant literature views peg-in-hole as a four-stage process ([Whitney 85]): approach, chamfer crossing, one-point contact, and two-point contact (cf. figure 5.6 here below). During this last a peg may get stuck and two situations

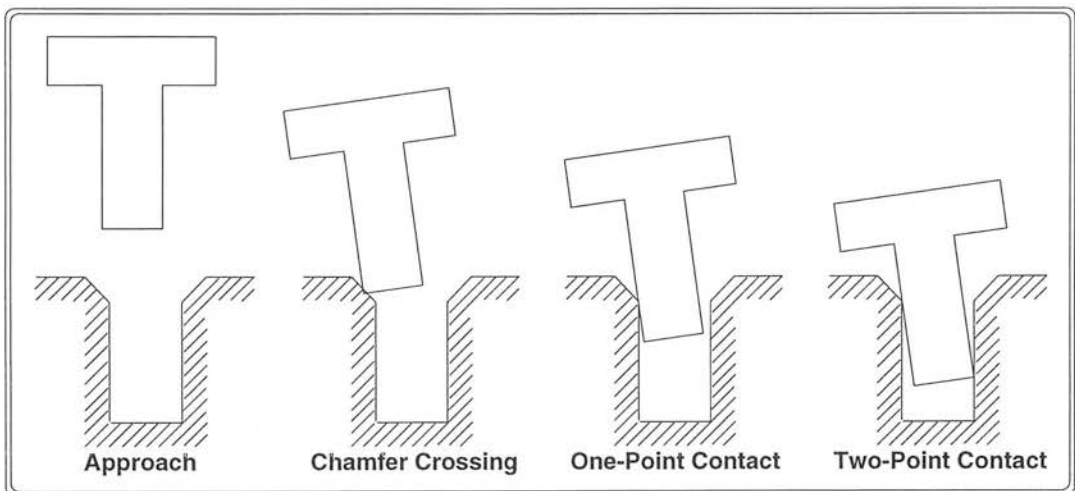


Figure 5.6: Peg in Hole Mating Stages.

may arise: jamming and wedging. The former occurs because of wrongly proportioned forces and moments applied to the peg through the support, and the latter because of linear dependency of the resultant forces at the constraints ([Dupont & Yamajako 94]). In order to limit the occurrence of these situations and to increase peg-in-hole mating success rate, special hardware devices exploiting mechanical compliance have been introduced: the remote centre compliance (RCC) for passive part mating ([Watson 78]), and the instrumented remote centre compliance ([deFazio *et al.* 84]). The latter, which enhances the former by adding active force sensing to correct the insertion, may be viewed as a hybrid between passive and active part mating. In this regard, proper active part mating, which fully exploits sensory data to drive the peg inside the hole, has been the subject of extensive research. A method for gathering information about mating from force sensor is presented in [Söderquist & Wernersson 92]. A high-precision, self-calibrating insertion strategy for cylindrical pegs, which exploits simple and accurate optical sensors, is proposed in [Paulos & Canny 94]. A new modular approach for solving the peg-in-hole problem using a camera is presented in [Kleimann *et al.* 95]. Such a work, which is developed for a complex system made of a 6 degree of freedom manipulator and a dextrous three-fingered gripper, proposes to divide the mating process in three parts: a module responsible for classifying the position of peg and hole with respect to each other, a second module for selecting the right insertion strategy according to the classification, and finally a module for handling the actual insertion by means of 5 primitives (lowering, displacement, shaking, hole-search, and lifting).

Peg-in-hole may be classified according to how many insertion tasks have to be handled at the same time. In this regard, we distinguish two different families: single and multiple peg-in-hole. In this respect, we mentioned in an earlier part of this thesis (cf. subsection 3.2.3 on page 73) that we focus our research just on single peg-in-hole because of its high occurrence rate (cf. figure 2.10 on page 53). As pointed out in subsection 2.3.1 on page 49, there are two classes of single peg-in-hole: round peg in round hole and orientation dependent peg in a matching hole. In this regard, a general treatment on assembly strategies based on hybrid force/position control is presented in [Aspragathos 91] for the large class of peg-in-hole assemblies having a plane of symmetry passing through the insertion axis. Our investigation will be concerned just with round peg-in-hole, and, in this respect, we distinguish two important varieties

whose assembly strategies are extensively analyzed in [Wu & Hopkins 90]: peg into a single hole and peg into two coaxial holes with different diameters¹² (tandem peg-in-hole¹³). Since this last may be viewed as a generalization of the former, and since a general-purpose module performing this task should be capable of coping with the former as well as with the latter, it is important for us to study such a variety. In particular, recalling our first experimental test bed (cf. benchmark assembly family in subsection 3.2.3 on page 74), we notice that the actual mating task involved is a general form of it which implies the two holes to be located on two loose, separate parts. Our research will be concerned with developing a general peg-in-hole capable of dealing with this kind of mating process.

Peg-in-hole, as discussed above, consists for us in inserting a round peg into a round hole. Such an extremely trivial description, though, is very deceiving because it hides an awful amount of unpredicted difficulties caused by uncertainties in the parts, misalignments, friction, *etc.* (cf. [Caine *et al.* 89] and [Wilson & Latombe 92]). But if the description is simple, why do all these difficulties arise? The answer is that the real world is not perfect and an agent operating in it has to cope with the various uncertainties embedded in its environment in order to accomplish useful physical tasks. In our case mating a peg with a hole, although simple to describe, is very hard to be reliably carried out by a machine. Several works have studied how to reduce or at least constrain uncertainty. It was shown that, except for an irreducible 180° ambiguity, some polygonal shapes can be stably grasped in a completely determined orientation without any sensing by performing a sequence of just two squeezes ([Mason *et al.* 88], [Mason 89]). Another interesting work presented a sophisticated two-arm robot system equipped with a vision system (camera) and two force-torque sensors ([Hörmann 92]). Such a system is capable by fusing its sensory readings of planning a sequence of robot commands for grasping and manipulating parts placed in any orientation within its vision field. In this regard, a method of systematically generating visual sensing strategies based on knowledge of the task to be performed is proposed in [Miura & Ikeuchi 95].

¹² The small diameter hole lies below the big diameter one.

¹³ Tandem peg-in-hole may be generalized to an N -tandem peg-in-hole by assuming a sequence of coaxial holes with decreasing diameters. However, this more complex form will not be considered here, because the strategy to accomplish it may be trivially deducted from the one to accomplish the simple tandem one.

Vision is no doubt a powerful tool for driving a peg towards a hole during the approaching stage (cf. figure 5.6 on page 117), but it is unable by itself to solve jamming and wedging during the stage of the two-point contact. In this respect, as mentioned above, force-torque sensors are more useful.

Observing the way in which we as human beings insert a peg into a hole, we notice that such a task is accomplished first by locating the hole, and then performing the mating. We label these two task components with the names hole search and peg insertion, respectively. However, since we rely very much on our eyes to locate the hole, and since we are assuming to equip our agent with a simple touch sensor and no visual sensing (cf. subsection 3.2.4 on page 80), we need to find a better example of modelling our peg-in-hole. A blind man may in a certain way represent a better model to follow: he does not know where exactly the hole is but he may have a rough idea where it may be found. Thus, first he tries to put the peg there, and then, in case of failure, he starts searching for the hole in the surrounding area until he finds it and proceeds with the insertion of the peg. Following the line of this example and assuming our agent to have already moved its gripper above the location where the hole is supposed to be, we can model our peg-in-hole with three components:

- one for putting the peg down inside the hole,
- one for searching for the hole, and
- one for the peg insertion.

The first module component attempts the insertion by simply putting the peg down. In case the peg shaft as a result of this action is at least 95%¹⁴ inside the hole, we can consider the peg fully mated with the hole. In case, instead, this does not happen, our agent should start looking for the hole in the neighbouring vicinity until one of the following situations occurs:

- 1) a hole is found,
- 2) search failed because of an obstacle, or

¹⁴ We do not require 100% because it is important to leave some room for tolerance.

- 3) search ended without finding a hole.

In case a hole is found, an insertion module, whose outcome may be either peg successfully mated or insertion failed, is then performed. Summarizing, we can classify the possible assembly situations in which our agent may be left at the end as follows:

- 1) peg successfully mated with a hole,
- 2) peg insertion aborted,
- 3) obstacle detected during search for hole,
- 4) hole not found.

Since we are not employing sophisticated sensory equipment such as, for instance, a vision system, each module component has to rely on some knowledge which has to be given as input. The first one requires to know the distance between the peg head and the surface whereon the hole is located, the second one needs to know the tilt angle between the plane of the hole and the X-Y plane of the robot frame, finally the last one requires to know the length of the peg shaft (cf. figure 5.7 here below). Notice that

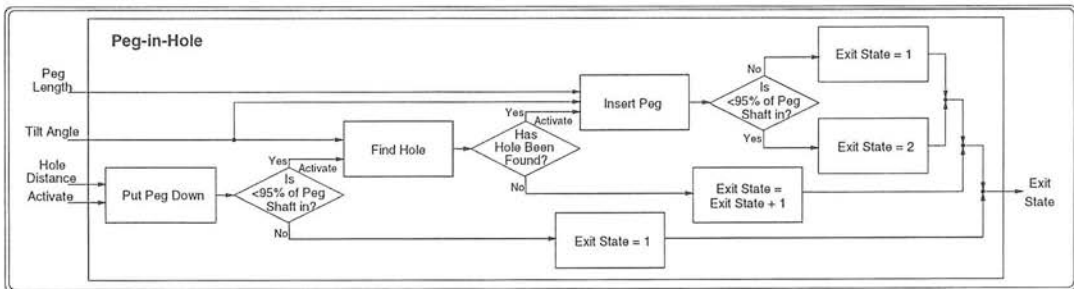


Figure 5.7: Peg in Hole Diagram.

the module `Find Hole` returns always an exit state code (*i.e.* 1, 2, or 3) which is then attributed to the variable `Exit State`. Thus, such a variable does not require to be initially set. In case the outcome of the search for the hole is positive (`Exit State=1`), then the diamond box activates the execution of the insertion module, otherwise it just increments the variable `Exit State` by 1, whose value may be 2 or 3, in order to discriminate these outcomes from those in the other branch of the diamond box.

Notice also that if the peg shaft is not almost totally inside the hole, it is better to take it out and proceed to the search for the hole even if that had already been found. The reason is that the little impacts caused by the hopping search (cf. section 5.2.1.1) can help to realign the two detached coaxial holes.

As last remark, we have to point out that we used in the diagram above an input line to the three modules which we labelled with *activate*. Such a line has to be interpreted as the flow of the control of the robot manipulator.

At this point, in order to understand the complexity of the problems involved with the accomplishment of peg-in-hole, we divide the rest of the discussion in two parts: the strategies to find a hole (subsubsection 5.2.1.1 here below) and the strategies to insert the peg (subsubsection 5.2.1.2 on page 128). We reserve the right to discuss the first module component of the strategy outlined above later in subsection 5.2.2 on page 143.

5.2.1.1 Strategies to find a Hole

Searching for a hole may be regarded as the approaching stage in the classic peg-in-hole decomposition discussed earlier in this subsection (cf. page 117). Most of the research has mainly concentrated on the stages involving the insertion leaving this specific sub-problem quite unexplored. However, locating a hole is as important as the actual insertion of a peg, because a good localization avoids most of the difficulties caused by jamming and wedging in later stages. In this regard, a special purpose hardware module is developed in [Martínez & Llario 87] to identify and locate round holes in three dimension by employing a stereo vision system. The strategy used is based on the matching of virtual points corresponding to the centres of the holes in the stereo pair. Another interesting technique for a round peg-in-hole is presented in [Paulos & Canny 93]. The approach proposed centers around recording the location of four points on the edge of the hole using a reflective optical sensor placed in an opportune position below the manipulator gripper. The sliding motion of the beam emitted by the optical source first along the X-axis and then along the Y-axis allows to find these four points. Once they have been recorded, the boundary of the hole, and therefore its centre, is determined. However, in case no edge point is initially detected, a spiral or grid based search strategy is performed in the neighbourhood until one is

found. Following this method, a similar technique employing a force-torque sensor is developed for round peg-in-hole in [Bruyninckx *et al.* 95]. The strategy proposed requires to lean a peg so that its axis and that of the hole are largely misaligned. Once the contact between peg and hole has taken place, three points are located on the rim of the hole by reading the sensory data of the force-torque sensor placed beneath the assembly (cf. figure 5.8 here below). At this point, using their position coordinates,

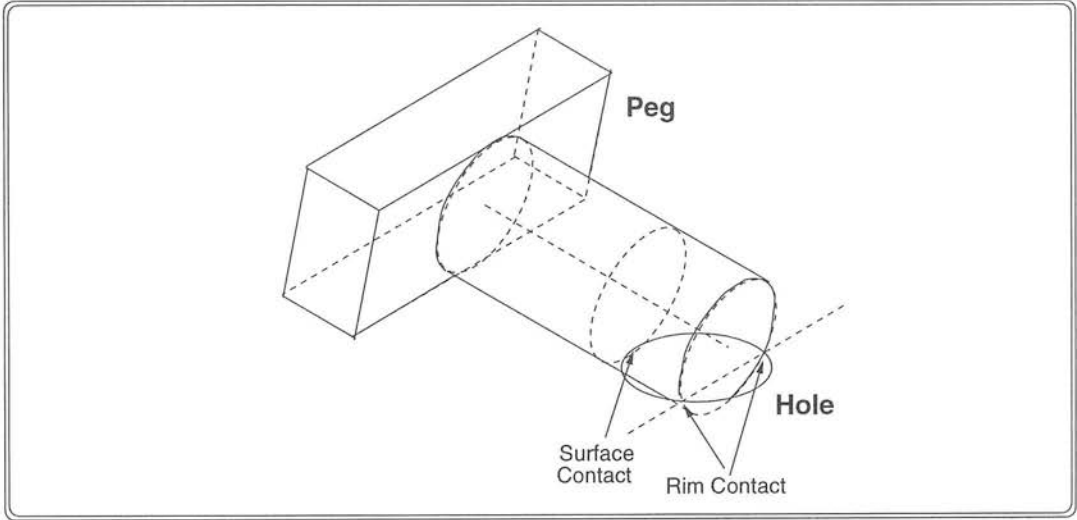


Figure 5.8: Peg on Hole.

we can determine the location of the centre of the hole with respect to the robot frame axes, and then carefully align the peg with the hole axis. This method is quite effective, however it is limited by the tacit assumption to have a peg located above its matching hole. The method does not in fact undertake any search for the hole in case this does not lie beneath the peg.

The strategy we are proposing in this subsection is based on some ideas from the works outlined above. First of all, let us recall that we are not employing any vision, optical or proximity sensors: we are simply using a cheap differential touch sensor wrapped around the fingers of our robot gripper. Such a sensor is capable of detecting just variation of forces, in other words contact events. Thus, the search is bound to be performed by exploiting on/off information from impacts between parts. In this respect, several solutions are possible using this kind of sensor, but they all require to scan the surface where the hole is, and therefore testing its presence, by hitting

either directly the surface along its normal axis with a finger, if this is thin enough, or indirectly with a thin stick (probe), in which case contacts are sensed as forces propagated through the probe. Since the fingers we are employing are very crude tools to be used for such a search, we opt for the second solution. If a peg tip is sharp, we can actually use the peg itself as a probe. Unfortunately, in general this is not the case, nevertheless we can still use it as a probe by leaning it with respect to the normal to the surface of the hole. This always guarantees a sharp contact edge for both chamfered and chamferless pegs.

At this point, before discussing a few relevant solutions to the problem of search for the hole, let us make one more remark. As detailed above, any search is bound for us to be made by hopping along the surface of the hole and sensing any contact event between probe and surface. A hole would be detected by the absence of any impact within a certain distance. As mentioned above, we supposed the surface of the hole to be tilted with respect to the X-Y plane of the robot frame axes. In this regard, we have to point out that slopes greater than 15° allow the metal parts of our benchmark under the push of gravity to overcome friction and slip upon the slope, whereas slopes smaller than 15° are not enough to allow any slipping to take place. This is an important observation, because the aluminium assembly we chose to experiment with (cf. metal benchmark in subsection 3.2.3 on page 74) has a hole which lies on a slope of 15° which is just on the edge of making any part (peg included) slip upon the surface. Thus, we should not be thinking of the peg shaft as an accurate probe device for the search. Each time, in fact, an impact takes place, the peg tip under the downwards push of the robot arm undergoes a surface force which makes the tip slightly slip along the slope and the peg rotate about the Y-axis of its head (cf. figure 5.9¹⁵ on page 125), and this in a long run may well cause our agent to tilt the peg so much that it will acknowledge a hole where there is not one, simply because it cannot detect a contact between peg tip and slope. Unfortunately, such a problem cannot be avoided by alternating tip side because this would introduce more uncertainty, nor by aligning the peg orthogonal to the slope, because, as mentioned earlier, a peg tip is relatively large and therefore not a very accurate tool to be used for a search.

¹⁵ Notice that an equivalent diagram applies for a chamfered peg.

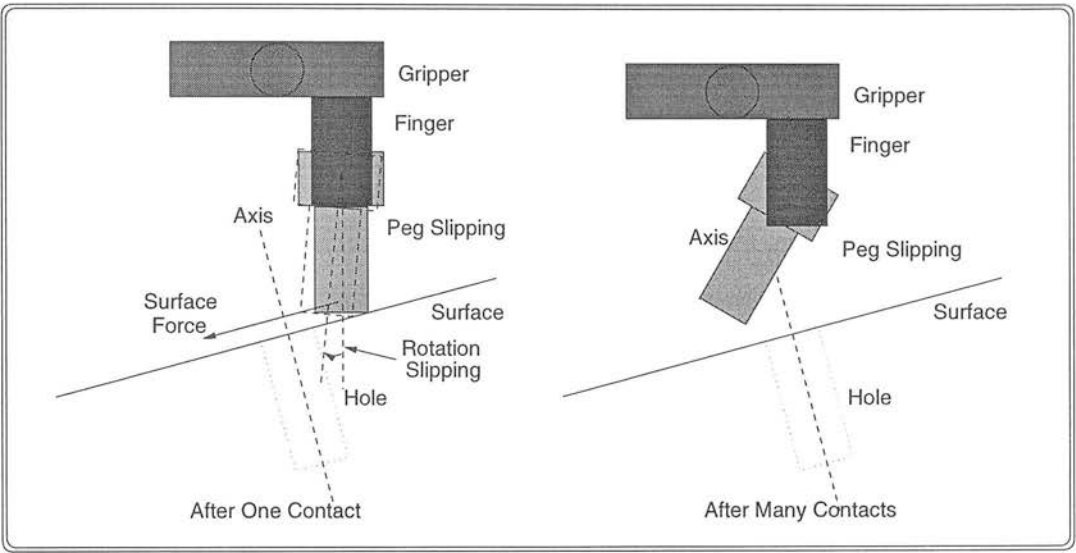


Figure 5.9: Example of Chamferless Peg Tilting because of a Contact.

Let us now return to focus our attention to the search strategy itself. We mentioned above that several search path solutions are possible. In this respect, we notice that they may be performed following any trajectory. However, the easiest ones to implement are those which are shaped along regular lines, and, among these, those particularly appealing to us are zigzag and spiral paths (cf. figure 5.10 here below). The two

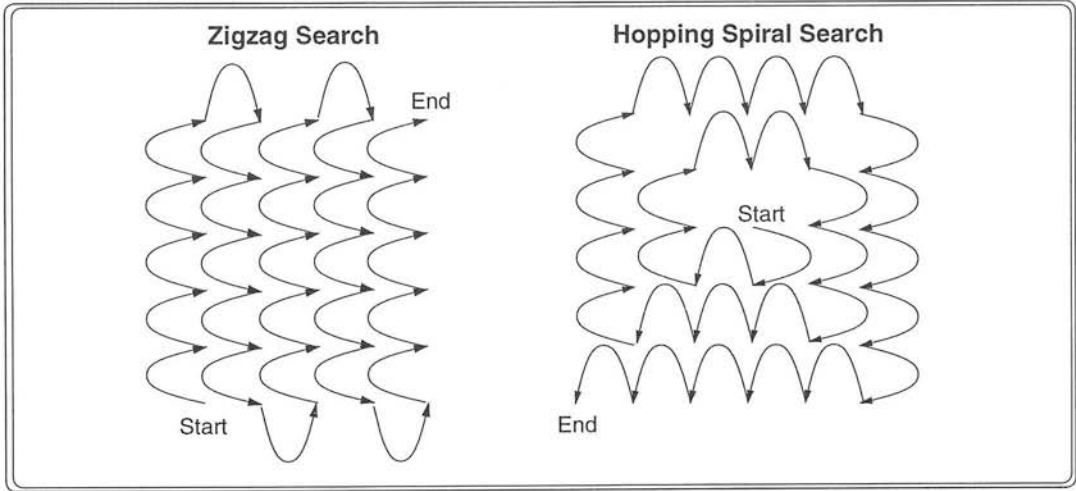


Figure 5.10: Zigzag and Spiral Searches Paths.

following emphasized paragraphs will discuss these two particular solutions. In any case, it is worth recalling that a hole search is triggered only if a peg does not find a

hole straight away. However, in order to avoid never ending loops, once any search is started, it has somehow to terminate with an exit state of either *hole found* or *search failed*.

Zigzag Search This kind of search is very simple (cf. figure 5.10 above) but not very practical because of its relatively slow speed to converge to the target (hole).

In order to test this strategy we set two experiments involving an L-shape part and a plate (cf. subsection 3.2.3 on page 74) which are made of metal for the first experiment and of wood in the second one. The two parts are in each case placed on a tilted jig made of the same material.

We started with the metal parts and, by running 40 searches, we recorded 38 successes distributed as follows: 10 after 9 steps, 18 after 14 steps, 6 after 20 steps, and 2 after 26 steps. The two failures were due to a wrong acknowledgement of a hole caused by a peg overtilted because of the high number of contacts.

We repeated the same experiment using wooden parts, and again, by running 40 searches, we recorded 39 successes distributed as follows: 11 after 8 steps, 19 after 13 steps, 7 after 21 steps, and 2 after 25 steps. The only failure recorded was even in this case caused by a wrong acknowledgement of a hole for the same reason.

Spiral Search This strategy pursue the search for a hole by following a spiral path. There are many kinds of spirals (*e.g.* ellipsoidal, circular, square, rectangular), but just two of them are very easy to implement: square and rectangular spirals (cf. figure 5.10 on page 125). The former, since it is not particularly biased on any directions, has in general better performance than the latter. The spiral is carried out by hopping along the surface in the same fashion as a zigzag search, but with the difference of turning direction of hops 90° clockwise after a certain number of them has occurred. This number is progressively increased after a direction turning has taken place until a limit number of hops, which can be passed as an input parameter, is reached.

In order to test this strategy, we set two experiments similar to those made for the zigzag strategy using metal and wooden parts. We started by placing the metal parts

containing the holes on a tilted jig. As done before, we ran 40 searches and recorded 40 successes distributed as follows: 2 after 1 step, 19 after 3 steps, 16 after 6 steps, and 3 after 10 steps. We repeated the same experiment using wooden parts and recorded again 40 successes distributed as follows: 4 after 1 step, 21 after 3 steps, 14 after 5 steps, and 1 after 11 steps (cf. table 5.1 on page 128).

Comparison of Search Strategies Considering the two search for hole strategies discussed in the previous two paragraphs, we have to make a few comments. First of all, we have to point out that zigzag requires in general a large number of contact events to be detected. This is particularly important because, as observed on page 124, a peg may get more and more tilted each time an impact takes place, and it may happen that in the long run a hole is wrongly acknowledged (cf. figure 5.11 here below). This

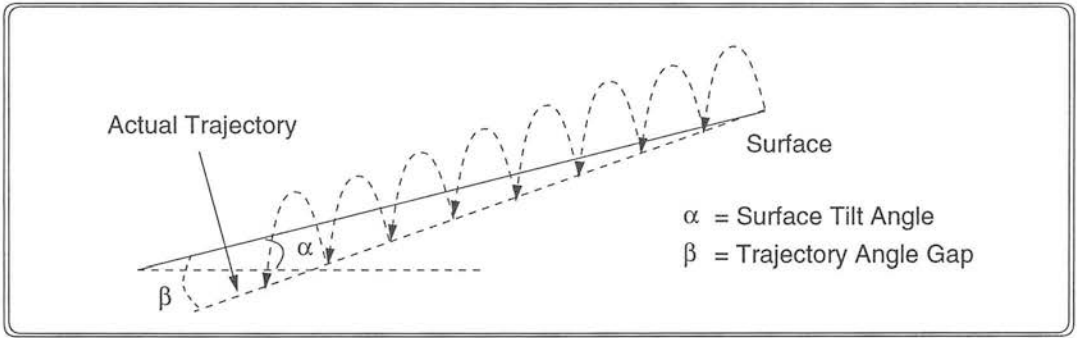


Figure 5.11: Trajectory followed during the Search which wrongly acknowledges a Hole.

drawback is difficult to be avoided even using more sophisticated strategies, because it is intrinsic of the kind of sensor used. The solution of having tighter grips is not very effective both because of the small friction between peg head and fingers' skin, and because of the small area actually gripped. Increasing friction does not avoid peg slippings when each impact takes place, therefore the best way to limit its negative effects, *i.e.* the failure rate, and improve the reliability of the solution is by drastically reducing the number of hops required for the search, in other words, by switching to a better search path. In this respect, a spiral search is in general far better than a zigzag search, because it optimizes the area of the search (cf. table 5.1 on page 128). In fact, assuming to start in the proximity of a hole, we will not need to complete the entire path in order to find a hole. The worst case occurs only when a hole is located very

Steps	Metal Parts		Wooden Parts	
	Search Strategies		Search Strategies	
	Zigzag n°	Spiral n°	Zigzag n°	Spiral n°
1-5	0	21	0	39
6-10	10	19	11	1
11-15	18	0	19	0
16-20	6	0	0	0
21-	2	0	9	0
Success Rate	38/40	40/40	39/40	40/40

Table 5.1: Search Strategies Experimental Data.

far away from the centre of the spiral, in which case a zigzag search is more efficient. Thus, in general a spiral search outperforms a zigzag one. However, we have to point out that, although following a better path, it does not resolve the problem of unwanted supplementary peg tilting, which always lurks and may lead any search to a failure. The reason is that the optimization regards just the search and not the way in which the presence of a hole is tested.

5.2.1.2 Strategies to insert a peg

Peg insertion is the other major research topic of the peg-in-hole problem and may be regarded as the one- and two-point contact stages of the classic decomposition discussed on page 117. The particular task carried out in this phase consists in actively inserting a peg (in our case a round one) into a matching hole. In this respect, we may distinguish two relevant outcomes: successful insertion, or failure caused by jamming, wedging, or unpredicted situations.

Although the location of the hole at this stage of the task is assumed to be known, mating a peg with its correspondent hole is not as trivial as it may at first appear. It involves many complex operations whose accomplishment is too often taken for granted. The main difficulty in solving the peg-in-hole problem is to find robust and reliable strategies to allow a machine to accomplish the task, that is to put it in condition of recovering from error situation. The relevant literature reports many solutions to the peg insertion problem, but they all resort to more or less complex hardware. Some techniques employing hybrid force-position control are described in [Strip 88] for a

wide variety of shaped pegs, and some strategies for chamferless insertion of planar and prismatic rectangle pegs are proposed in [Caine *et al.* 89]. Other research studies tackled the insertion problem more theoretically by conducting analytical work on the various difficulties involved with insertions. In this regard, an interesting analysis of the dynamics of a peg-in-hole insertion is carried out in [Shahinpoor & Zohoor 91] where it is shown that the conditions of successful insertion are determined by a set of generalized inequalities. Another work analyzing uncertainties involved with peg insertion is presented in [Paetsch & vonWichert 93] where the use of a force feedback loop for driving a multifingered gripper is proposed as a solution for resolving them. As a general remark we have to point out that all the works outlined above share the tacit assumption of dealing with rigid parts. In this respect, a study involving compliant parts worth mentioning is reported in [Meitinger & Pfeiffer 94] where the forces and torques acting on a gripper during a mating process are modeled and analyzed.

As realized from the few examples mentioned above, there is a great variety of strategies resolving the insertion problem. However, there is not as yet any robust and reliable general solution to such a problem. The great majority of those proposed and developed applies to a particular class of peg shapes and relies on more or less sophisticated sensor availability. In this respect, as stated earlier, we are employing a very simple touch sensor and we are restricting our investigation just to round peg insertions (cf. page 118). Notice, however, that our concern is not limited with what we labelled as *simple* peg-in-hole, but it extends to cover a form of tandem peg-in-hole involving two loose parts tilted with respect to the X-Y plane of the robot frame system. In this regard, recalling the general structure of the task outlined earlier (cf. subsection 5.2.1 on page 120), we have to point out that a peg insertion would be initiated just after search for a hole, which, if successful, implies part of the tip of the peg to be inside the hole. Thus, we can assume without any loss of generality to start the insertion process with a peg partially inserted.

As mentioned on page 121, we need two parameters¹⁶ as input: the length of the peg shaft and the tilt angle. The latter is necessary in order to derive the insertion axis, whereas the former in order to realize when a peg is successfully inserted. In

¹⁶ Notice that the assumption of using a vision system would make these parameters redundant.

this respect, the test of successful insertion is accomplished by comparing the distance travelled by the manipulator wrist with the rest of the peg shaft length which at the beginning is still out of the hole.

The rest of this subsection is dedicated to present and discuss three possible solutions to the round peg insertion problem (cf. figure 5.12 here below), which will be

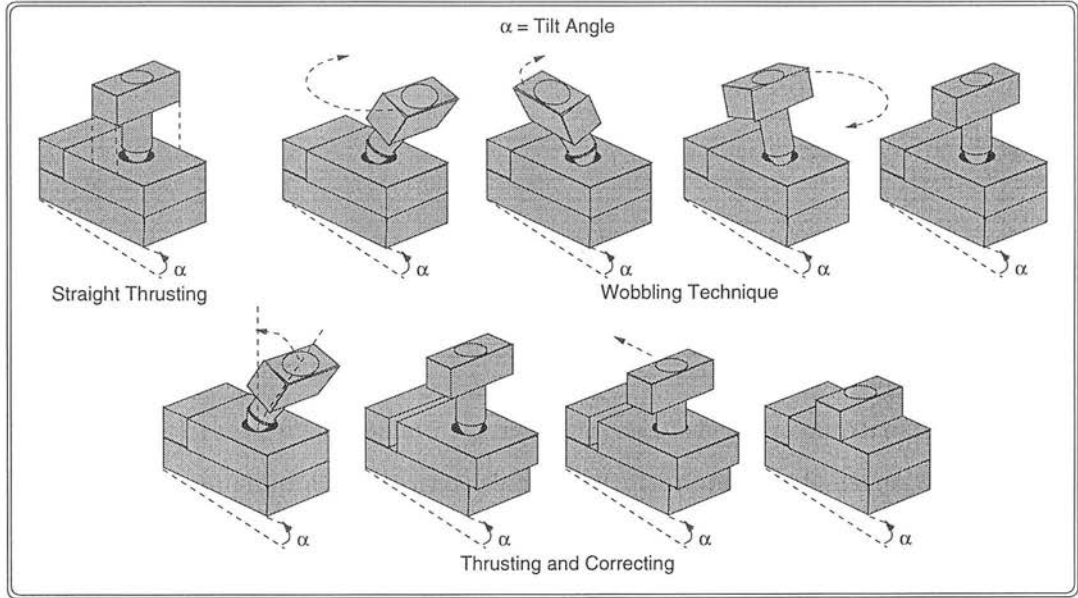


Figure 5.12: Different Strategies for solving the Peg Insertion Problem for a Chamfered Peg, but a similar Diagram applies for a Chamferless one.

opportune compared in order to select the one which best guarantees robustness and reliability. For sake of generality we assume in the rest of this discussion to carry out the insertion α° tilted with respect to the X-Y plane of the robot frame system.

Straight thrusting The first solution we examine consists in thrusting a peg into a hole directly without caring of any possible jamming or wedging. If a peg gets stuck inside a hole, its insertion is aborted and the whole task has to be repeated.

In order to test this solution, we set two different experiments using two kinds of parts (cf. L-shape and plate in subsection 3.2.3 on page 74) made of different material (metal and wood) and having coaxial holes of the same diameter.

To start with, we placed the two metal parts one above the other on a jig 15° tilted with

respect to the table, and, in line with the assumption of working with loose tandem peg-in-hole (cf. page 119), we did not fix them to each other. As regards the peg, we assumed it to have its tip slightly dipped inside the hole (cf. above). We ran a set of 50 trials for this experiment and we observed 39 successful matings of both holes (78% success rate). The 11 failures recorded were caused partly by a misalignment of the two holes' axes and partly by jamming and wedging. Indeed, we noticed that the metal peg successfully penetrated completely the first hole in 5 out of the 11 failures, but it got stuck at the rim of the second hole because of the misalignment. Thus, considering just the first hole, we may say that we recorded 44 complete matings of metal peg in one hole (88% success rate). As regards the remaining 6 failures, we noticed that four of them were caused by wedging and the other two of them by jamming.

As mentioned above, we ran a second experiment involving similar parts made of wood placed on a jig 10° tilted with respect to the table. By running a set of 50 insertion trials as did before, we observed 41 successful matings of both holes (82% success rate). Analyzing the 9 failures recorded, we noticed that they were caused this time by axis misalignment (5 failures), jamming (1 failure), and wedging (3 failures). Thus, considering just one hole, we may say that we recorded 46 successful wooden peg in one hole (92% success rate).

The results of these experiments are all reported in table 5.2 on page 134.

Wobbling technique The second solution we consider is what we may label as insertion by wobbling. It consists in deliberately changing the direction of peg insertion by rotating a peg about both the insertion axis and its tip. The algorithm describing it may be outlined as follows:

- tilt peg with respect to the surface of the hole,
- repeat
 - push peg inside hole,
 - rotate peg anti-clockwise slightly about the hole axis,
 - increase peg tilting,
- until most of the peg shaft is inside the hole.

The idea behind this technique is to resolve jamming and wedging by taking advantage of the coupling between peg and hole. As a remark, we have in fact to point out that every time a direction of insertion takes place, a different two-point contact is determined, and in turn this allows more peg shaft to slip inside. At the end of the wobbling process most of the shaft is inside the hole and both peg and insertion axes are aligned.

In order to test this technique, we set two experiments similar to those described before for the straight thrusting solution. We started with the metal parts on the metal jig and, by performing 50 wobbling insertions, we recorded 44 successful matings of the two holes (88% success rate). As regards the 6 failure cases, we observed that 3 were due to axis misalignment, 1 to jamming, and 2 to wedging. Thus, considering just the first hole, we recorded 47 successful insertions by wobbling (94% success rate).

The second set of 50 wobbling insertions performed with the wooden parts on the wooden jig showed similar results: 44 successful matings of both holes (88% success rate) and 6 failures of which 4 were due to axes misalignment, 1 to jamming, and 1 to wedging. As far as the first hole is concerned, we recorded 48 successful insertions by wobbling (96% success rate).

Also in this case the results of these experiments are all reported in table 5.2 on page 134.

Thrusting and correcting The third solution we examine may be regarded as an optimized version of straight thrusting whose main drawback, as showed earlier, was its inability to resolve a stuck situation which mainly happens at the rim of the second hole (cf. figure 5.13 on page 133). This strategy, which assumes that any misalignment between the two parts involved with the peg mating is within $\frac{1}{2}mm$ along the X-axis or Y-axis but not along both¹⁷, adds the capability of adjusting a peg with the axis of the second hole when a stuck situation caused by misalignment occurs (cf. figure 5.12 on page 130). This is achieved by attempting in sequence at most four shiftings of a peg from its initial position: one of $\frac{1}{2}mm$ along the X-axis, one of $-\frac{1}{2}mm$ along the

¹⁷ This assumption is consistent with the with the fact that some misalignments are caused by rotations of the peg about its tip along the X-axis after a search for a hole has succeeded.

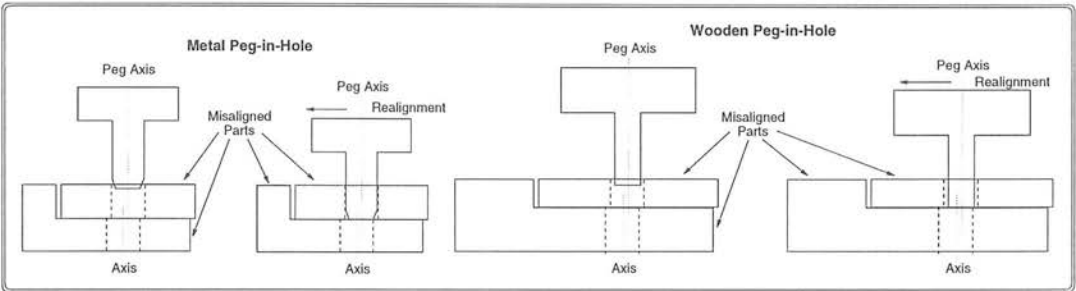


Figure 5.13: Diagram of the Stuck Situation.

X-axis, one of $\frac{1}{2}mm$ along the Y-axis, and finally one of $-\frac{1}{2}mm$ along the Y-axis (cf. figure 5.14 here below).

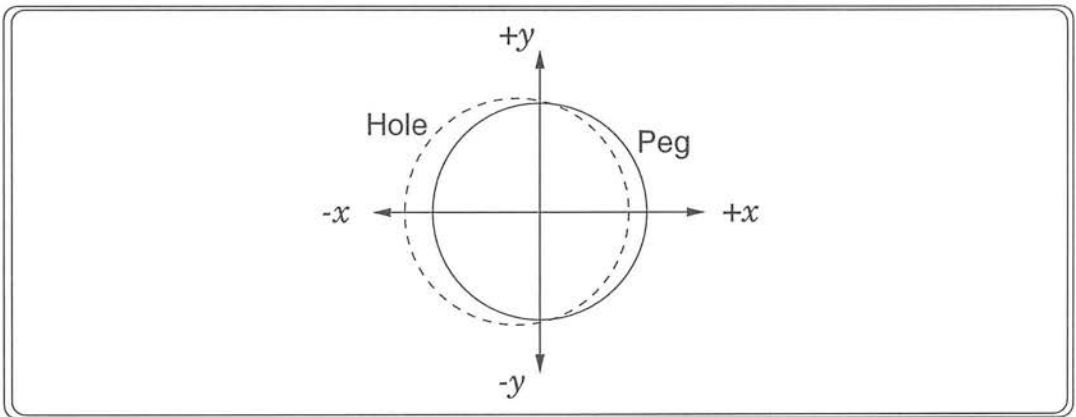


Figure 5.14: Diagram of Shifting Directions.

In order to test this solution, we set the same two experiments we did for the other two insertion strategies discussed earlier: two parts with coaxial holes placed loosely on a tilted jig and a peg with its tip slightly dipped inside the hole. We started with performing 50 insertions with the metal parts and we recorded 48 successful matings of both holes (96% success rate). The two failures were due one to wedging and one to an axis misalignment larger than $\frac{1}{2}mm$. Thus, considering just the first hole, we may say that we obtained 49 successful matings of peg in one hole (98% success rate). By repeating the previous experiment with wooden parts, we recorded 49 successful matings of both holes (98% success rate). The only failure recorded was due to wedging.

The results of these experiments are all reported in table 5.2 on page 134.

Comparison of Insertion Strategies Having described the three insertion strategies in the previous three paragraphs, we can now compare and discuss the experimental results we obtained. First of all, let us summarize the data for the metal and wooden tandem peg-in-hole remembering that the total number of trials for each strategy was 50 (cf. table 5.2 here below).

Insertion Strategies	Metal Tandem Peg-in-Hole							
	Successes		Failures					
			Misalignment		Jamming		Wedging	
	n°	Rates	n°	Rates	n°	Rates	n°	Rates
Straight Thrust	39	78%	5	10%	2	4%	4	8%
Wobbling Technique	44	88%	3	6%	1	2%	2	4%
Thrust & Correct	48	96%	1	2%	0	0%	1	2%
	Wooden Tandem Peg-in-Hole							
	Successes		Failures					
			Misalignment		Jamming		Wedging	
	n°	Rates	n°	Rates	n°	Rates	n°	Rates
Straight Thrust	41	82%	5	10%	1	2%	3	6%
Wobbling Technique	44	88%	4	8%	1	2%	1	2%
Thrust & Correct	49	98%	0	0%	0	0%	1	2%

Table 5.2: Experimental Data of Tandem Peg Insertion.

As a general comment on the results shown above, we have to say that the material which the the parts were made of affects the performance of the three strategies. The insertions performed using the wooden parts showed a relatively higher success rate and lower failure rate caused by jamming and wedging. We can explain this outcome by observing that wood despite having a coefficient of friction higher than metal is actually softer. Thus, several cases of jamming and wedging are resolved by a little deformation of the peg at the level of its tip.

As regards the strategies themselves, there are a few remarks which we have to point out. Straight thrust may be regarded as the simplest of the three strategies from the implementation point of view, however, as can be seen from the table above, it is neither very reliable nor very robust. In this respect, wobbling showed better performance, but it is unfortunately more complex to be implemented and requires a manipulator agent which is capable of at least two rotations at the wrist level: one along the Z-axis and one along the X- or Y-axes. This characteristic makes it not so appealing to

be developed as a more general tandem peg insertion. The third strategy examined (thrusting & correcting) retains the simplicity of the straight thrust but, besides, it adds the capability of resolving slight axis misalignment between peg and second hole, which were one of the main causes of failure. However, the misalignments which can be corrected are limited to $\frac{1}{2}mm$ along the X-axis or Y-axis but not along both. Notice that the success rate also depends on the relative peg and hole sizes, the amount of peg tapering and hole beveling. In this regard, we have to say that the strategy was tested on rather short peg shafts with a diameter/length ratio of about 0.4.

At this point, let us summarize the experimental data relative to the first hole (cf. table 5.3 here below). As already stressed before, considering just the first of the two

Insertion Strategies	Simple Peg-in-Hole							
	Metal				Wooden			
	Successes		Failures		Successes		Failures	
	n°	Rates	n°	Rates	n°	Rates	n°	Rates
Straight Thrust	44	88%	6	12%	46	92%	4	8%
Wobbling Technique	47	94%	3	6%	48	96%	2	4%
Thrust & Correct	49	98%	1	2%	49	98%	1	2%

Table 5.3: Simple Peg in One Hole Experimental Data.

tandem peg insertions, we notice that the three strategies have a higher success rate, and once again thrust & correct outperformed the others. Indeed, this last within its limits of applicability¹⁸ was the only one among the three of them which was not affected by the specific material of the parts. This particular characteristic makes such a strategy very appealing.

At this point, taking into account the different success rates for both tandem and simple peg-in-hole relative to each strategy reported in tables 5.2 and 5.3, we can conclude that thrust & correct is the most reliable and robust among them, and, because of this, it is the one which we select to be developed as our peg insertion module (cf. peg-in-hole diagram in figure 5.7 on page 121).

This concludes the discussion of peg-in-hole. We are now ready to focus attention back to the benchmark assembly.

¹⁸ Misalignments of the coaxial holes within $\frac{1}{2}mm$ along either the X-axis or the Y-axis.

5.2.2 Partial Benchmark Assembly

The benchmark assembly, as described at the beginning of this section (cf. page 115), consists first in placing two hollow parts with coaxial holes (L-shape and plate) on top of a tilted jig and then in fixing them by inserting a peg through the common hole. However, in line with our choice of working with a bottom up approach, we disregard the jig in this particular part of the document and consider the assembly of just L-shape, plate, and peg (partial benchmark assembly). Indeed, we consider two different sets of these parts: one made of metal (aluminium) and one made of wood (cf. figure 5.15 here below).

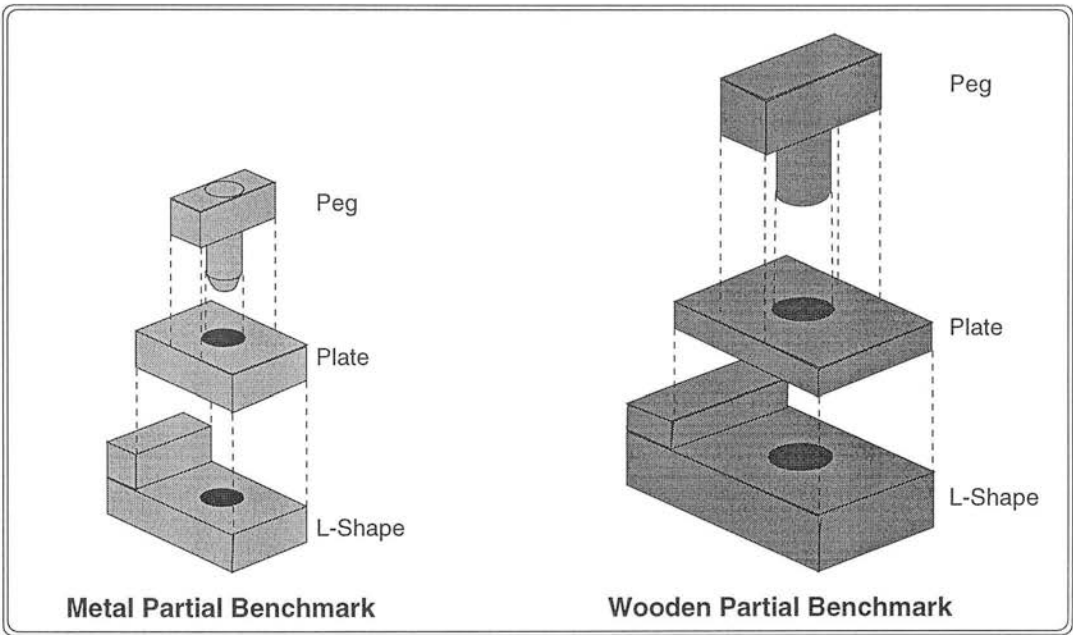


Figure 5.15: Metal and Wooden Partial Benchmarks.

The aim which we want to achieve by experimenting with this simplified version of the benchmark assembly is on the one hand to get a clear understanding of the problems involved in this task, and on the other to identify the behavioural components required to accomplish it. The final goal of the whole assembly experiment is to generalize opportunely these components so that they can be used to assemble the metal benchmark kit as well as the wooden one.

The description of the task is very trivial: it consists first in stacking a plate on top of

an L-shape, and then in inserting a peg into the hole on top of the plate. Taking into account that each part needs to be collected before being stacked, we can view this assembly as a typical pick-and-place problem.

There are two general assumptions which are now worth pointing out: first, the L-shape part is fixed onto the working table, and second, the manipulator gripper is equipped just with a simple differential touch sensor on each finger (cf. page 80). The former is important for constraining the uncertainty due to eventual part slippings along the table, whereas the latter to show that the entire assembly task itself does not require very complex sensing capabilities in order to be accomplished.

Since we want the robot to be able to accomplish both metal and wooden partial benchmarks, we have to devise a way to enable our manipulator agent not to be bound to particular part sizes within the gripping capability of its end-effector (max. 103 *mm*). Because we assume to use an electric gripper capable of reading the gap between its fingers ([Pettinaro & Malcolm 94]), we can actually make our agent aware of the particular set which one part belongs to. The only information which cannot be obtained by using solely such a gripper is the length of the peg shaft. As discussed in subsubsection 5.2.1.2 on page 128, this information is crucial for determining both when a peg is fully inserted into a hole, and when a peg, because of jamming or wedging, is actually stuck inside it. In this regard, it is worth pointing out that we would not require this information if we were employing more powerful sensing capabilities like vision, in order to accomplish successfully this assembly, because an agent would be able to obtain it autonomously by simply processing its sensory data. However, even if we employ a very simple sensory system, we can still allow our agent to measure the peg shaft length by itself. This can be achieved first by gripping the peg with its shaft outwardly parallel to the gripper Z-axis, and then by putting it down on the table: the length of the shaft is computed as the difference between the vertical component of the location where the peg tip touches the table with respect to the robot frame system and the table height itself (cf. figure 5.16 on page 138). Notice that the technique assumes to find the position of the table surface by shifting the gripper +15 *cm* away from the position of the jig along the X-axis.

Recalling now the description of the assembly task which we gave a while back, we

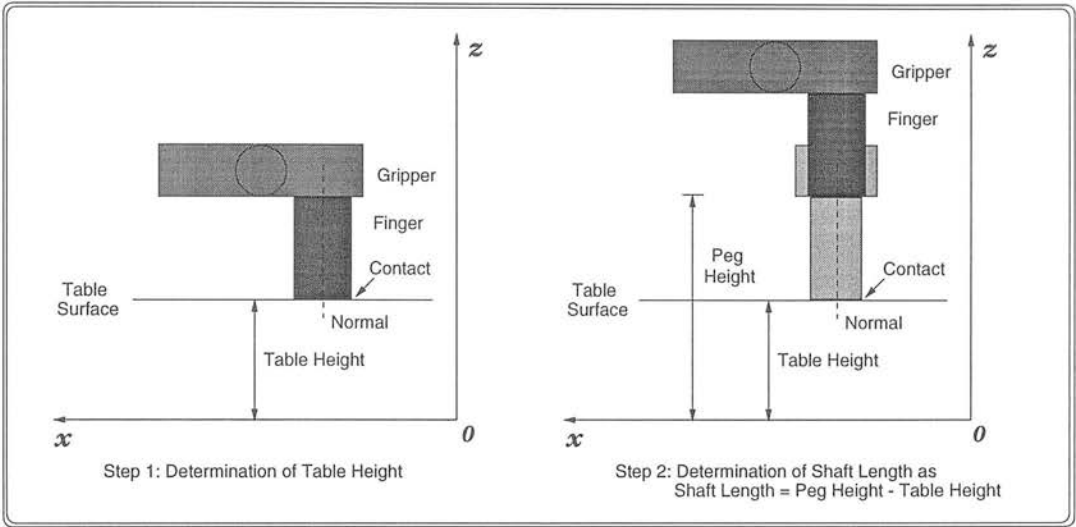


Figure 5.16: Peg Shaft Length Determination Technique.

can say that we need first of all a module for collecting the plate, and then a module for depositing it on top of the L-shape. In case both modules are successfully carried out, we need another collecting module to pick up the peg from its home location, and a peg-in-hole module to insert it inside the tandem hole passing through plate and L-shape.

As discussed in subsection 5.2.1 on page 121, our peg-in-hole may terminate in one of four possible states: peg successfully mated, insertion aborted, obstacle detected during search for hole, and hole not found. Thus, considering the other outcomes of the entire assembly process (plate not found, stacking failed, and peg not found), we can say that the partial benchmark assembly may terminate in one of 7 possible states (cf. table 5.4 here below).

Partial Benchmark	
Code	Exit States
1	peg inserted in hole and assembly successfully completed
2	peg insertion failed
3	Obstacle detected during search for hole
4	Hole not found
5	peg not found
6	plate stacking failed
7	plate not found

Table 5.4: Partial Benchmark Outcomes.

As regards inputs, we require two parameters to perform the first collecting module (gripping command and plate location), two to perform the stacking (stacking command and L-shape location), other two to perform the second part collection (peg), and three to perform peg-in-hole (peg length, hole distance, and tilt angle). In this respect, we have to point out that this simplified version of benchmark assembly is not tilted, thus the tilt angle parameter is set to 0° .

We report here below in figure 5.17 the behavioural decomposition of the assembly task discussed above. Notice that such a description omits the measurement of the peg shaft because strictly speaking it is not actually part of the assembly process. As last

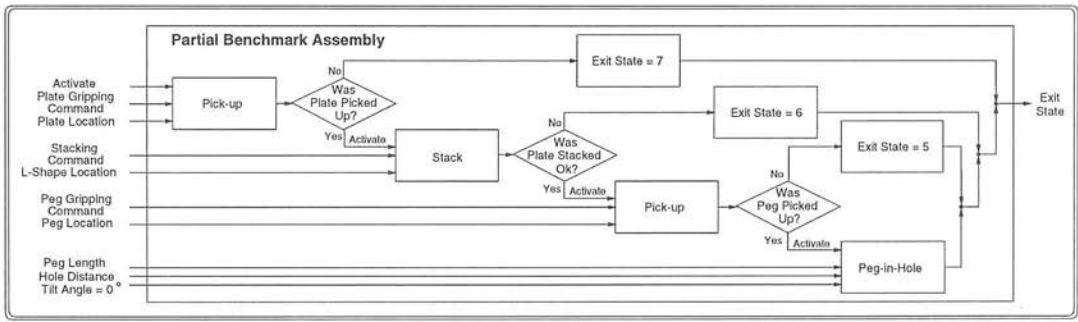


Figure 5.17: Partial Benchmark Assembly Diagram.

remark, we have to point out that the line, which we labelled *activate* and which was given as input to the four modules in the aforementioned diagram, has to be interpreted as the flow of the robot manipulator control.

The diagram above enhances the three kinds of components in this assembly task: pick-up, stack, and peg-in-hole. We have already extensively discussed peg-in-hole in subsection 5.2.1 on page 116, thus we concentrate now on the other two of them.

5.2.2.1 Pick-Up Module

As outlined above, this module is responsible for collecting an object lying at a certain location. Stable grasping is the key to a successful picking up, and much research has been carried out for both simple parallel jaw-grippers and multifingered end-effectors ([Paulos 93]). In this respect, we have to point out that the former are less versatile than the latter, because they can only manage just about 40% of

the total number of possible manipulations which a human 5-fingered hand can do ([Lundstrom & Rooks 77]). Incidentally, we have to observe that a three-fingered gripper would reach at least 90% of them. In this regard, as discussed earlier (cf. subsubsection 3.2.1.1 on page 71), we are employing a parallel jaw-gripper as our end-effector. Thus, what we can safely and stably grip with it consists basically in prismatic objects with parallel faces. However, we can still use it to grasp cylinders and spheres as well, assuming, of course, to perform the grip at their diameters¹⁹.

Several studies have analyzed and modeled object contacts with hard fingers. An interesting exception, since we are using a touch sensor wrapped around rubber padded fingers, is the analytical modeling of stability conditions during contact between an object and soft fingertips filled with powders or plastic fluids ([Akella & Cutkosky 89]). As regards grasp stability, we can distinguish two kinds of them: spatial and contact ones ([Montana 91]). The former may be regarded as the tendency of the grasped object to return to an equilibrium location in space, whereas the latter as the tendency of the points of contact to return to an equilibrium position on the object's surface. As regards contact itself, the problem of resolving its location and of determining forces and moments involved with it is addressed in [Bicchi *et al.* 93] which proposes a method for gathering contact information by measuring robot internal forces and torques.

There are specific positions which guarantee optimal gripping stability. In this respect, an algorithm for calculating non-graspable regions of a part is proposed in [vanBruggen *et al.* 93]. The method suggested in such an algorithm reduces the 3D grasp planning problem to a number of $2\frac{1}{2}$ D subproblems based on mechanical properties of industrial grippers. A different approach is proposed in [Caselli *et al.* 93] where a tool for selecting feasible grasps of a robotic hand under various situations is developed using a rule-based expert system with a neural net based classifier. Another solution for planning robust grasping operations is suggested in [Joukhandar *et al.* 94] which proposes the use of a technique employing physical models, labelled as physically-based, which allows to analyze the dynamic object/hand interactions. Interesting because it involves grasp planning for a two-fingered gripper with parallel faces, is the work

¹⁹ Cylinders can actually be grasped also by their parallel bases, if their height is within the range of the parallel jaw-gripper.

carried out in [Taylor *et al.* 94] which resolves the problem of grasping unmodeled 3D objects by using visual information. In this regard, it is also worth mentioning the method proposed in [Rodrigues *et al.* 95] which searches for suitable gripping points on the outline of generic shapes gathered by a vision system.

As realized above, the use of powerful sensing capabilities such as vision and force-torque sensors allows an agent manipulator to plan how to grip an object without requiring particular knowledge about it. In this respect, we have to point out that we have assumed to employ a very simple form of differential touch sensor which can only detect on/off contact events (cf. subsection 3.2.4 on page 80). Because of such an assumption, the behavioural module performing pick-up requires to have the work-cell location to which to drive the agent manipulator as an input parameter. Incidentally, we have to observe that some parts such as hollow cylinders may be actually grasped by closing the gripper jaw-fingers as well as opening them. Thus, in order to specify which kind of gripping has to be performed, we have to feed a second input parameter which encodes both the gap between fingers to be used for the grasping and the kind of gripping to be performed which can be by opening or closing fingers. The speed to be used to approach the vicinity of the part to be collected (gross motion) can also be encoded in this second input parameter. Summarizing, pick-up requires the following two parameters as input:

- a work-cell location, and
- a pickup command.

At this point, let us define the internal structure of such a module (cf. figure 5.18 on page 142). First of all we require a gross guarded motion in order to drive the agent manipulator above the target location where the object to be collected is. Then, if no obstacles have obstructed the way there, we need another guarded motion to approach the target location. If again no obstacles obstructs the operation, we have to activate a gripping module which would close or open the fingers according to the gripping command given as input to pick-up. If the clamping is successful, then we need to lift the gripped part.

As a remark, we have to observe from the analysis of the operations involved in a

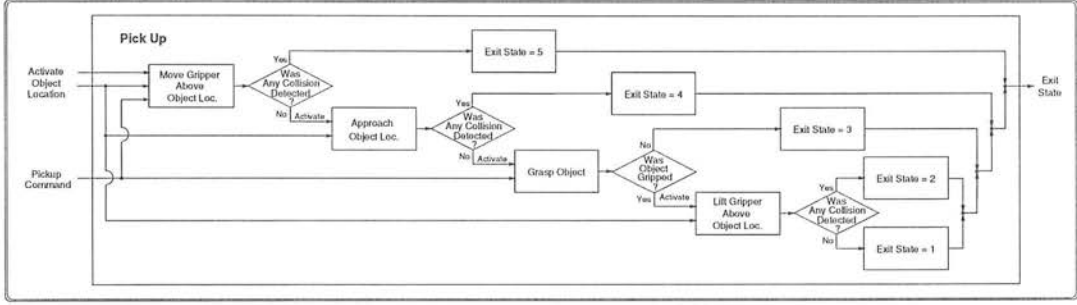


Figure 5.18: Pick Up Diagram.

part collection as outlined above that we basically require two guarded motions for positioning the gripper above the object and approaching it, a grasping module for the actual collection, and finally another guarded motion for departing from the current location.

Considering the structure of pick-up as outlined above, several outcomes are possible (cf. its diagram in figure 5.18 above). First, let us point out that if all the operations involved are carried out successfully, then we may view the result of the whole module as a successful part collection. All the other module outcomes come from possible failure along the sequence of operations involved in the collection. A close examination reveals four more outcomes besides the one corresponding to a successful pickup. Thus, we can synthesize the possible exit states of the pick-up module as follows:

- 1) part successfully collected,
- 2) obstacle obstructs part lifting,
- 3) part to collect not found,
- 4) obstacle obstructs part approaching,
- 5) obstacle encountered on the way to the location above part.

In order to test this module, we considered 3 prismatic parts (a plate, an L-shape part, and a peg with a prismatic head), and 2 cylindric ones²⁰ (a large prismatic ring, and hollow tube). We set a series of 10 collections for each part and recorded every time a successful pickup. In this regard, we have to point out that our module does

²⁰ These parts are taken from the largest kit of the torch assembly family.

not perform any grasp planning, thus it assumes²¹ that the gripping position given as input correspond to a stable grasping. As regards the cylinders, we have to stress that we tested both inner and outer pickup and recorded a success each time. However, we ought to stress again the point that outer grasping of cylinders is possible only if performed at their diameter.

5.2.2.2 Stack Module

A stack module may be regarded as a particular form of fine motion (cf. section 5.1 on page 102), which implies a part to be laid down at a specified work-cell location. In this regard, since we are not employing sophisticated sensors, we have to assume, as done with pick-up, to feed such a knowledge as an input parameter. Incidentally, we have to remark that a part may be released on top of another either by opening the gripper fingers or by closing them. Thus, stack like pick-up requires a second input parameter (stacking command) encoding the speed to be used to reach the vicinity of the stacking site and the way in which a part has to be released²², and hence stacked, on top of another once site has been reached.

As regards the internal structure of our stacking module, we have to analyze the operations involved (cf. figure 5.19 here below). First, let us observe that we need to perform

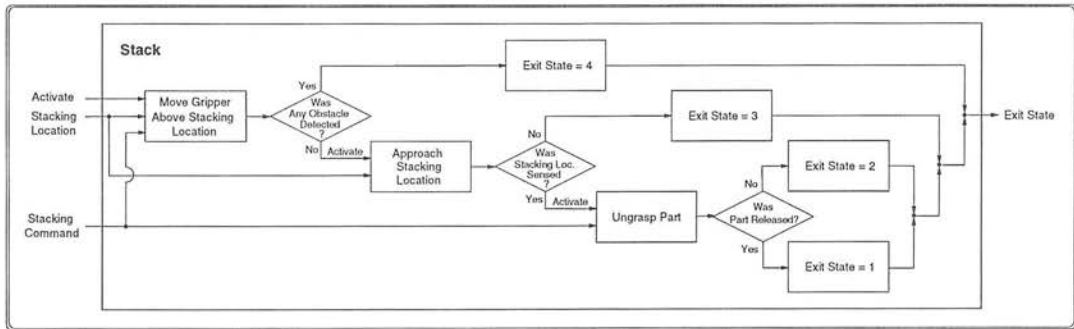


Figure 5.19: Stack Diagram.

a gross guarded motion to locate the gripper above the stacking location. Then, if no event contact has been detected, we need to perform a fine guarded motion in order to

²¹ This is in line with the assumption of having a hybrid planner which provides these information to the module.

²² Either by opening or closing fingers.

approach the stack. At this stage, if no obstacle has obstructed the approaching, we need to release the part held by the gripper according to the stacking command given as input. If the release is successful, the result of the entire module is a successful part stacking. Other possible outcomes which may derive from part stacking are all caused by possible failures of the operations involved in stacking. In this regard, looking back at the task description discussed above, we may synthesize the exit states of stack as follows:

- 1) part successfully stacked,
- 2) no part to be stacked,
- 3) obstacle obstructs stacking location approaching,
- 4) obstacle encountered on the way to the location above stack.

As a remark, we have to observe from the analysis of the part stacking operations discussed above that we basically require a guarded motion for driving the manipulator above the stacking site, another one for approaching the site itself, and finally an ungrasping module for releasing the part held by the gripper.

In order to test the stacking module as modeled above, we used the same 5 parts considered for pick-up: a plate, an L-shape part, a peg with a prismatic head, a large prismatic ring, and hollow tube. We set a series of 10 stacking at a specified location on the table for each part and recorded a success each time. By opportunely fixing obstacles during part stacking, we also tested the other outcomes.

5.2.2.3 Partial Benchmark Assembly Experiments

After having successfully developed modules implementing the task decomposition described in subsection 5.2.2 on page 136 and reported in figure 5.17, we have now to describe the experiments relating with the partial benchmark assembly. In this respect, we have to point out that our aim here is not only to show that two different kits can be assembled using the same program expressed in terms of behavioural modules, but also to find out which module components implementing the behavioural decomposition discussed in subsection 5.2.2 on page 136 are actually sufficiently versatile that we can regard them as general-purpose modules.

As done for peg-in-hole (cf. subsection 5.2.1 on page 116), we shall work with two different kinds of kits: one made of metal parts (aluminium kit) and one made of wooden ones (wooden kit). The two following paragraphs describes the experiments involving each of them.

Partial Metal Benchmark Assembly The first experiment we are discussing involves the metal kit. To start with, we stably fixed the L-shape part at a known location on the table. Then, after having recorded with a teach pendant both plate and peg home locations, we finally implemented the task as shown in subsection 5.2.2 on page 137 by taking advantage of the pick-up, stack, and peg-in-hole implementations developed earlier (cf. subsection 5.2.1, subsubsection 5.2.2.1, and subsubsection 5.2.2.2 on page 116, 139, and 143, respectively).

As described on page 139, the assembly of the partial benchmark requires 9 parameters as input. Two of them (plate and peg pickup commands) carried specific information on how to approach and grasp a part such as the speed to be used and the gripping gap between fingers. In this regard, we have to remark that providing gaps which are just enough to clamp these particular parts (metal plate and peg) would make the assembly program too bound to this specific benchmark kit. Thus, we decided to feed pickup commands which encode the maximum gap between fingers for both cases.

As regards plate and peg home location, we have to point out that they are very important in order to accomplish the assembly task successfully, and that an inaccurate placement of them may lead to a failure. Thus, it is crucial carefully arranging these parts at their home location each time a new experiment is restarted.

As regards the three peg-in-hole input parameters (peg length, hole distance, and tilt angle), we have to make a few remarks. First, we have to point out that our partial assembly is carried out on a table which is not tilted with respect to the X-Y plane of the robot frame system, thus the benchmark slant is assumed to be always 0° . Second, the peg length is assumed to be determined beforehand according to the technique outlined on page 137. Third, the hole distance is set to be 10 *cm* with respect to the stacking position of the plate. In this regard, we have to observe that this particular parameter is actually independent from specific part sizes.

We performed a series of 50 experiments and recorded 48 successful assemblies (96% success rate). The two failures were due to peg insertion failures (4% failure rate). The first one was brought about by an inaccurate positioning of the plate at its home location which caused a misalignment between plate and L-shape part greater than $\frac{1}{2}mm$, and the second one by peg wedging.

Partial Wooden Benchmark Assembly The second experiment we are describing here involves the wooden kit. The aim was to generalize the modules developed for the metal kit so that they can be applied to the wooden one, which, we stress, is similar to the metal one but not a scaled version of it (different sizes and different proportions despite similar overall shape).

As done in the previous experiment, we first fixed the wooden L-shape part at a specific work-cell location on the table and then recorded the home location of both plate and peg. In order to test the applicability of the program previously developed for the metal kit, we set the same pickup and stacking commands used there as inputs to this assembly.

As regards peg-in-hole parameters, we kept the same hole distance as before and set the assembly slant to 0° . The length of the peg shaft was then determined as done with the metal kit by following the technique outlined on page 137.

At this point, we performed the assembly of the wooden kit by running the assembly program developed for the metal benchmark instantiated with the new peg shaft length, new L-shape part, plate and peg home locations, and same other input parameters. As done before, we ran a series of 50 experiments and recorded 49 successful assemblies (98% success rate). The only failure observed was due to a peg-in-hole failure caused by peg wedging (2% failure rate).

As final remark, we have to point out that the assembly of the wooden kit was achieved by running the same unchanged program developed for the metal benchmark. This is possible because such a program is not only independent from specific part sizes, but also from part motions²³. However, because of the sensing capabilities we assume our

²³ Notice that this is achieved by not binding motions to specific parts: they are picked up and stacked always following the same path

agent to be equipped with, we still need to provide the robot with specific part feature locations. Moreover, the program still depends on the assumption that the part sizes are not so large that the robot bumps into them when making the initial movements.

5.2.3 Full Benchmark Assembly

The experimental assembly task discussed here in this subsection is practically the same one described in subsection 5.2.2 on page 136. The only difference consists in performing it on a fixture (jig) whose upper surface is slightly slanted with respect to the table²⁴. Thus, first we place an L-shape part on a jig, then we stack a plate on top of it, and finally we mate a peg with the tandem hole passing through both plate and L-shape part (cf. figure 5.20 here below). We have to remark that, since this assembly

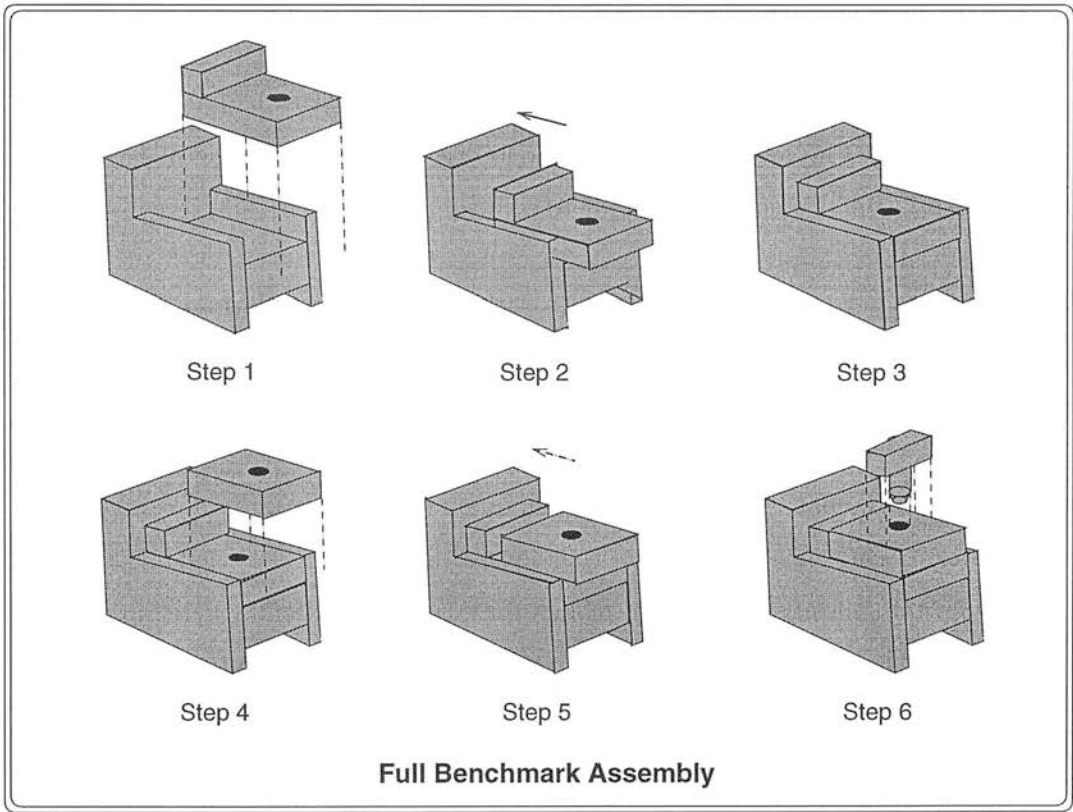


Figure 5.20: Full Benchmark Assembly.

is carried out on a slanted surface with respect to the table, both L-shape and plate in

²⁴ The table is assumed to be parallel to the X-Y plane of the robot frame system.

order to be placed on the jig, need to be tilted so that to make them parallel to the slope.

As experimental test beds, we consider also in this case two assembly kits: one made of metal and the other of wood. Both kits are composed of the same parts as the partial benchmark (an L-shape part, a plate, and a peg) but with the addition of suitable metal and wooden jigs (cf. figure 3.2 on page 74). In this regard, we have to observe that our fixture this time is the jig and not the L-shape part, thus, as done before for the partial benchmark assembly, we have to assume it firmly fixed at a specific location within the work-cell. Moreover, we have also to point out that we intend to use the same simple differential touch sensor employed to carry out the previous simplified version of this assembly.

The goal we aim to achieve by experimenting with this form of assembly is the same as in the case of the partial benchmark: to show that, by running an opportune generalization of the program developed for the partial benchmark, we can assemble both full benchmark kits.

We have to observe that since this assembly task involves a peg-in-hole operation, we have to cope with two specific kit dependent information: the peg shaft length and the tilt angle between table and jig slanted surface. The former is crucial in order to make the agent aware of when the peg is fully mated, whereas the latter is important for part stacking.

As discussed in subsection 5.2.2 on page 137, we can gather the knowledge of the peg shaft length by applying the technique of collecting the peg and making it touch the working table whose height with respect of the X-Y plane of the robot system is assumed to be determined in a similar way by a previous touch: the difference between the height at which the peg touches the table and the height of the table itself is the length of the peg shaft²⁵.

As regards the tilt angle, we can gather this knowledge by performing a similar touching technique where we assume the jig to be arranged so that the height of its slope

²⁵ Notice that we assume the peg to be gripped so that both fingertips and bottom face of the peg head lie on the same plane parallel to the X-Y plane with respect to the gripper (cf figure 5.16 on page 138).

decreases going along +X-axis. The technique is based on three equidistant close touches of one of the gripper fingertips with the upper face of the jig along +X-axis starting $\frac{1}{2}cm$ upward-sloping²⁶ with respect to the jig location (cf. figure 5.21 here below). First of all, we drive the manipulator gripper so that its touches the jig upper

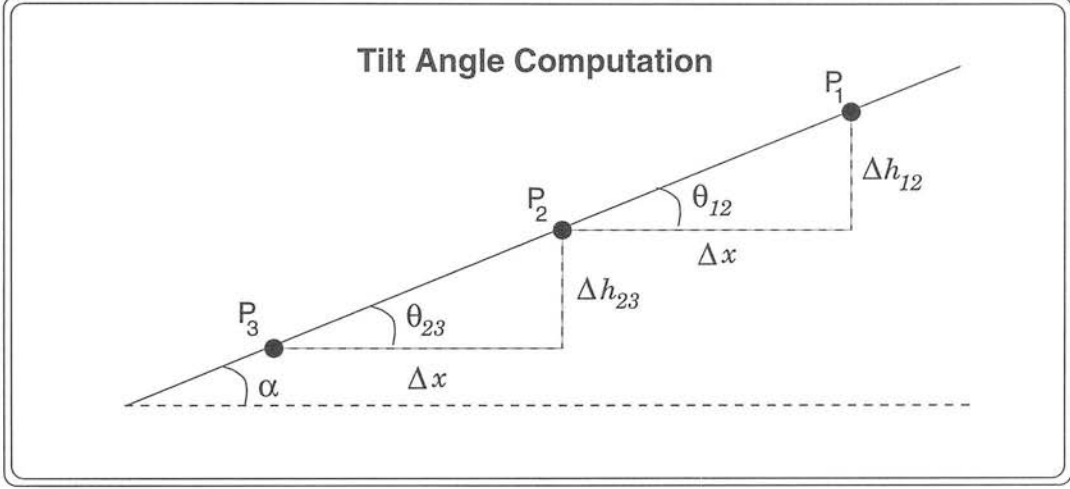


Figure 5.21: Procedure to Compute the Tilt Angle.

face at the first point (P_1). Once the contact takes place, we record the height with respect of the X-Y plane of the robot frame system (z_{P_1}) and then, by driving it along the X-axis, we make the fingertip touch again the slope at the second point (P_2). After having recorded the second height (z_{P_2}), we drive once again the fingertip as before to touch the third point (P_3) and record its height (z_{P_3}). By using these heights, we can compute the angle θ_{12} between slope $\overline{P_1P_2}$ and table as follows:

$$\theta_{12} = \arctan \left| \frac{z_{P_1} - z_{P_2}}{x_{P_1} - x_{P_2}} \right| = \arctan \left| \frac{\Delta h_{12}}{\Delta x} \right| \quad (5.1)$$

Similarly, we can calculate the angle θ_{23} between slope $\overline{P_2P_3}$ and table, and the angle θ_{13} between slope $\overline{P_1P_3}$ and table as follows:

$$\theta_{23} = \arctan \left| \frac{z_{P_2} - z_{P_3}}{x_{P_2} - x_{P_3}} \right| = \arctan \left| \frac{\Delta h_{23}}{\Delta x} \right| \quad (5.2)$$

$$\theta_{13} = \arctan \left| \frac{z_{P_1} - z_{P_3}}{x_{P_1} - x_{P_3}} \right| = \arctan \left| \frac{\Delta h_{13}}{2\Delta x} \right| \quad (5.3)$$

Thus, averaging on the angles θ_{12} , θ_{23} and θ_{13} , we can determine the tilt angle α of

²⁶ This information is derived from the jig location which is assumed to be known.

the slanted jig as follows:

$$\alpha = \frac{\theta_{12} + \theta_{23} + \theta_{13}}{3} \quad (5.4)$$

We have to remark that the third angle θ_{13} is introduced in order to reduce possible uncertainties due to the crudeness of the sensor used for measurement.

As already pointed out earlier (cf. section 4.1 on page 86), any assembly task within the behaviour-based assembly paradigm may be viewed as a single grand behavioural module whose execution leads to the accomplishment of the task. Thus, the first step is decomposing our full benchmark assembly task in these terms. In this respect, recalling the description given at the beginning, we can say that we need first of all two modules: one for collecting the L-shape part, and one for laying it down on top of the jig. If both of them do not detect any obstacle, we need to repeat them for the plate. If, once again, they are successful, the plate would at the end of the process lie down stacked on top of the L-shape. In which case we would need first another collecting module to pick up the peg from its home location, and then a peg-in-hole one to insert it inside the tandem hole passing through both plate and L-shape.

As pointed out for the partial benchmark assembly (cf. subsection 5.2.2 on page 138), if all the operations described above are successful, then the outcome of the entire assembly process is assembly successfully accomplished. All the other possible outcomes derive from failures of the operations during the process. In this regard, we have to recall that peg-in-hole in particular may terminate in four possible states: peg successfully mated, insertion aborted, obstacle detected during search for hole, and hole not found. The first one correspond to the outcome of successful assembly, whereas the others are two more outcomes of the full benchmark assembly task. As regards the failures of the other modules composing the assembly process, we may have L-shape part not found, L-shape stacking failed, plate not found, plate stacking failed, and peg not found). Thus, summarizing, we can say that the full benchmark assembly may terminate in one of 9 possible states: (cf. table 5.5 on page 151).

As regards inputs, we require two parameters to perform the first collecting module (an L-shape pickup command and the L-shape location), two to perform the first stacking (an L-shape stacking command and the jig location), other two more to perform the

Partial Benchmark	
Code	Exit States
1	peg inserted in hole and assembly successfully completed
2	peg insertion failed
3	Obstacle detected during search for hole
4	Hole not found
5	peg not found
6	plate stacking failed
7	plate not found
8	L-shape part stacking failed
9	L-shape part not found

Table 5.5: Full Benchmark Outcomes.

second part collection (plate), another one to perform the second stacking (a plate stacking command), two more to perform the third part collection (peg), and three to perform peg-in-hole(peg length, hole distance, and tilt angle).

Extending the diagram of the partial benchmark assembly (cf. figure 5.17 on page 139), we derive the more general behavioural decomposition illustrated here below in figure 5.22. Notice that such a representation omits the measurements of peg shaft and

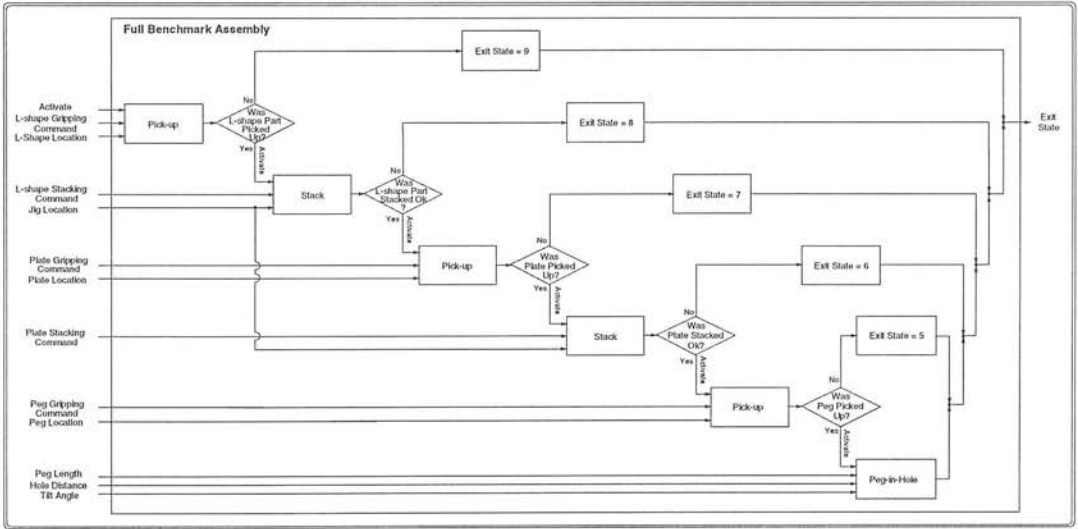


Figure 5.22: Full Benchmark Assembly Diagram.

jig tilt angle because strictly speaking they are not actually part of the assembly process. They are assumed to be computed following the two techniques discussed on page 137 and 148, respectively.

As last remark, we have to point out also in this case of full benchmark assembly that the line labelled *activate* and given as input to the six modules of the decomposition in the diagram above has to be interpreted as the flow of the robot manipulator control.

5.2.3.1 Full Benchmark Assembly Experiments

We have now reached the point of describing the experiments run to assemble the parts. As done with the partial benchmark assembly, our first concern was to set the work-cell and develop the assembly program implementing the decomposition shown here above in figure 5.22 which makes use of the modules developed earlier for the partial assembly.

The goal which we aimed to reach was not simply to show that a metal kit as well as a similar wooden one may be assembled by the same program, but was to show that a task such as the full benchmark assembly, which is close to the partial one examined earlier, may be accomplished by an opportune generalization of the same program developed for a similar task (partial assembly). Thus, we set also in this case two assembly experiments: the metal kit and the wooden one. The following two paragraphs are dedicated to describe the experiments involving each set of parts.

Full Metal Benchmark Assembly The first experiment we describe here involves the metal benchmark kit. As done with the partial benchmark, first, we firmly fixed the jig at a specific location within the work-cell and then, by using a teach pendant, we recorded the home locations of the various parts of the kit (L-shape, plate, and peg). Once the work-cell was set, we implemented the program as outlined in the diagram of figure 5.22 by using the same pick-up, stack, and peg-in-hole modules employed for the partial benchmark.

As pointed out for the partial benchmark, in order to instantiate such a program, the assembly of the full benchmark described on page 150 requires 12 parameters as input. Three of them (L-shape, plate, and peg pickup commands) carries again specific information on how to approach and grasp a part such as the speed to be used and the gripping gap between fingers. Thus, in order to be independent from any particular part size, we have also in this case to feed pickup commands encoding the maximum

gap between fingers. As regards the two stacking commands, they are not bound to a specific set of parts because they carry information merely about how to stack a part such as how a part has to be released on the stacking site.

As regards L-shape, plate, and peg home location, we have to remark that they are very important also in this case in order to achieve a successful assembly, because an inaccurate placement of them may cause a task failure. Thus, it is crucial to arrange carefully them at their home location every time a new experiment is restarted.

As regards the three peg-in-hole input parameters (peg length, hole distance, and tilt angle), we have to make a few remarks. Two of them (peg shaft length and jig tilt angle) require to be computed before the whole assembly can be started. We did this by following the two techniques discussed in subsection 5.2.2 on page 137 and subsection 5.2.3 and 148, respectively. The two measurements resulted in $24.1mm$ for the peg shaft (real length $23.5mm$) and 14° for the tilt angle (real slant 15°). The overestimation of the shaft length was due to an inaccurate gripping of the peg which had its head bottom face and fingertips plane not lying on the same plane. The third parameter, which corresponds to hole distance, is set also in this case to be 10 cm with respect to the stacking position of the plate. We have to point out in this regard that this particular parameter is independent from specific part sizes.

Instantiating the assembly program with the parameter settings presented above, we performed a series of 50 experiments and recorded 48 successful assemblies (96% success rate) and two failures (4% failure rate). The first of these failures was brought about by a slight misalignment between peg and insertion axes caused by 5 slips of the peg head between the gripper fingers during the search for hole stage, whereas the second one was due to peg wedging.

After having glued some emery paper to the outer skin of the fingers, we repeated the same series of experiments and this time we recorded 49 successful assemblies out of 50 (98% success rate). Once again, the only failure was due to peg wedging.

Full Wooden Benchmark Assembly The second experiment involved the wooden benchmark kit for which we performed the same preliminary work-cell setups as done

for the metal kit. Thus, after having fixed the jig and recorded its location together with L-shape, plate, and peg home locations, we used the same program developed for the metal kit opportunely instantiated with the parameters specific of the wooden kit. In this respect, in order to test the generality of the program, we fed the same pickup and stacking commands and the new home locations.

As regards peg-in-hole input parameters, we determined peg shaft length and jig tilt angle following, as done before, the same two techniques outlined on page 137 and 148, respectively. The two measurements resulted to be $31mm$ for the peg shaft (real length $30.8mm$) and 11.5° for the tilt angle (real slant 12°). As concerns the hole distance parameter, we kept it as before (10 cm).

At this point, instantiating the assembly program of the partial benchmark with the new parameter settings presented here above, we performed a series of 50 experiments recording 49 successful assemblies (98% success rate) and one failure (2% failure rate) due to a peg wedging.

It is worth observing that this series of experiments was performed with emery papers wrapped around the outer skin of the gripper fingers.

5.2.4 Benchmark Assembly Family Summary

This section was dedicated to discuss an artificial assembly task which we called the benchmark assembly. Such a task consisted of four parts (jig, L-shape, plate, and peg) which had to be mated in sequence: first, L-shape on top of jig, then plate on top of L-shape, and finally peg into the tandem hole passing through both L-shape and plate.

The goal which we aimed to achieve by experimenting with this test bed was two-fold: first, analyzing the task and decomposing it in terms of opportune behavioural modules, second, showing that our manipulator agent was capable of assembling two instantiations of the same problem (two benchmarks belonging to the same assembly family) simply by running two different instantiations of the same assembly program expressed in terms of the aforementioned modules.

Analyzing the assembly task, we noticed that it involved a peg-in-hole operation which is one of the most common manufacturing tasks (cf. subsection 2.3.1 on page 52). A

close examination of this particular operation revealed two basic components (cf. subsection 5.2.1 on page 116): search for a hole and peg insertion. As regards the former, since we were employing a simple differential touch sensor, we chose to implement it by means of a hopping spiral search (cf. subsection 5.2.1.1 on page 122). As regards the latter, we decided to implement it with a thrusting and correcting strategy which allowed the recovering of a misalignment between peg and insertion axes of up to $\frac{1}{2}mm$ along either X- or Y-axis (cf. section 5.2.1.2 on page 128).

Examining the benchmark assembly problem as a whole, we decided to tackle it first by considering a simplified version of it, which included only an L-shape part, a plate, and a peg and which we labelled as partial benchmark assembly (cf. subsection 5.2.2 on page 136), and then by extending such a solution so that to include a jig and hence perform the assembly on top of it. In this regard, we have to observe that our investigations were carried out by testing the assembly program solving the simplified instantiation of benchmark problem on two different members of the benchmark family (metal and wooden kits).

Analyzing the partial assembly, we noticed that it involves other two subtasks besides peg-in-hole: pick-up and stack. Considering the limitations of the manipulator gripper (two-fingered jaw gripper), we limited our investigations on such modules to simple polyhedrons with parallel faces and to cylinders, as long as these last are gripped by their diameters²⁷. Tests of the implementations of both pick-up and stack on the metal kit as well as the wooden one showed that they were very reliable and general-purpose as far as this restricted domain of parts is concerned.

The assembly program solving the partial benchmark problem required 9 parameters in order to be run. One of them in particular (peg shaft length) was very assembly kit dependent. However, we proposed a technique which allows a manipulator agent to gather such an item of knowledge by itself before the assembly starts (cf. page 137). We tested this program, instantiated with the same parameters except for the peg shaft, on both metal and wooden kit and recorded 96% and 98% success rate, respectively.

Extending the program developed before to solve the full benchmark assembly problem,

²⁷ Notice that our pick-up module as it was developed can grip hollow cylinders by opening fingers instead of closing them.

we observed that the only difference with the partial benchmark consisted in performing the various part matings on a jig whose upper face was slanted with respect to the X-Y plane of the robot frame system (table). Thus, in order to extend the program, we had simply to add two more modules: a pick-up one for collecting the L-shape part and a stack one for depositing it down on the jig (cf. subsection 5.2.3 on page 147).

The program solving the full benchmark problem required 12 input parameters, two of which (peg shaft length and jig tilt angle) were very benchmark kit dependent. In this regard, we used the same technique employed for the partial benchmark to determine the former, and we proposed a new one to determine the latter.

Instantiating the same assembly program with the same parameters, except for the peg shaft length and jig tilt angle, we performed both metal and wooden benchmark assemblies recording 98% success rate in both cases.

As last remark, we have to point out that the task decomposition of the benchmark assembly problem brought to light three useful modules, peg-in-hole, pick-up and stack, which are ergonomic and general enough within their scope of applicability (polyhedrons with parallel faces and cylinders) to be included in the set of elementary behavioural modules which we are looking for.

5.3 STRASS Assembly

The second assembly task considered in our quest for an ergonomic set of elementary general-purpose behavioural modules is what we labeled as STRASS which was kindly provided by the GEC-MARCONI Hirst Research Centre at Great Baddon. As discussed in subsection 3.2.3 on page 75, such an assembly allows us to study both retaining and snapfit operations. In this regard, we have to point out that these operations, despite being quite common as shown in another survey on manufacturing assembly tasks ([Byrne 89]), have not been the subject of much research as robotic applications. In this respect, we cite the work in [McLachlan *et al.* 92] which proposes the use of statistical techniques to overcome control and monitoring of sensor information during a snapfit assembly of a light bulb. Another work worth mentioning is [McManus *et al.* 92] which presents the structure of a systematic procedure that automatically generates

task information space regions for utilization in control and monitoring of assembly operations such as snapfits.

STRASS is a close relative to *insert-peg-with-retainer* which is still today quite a common manufacturing operation (cf. subsection 2.3.1 on page 47). However, we have to remark that STRASS although involving a retaining operation, differs slightly from the retaining assembly process examined for instance by Kondoleon because its retainer is not a separate part inserted at the end of the mating process but an actual integral component of one of the assembly parts which is kept in a retaining position by an internal spring. In this respect, STRASS may be viewed as a particular form of snapfit operation.

The rest of this section is dedicated to analyze and discuss both retaining and snapfit aspects of STRASS. Thus, we organize the section analyzing first which elementary operations are involved in order to define modules capable of performing retainings and snapfits (subsection 5.3.1 here below), then we present an implementation for resolving the particular retaining operation required by STRASS and an implementation for resolving snapfits (subsection 5.3.2 on page 160), and, finally, we discuss the description of the experiments carried out in order to test the proposed solutions for retaining and snapfit (subsection 5.3.3 on page 163). The last part of the section will summarize the results achieved by our investigations on the STRASS assembly (subsection 5.3.4 on page 164).

5.3.1 STRASS Analysis

STRASS is basically made of two parts (cf. figure 3.4 on page 76 and figure 5.23 on page 158), which we labelled with STRASS A and STRASS B respectively, and consists merely in mating them together. This task is very trivial to describe to any human operator who would accomplish it straight away without any particular problem. Unfortunately, a robot cannot rely on a complex and effective sensory system (our eyes and finger skin) with a powerful, compliant and accurate general-purpose tool (our hand). However, the main difficulty here is not just the lack of sensing capabilities, nor is it the decomposition of the task in elementary modules. The problem which really makes the task difficult to be carried out by our manipulator agent is the very

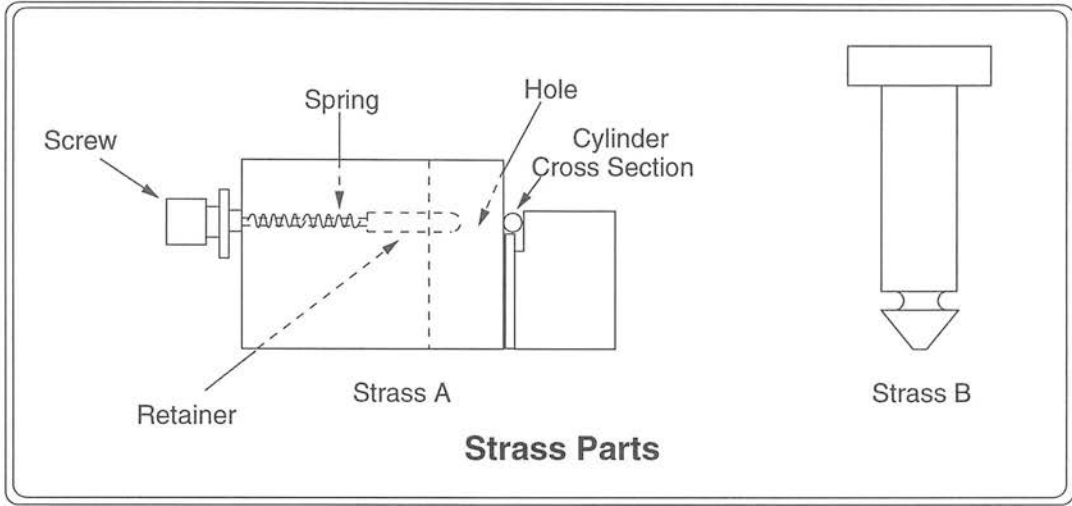


Figure 5.23: Two-dimensional Diagram of the STRASS Parts.

little compliance between end-effector and part and the lack of a sensible strategy which allows it to apply the right strength in order to bypass a temporary opposing force.

As pointed out earlier, STRASS viewed as a retaining operation shows many similarities to peg-in-hole, even so, there is at the same time a crucial difference between the two of them because STRASS assembly requires to exert a push to the peg (STRASS B) in order to bypass the retainer and insert the peg inside the hole (STRASS A). Thus, the simple strategy of driving a peg to a hole rim and then, by exploiting gravity and mechanics, letting it fall in would not be good²⁸, nor would pushing directly STRASS B inside STRASS A²⁹. However, we cannot employ peg-in-hole as developed for the benchmark assembly (cf. subsection 5.2.1 on page 116), because the main component of its insertion module is based on a guarded motion (cf. section 5.1 on page 102) which relies solely on the information gathered by a crude differential touch sensor (cf. page 80) capable of merely detecting on/off event contacts with other objects. In this regard, we have to remark that basing STRASS mating on such a guarded motion yields the problem of having a sensor triggering an inhibition signal to the manipulator controller every time a contact between peg (STRASS B) and retainer occurs, which

²⁸ We assume that the weight of the peg is such that the force opposed by the retainer is greater than the one generated by gravity

²⁹ We suppose in this case to know already the home location of STRASS A.

in turn prevents our manipulator agent from moving further. The retainer under this point of view plays the role of an obstacle obstructing a smooth insertion inside the hole. Therefore, the main problem we have to face with a straight insertion strategy is that our guarded motion, which we want our retaining insertion to be based on, was designed and developed so that every contact event always triggers an inhibitory signal to stop the manipulator agent each time a mating is attempted. Considering this problem and our intention to employ the same guarded motion used for the benchmark assembly, we need to develop a special strategy performing the insertion under the opposing force of the spring retainer.

Observing the chamfered shape of the STRASS B tip and the cylinder at the top of the STRASS A hole in front of the retainer (cf. figure 5.23 on page 158), we notice that there is a better and smoother way to mate the two parts and, at the same time, to make use of the same guarded motion developed for the benchmark assembly. To this end we need to assume that manipulator motions, rotations included, can be viewed as guarded motions. Bearing this in mind, if we tilt the peg and bring it to match the aforementioned cylinder at the hole rim, then we can achieve the STRASS mating with a simple rotation³⁰ about the axis of such a cylinder without that the opposing force of the retainer, by triggering the touch sensor, prevents STRASS B from being inserted (cf. figure 5.24 on page 160). This is possible because of the low coefficient of friction of the material (aluminium) which both parts are made of. The retainer, sliding along the chamfered surface of the peg, actually moves back so that the peg smoothly passes by without triggering the sensor. When STRASS B reaches the orthogonal position with respect to STRASS A, the retainer, closing the gap between itself and the rounded part of the tip of the peg, causes a sudden hit which gets the sensor to trigger an inhibitory signal and, hence, to detects the termination condition of the task.

The strategy outlined above resolves this particular STRASS assembly problem, but such a solution cannot of course be applied to all forms of retaining or snapfit operation. However, given the restricted domain of assembly parts which we assumed to work with, we considered it good enough for our purposes.

At this point we have to remark, as done earlier, that STRASS may also be regarded as

³⁰ Rotation about a remote centre.

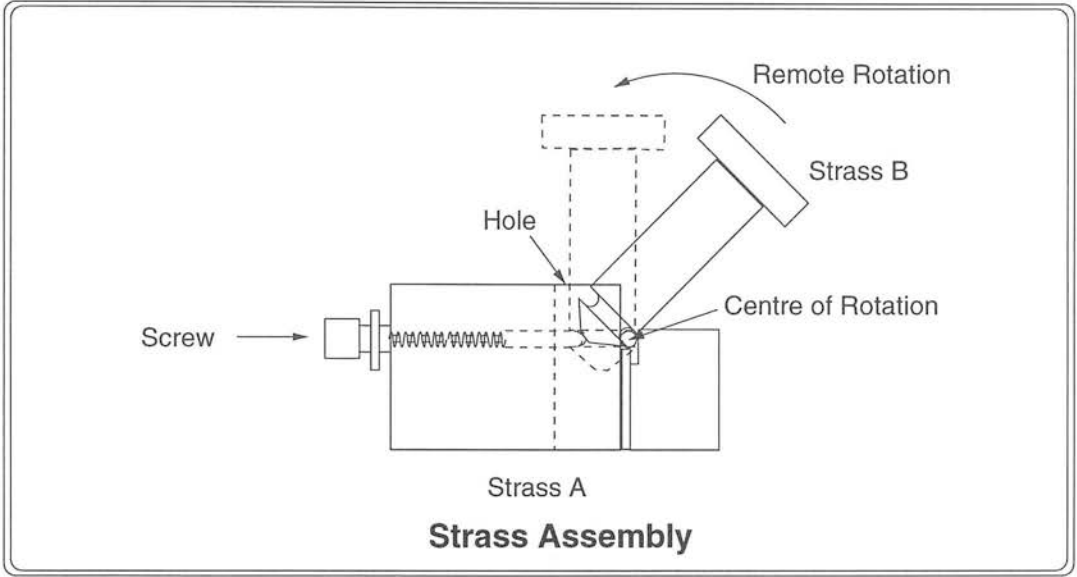


Figure 5.24: Two-dimensional Diagram of the Insertion with the Retainer.

a particular form of snapfit. In this respect, the most trivial strategy for accomplishing it is by directly thrusting the peg (STRASS B) inside the hole. However, if we base it on the same simple guarded motion module used for the benchmark assembly, we would incur the same sensor triggering problem mentioned above for the retaining insertion. Thus, if we want to develop a module resolving a more general form of snapfits without employing any force-torque sensor, the only choice we have is basing such a module on the more usual unguarded motion.

5.3.2 STRASS Retaining and Snapfit Implementations

A close examination of the strategy exploiting the particularities of STRASS outlined in the previous subsection shows that, in order to be performed, it requires first to tilt STRASS B (peg) with respect to the orthogonal axis to the X-Y plane of the robot frame system, then to match it with the cylinder located at the rim of the hole, and finally to rotate it about the cylinder axis (remote centre) up to a normal position to the aforementioned X-Y plane. Observing that these three operations are actually different instantiations of our guarded motion, we can decompose the task, which we label as insert-peg-with-spring-retainer, in three guarded moves as illustrated in figure 5.25 on page 161.

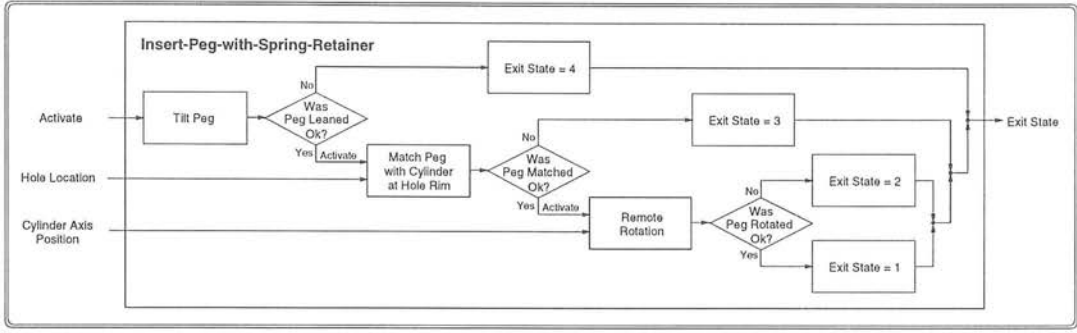


Figure 5.25: Diagram of Insertion with Spring Retainer.

Examining such a decomposition, we notice that, in order to carry out the task, we require to feed two parameters as input: the hole location and the position of the cylinder axis at the rim of the hole.

As regards the possible outcomes of the task, we observe that if all the operations involved are successfully performed, then the entire module insertion-with-spring-retainer will result in STRASS successfully mated. The remaining outcomes may come only from possible failures during any of the three operations. In this respect, it may happen that a peg cannot be rotated because of an obstacle in the hole, or that a peg is not matched because of unpredicted reasons with the cylinder at the rim of the hole, or finally that a peg cannot be leaned because of environment constraints. Thus, synthesizing, we can have the following four exit states:

- insertion with spring retainer accomplished,
- obstacle prevented insertion by rotation about the cylinder axis,
- peg not matched with cylinders at hole rim,
- environment constraints prevented peg tilting.

Looking back at the outcomes above, we have to remark that the three operations involved in the mating task, although being three different instantiations of our guarded motion and therefore having just two exit states (motion successfully accomplished, or obstacle detected), are in this case considered successful only if the touch sensor triggers (cf. section 5.1 on page 102). During the matching between peg and cylinder at the

rim of the hole, the event contact notifies that a peg has reached the position to start the remote rotation, whereas in the remote rotation itself the sensor triggering notifies a successful peg insertion.

It is worth pointing out that the two exit states of the guarded motion mentioned above, although being exactly the same ones, are in this case used with a different meaning. In this regard, the state of *obstacle detected*, which formerly expressed a failure, is this time employed as a goal state expressing the success of the operation, whereas the state of *motion accomplished*, formerly the goal state, is now considered as a failure because the rotation cannot be performed.

At this point examining the more general strategy for resolving snapfit outlined at the end of the previous subsection, we notice that it requires two operations: a guarded motion for putting the peg held by the manipulator agent in contact with the hole rim, and an unguarded move to thrust the peg in (cf. figure 5.26 here below).

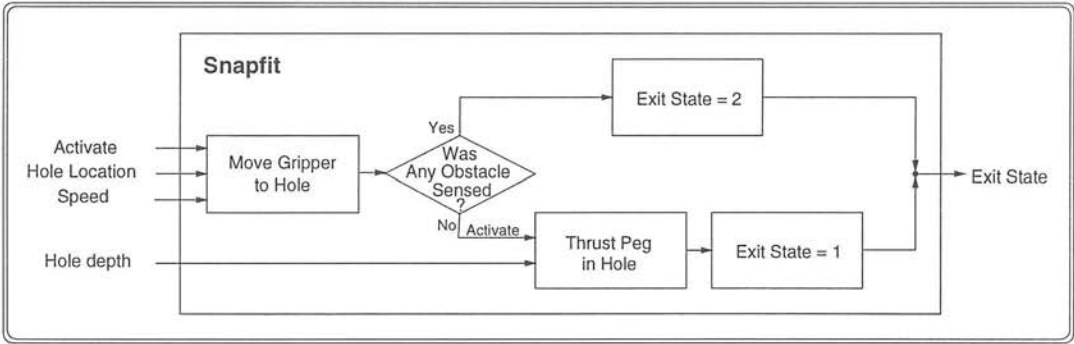


Figure 5.26: Diagram of Snapfit.

The information required to perform such a strategy, which we label as snapfit, are basically three parameters: the hole location, the speed to reach it, and hole depth. The last one is necessary as the only means available to an unguarded motion of stopping the thrust.

As regards the possible outcomes of the module, we observe that just two situations may arise: either peg successfully inserted or environment obstacle detected on the way to the hole.

5.3.3 STRASS Assembly Experiments

The two modules presented and analyzed in the previous subsection (insert-peg-with-spring-retainer and snapfit) were tested on the STRASS assembly. In this regard, we have to point out that the push required to overcome the force opposed by the retainer can be manually controlled by tightening or loosening a screw on the back of STRASS A (cf. figure 5.23 on page 158): tighter the screw, higher the push required for the insertion. The screw is completely tightened after four revolutions about its axis.

We started our experiments with insert-peg-with-spring-retainer. We first fixed STRASS A at a known location and then recorded the position of the hole by means of a teach pendant. After having set the screw to $\frac{1}{4}$ of its thread, we performed a series of 50 assembly experiments recording 50 successful STRASS matings (100% success rate). Tightening the screw up to $\frac{1}{2}$ of its thread and repeating another series of 50 assembly tests, we recorded again 50 successful matings (100% success rate). A further tightening of the screw up to $\frac{3}{4}$ of its thread showed once again similar results (50 successes out of 50 experimental tests). After having completely tightened the screw to STRASS A, we recorded only 48 matings out of 50 experiments (96% success rate). The explanation accounting for such a different experimental outcome derives from the increased force opposed by the retainer which vectorially summed with the friction between peg and retainer is just about to overcome the force which makes the retainer slide backwards. Thus, when dusts increases friction, the total opposing force prevents the retainer from sliding back, and therefore makes the sensor trigger and stop the manipulator.

At this point we considered the second module we mentioned at the beginning: snapfit. We kept STRASS A in the same location and used the position of the hole recorded before. After having reset the screw to $\frac{1}{4}$ of its thread, we performed another series of 50 experiments recording again 50 successful matings (100% success rate). Increasing the screw tightness up to $\frac{1}{2}$ of its thread and repeating a new series of 50 tests, we recorded again 50 successful matings (100% success rate). Tightening once more the screw to $\frac{3}{4}$ of its thread, we recorded a similar success rate (50 successes out of 50 experimental tests). The final series of 50 tests with the screw completely tighten to STRASS A showed again 50 successful matings (100% success rate). The different performances of

snapfit emerged here compared with those relative to insert-peg-with-spring-retainer is due to the different kind of motion on which their insertions are based. The absence of sensing during the insertion makes snapfit more robust to spurious sensor triggerings.

5.3.4 STRASS Assembly Summary

At this point let us summarize the results achieved by our investigations on the STRASS assembly. As mentioned at the beginning of the section, STRASS is very simple to describe, since it merely consists in mating its two part components (STRASS A and STRASS B), but not so easy to be reliably carried out by a robot equipped solely with a differential touch sensor such as ours. The main difficulty lies in the lack of a sensible strategy which allows the manipulator agent to apply the right strength in order to bypass the temporary opposing force of the retainer.

STRASS, as pointed out in the introduction to this section on page 157, may be viewed as a retaining assembly task as well as a snapfit. In this regard, we pointed out that we cannot assemble STRASS by resorting to the peg-in-hole module developed for the benchmark assembly because of the presence in the hole of a retainer which acts as an obstacle obstructing the insertion³¹. Each time an insertion is attempted and the contact with the retainer detected, the sensor would simply trigger an inhibitory signal to the manipulator agent which would as a consequence prevent the robot from moving further. In order to resolve the retaining operation involved by STRASS, we proposed a solution exploiting the mechanics of its particular parts and the smoothness of the metal surfaces in contact (cf. subsection 5.3.1 on page 159). Experimental tests performed by varying the force opposed by the retainer³² showed that the module implementing the strategy of insert-peg-with-spring-retainer has a considerable degree of reliability up to an opposing force generated by tightening the screw up to $\frac{3}{4}$ of its thread (cf table 5.6 on page 165). As regards STRASS as a snapfit task, we proposed a very simple strategy consisting in a direct thrust of the peg (STRASS B) into the hole (STRASS A). Experimental tests of the module implementing such a strategy (snapfit) showed a consistent degree of reliability and robustness within the limits of the robot

³¹ This is due to the fact that peg-in-hole is based on our form of guarded motion which makes use of a very crude touch sensor capable of detecting on/off contact events only.

³² This is achieved by tuning a screw on the back of STRASS A (cf. figure 5.23 on page 158).

Screw Thread Tightened	insert-peg-with-spring-retainer	Success rate	snapfit	Success rate
1/4	50/50	100%	50/50	100%
1/2	50/50	100%	50/50	100%
3/4	50/50	100%	50/50	100%
4/4	48/50	96%	50/50	100%

Table 5.6: STRASS Retaining and Snapfit Experimental Data.

strength (cf. table 5.6 above). In this respect, we have to remark that the different experimental performance between the two strategies are due to the different guarded motions which they are based on.

The investigations on the STRASS assembly allowed us to study two close clinching operations (retaining and snapfit). We proposed a solution resolving the particular case of STRASS by resorting to our simple guarded motion and developed a behavioural module which performs it reliably and robustly within certain limits (insert-peg-with-spring-retainer). As pointed out earlier, such a solution is applicable only to our restricted domain of rigid parts. A different solution based on a common unguarded move gave a possible different answer and the module implementing it (snapfit) showed consistent robustness and reliability in several experimental tests.

Considering that both retaining and snapfit are quite common manufacturing tasks, the implementations which we developed, although being valid within a specific set of parts, show the interesting characteristic of being simple and ergonomic within such a domain. Thus, although a more general applicability of them would require further development of the modules, this points to the need of including them to the set of elementary behavioural modules which we are building.

5.4 Torch Assembly Family

The assembly tasks discussed so far have always been very simple and in many ways very artificial. What we are going to consider here in this section concerns instead real industrial products: the assembly of electric torches (cf. figure 3.5 on page 77). The aim which we want to achieve by analyzing this test bed is not just to show that the different parts of one torch kit can be assembled together, but to prove that, given

a family of similar assembly kits, it is possible to program a robot to assemble all of them. As explained in subsection 3.2.3 on page 73, we chose a family of electric torches because it is not only a real industrial product, whereon actually testing the usefulness and descriptiveness of the modules developed so far, but it also includes an extremely common assembly parts joining operation: *screw* (cf. subsection 2.3.1 on page 47).

Considering one kit of the torch family, we notice that it is made of six parts: a tube, a reflector with a bulb³³, a plastic glass, a threaded large ring to retain the reflector and the glass, and two batteries which we label as A and B (cf. figure 5.27 here below). In

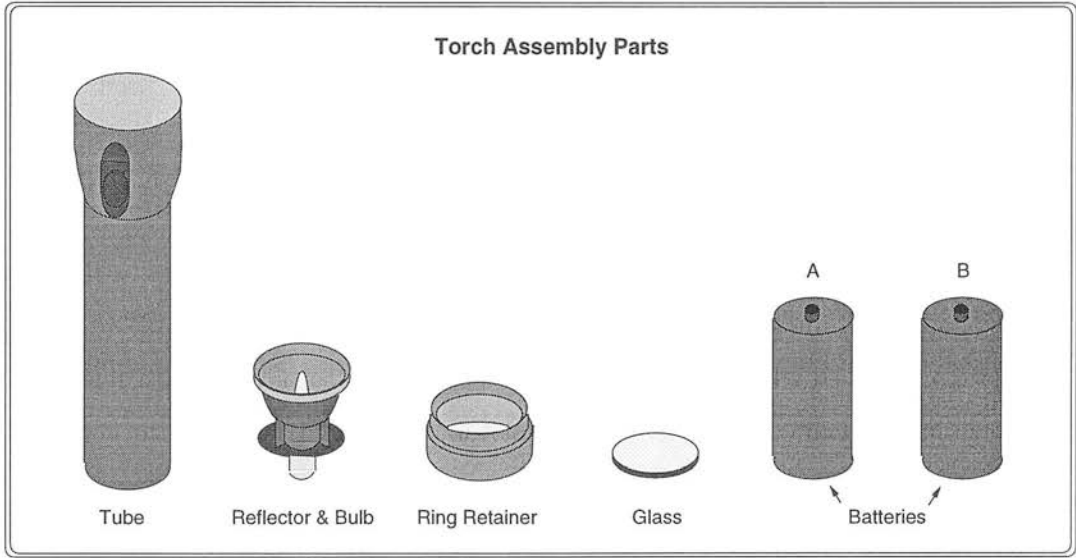


Figure 5.27: Torch assembly Kit.

order to assemble such a kit, any human operator would have to follow a sequence of steps:

1. inserting the two batteries in turn inside the tube,
2. stacking the reflector on top of the subassembly,
3. stacking the glass on top of the previous parts, and finally
4. screwing the retainer to the tube.

Such a description is sufficient for accomplishing the assembly of not simply one par-

³³ This is actually a subassembly made of the reflector itself, an appropriate bulb and a screw to fix the bulb to the reflector.

ticular torch kit but an entire family of them (cf. figure 5.28 here below for its graphic representation). If we want to keep this task decomposition of our torch assembly, we

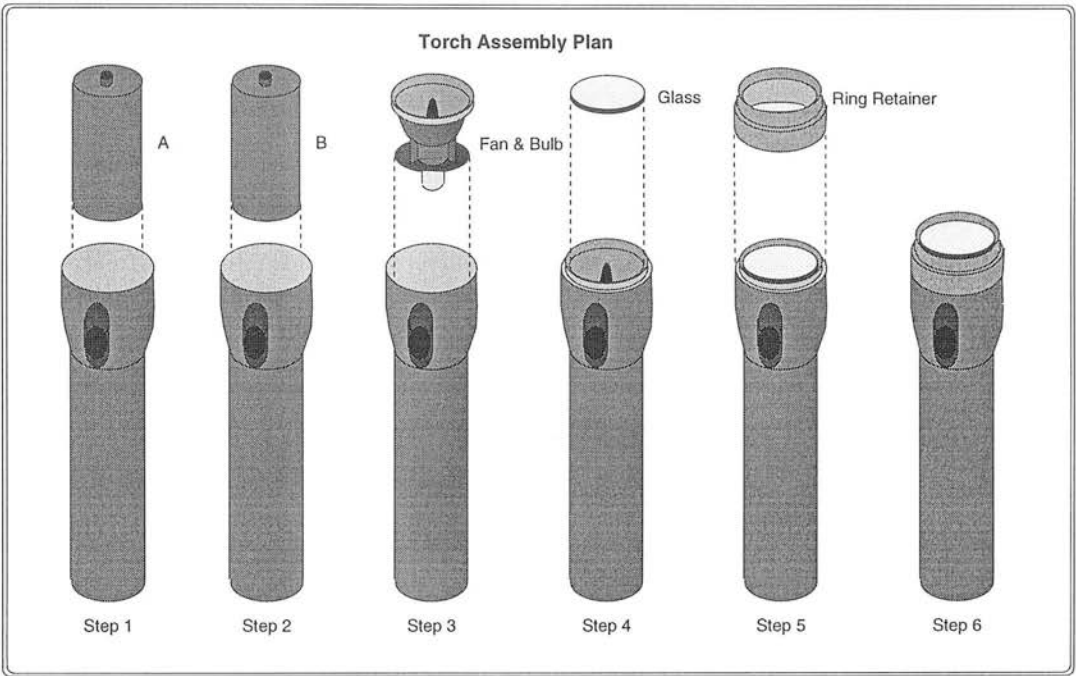


Figure 5.28: Diagram for describing a Torch Assembly.

need to analyze each step of the description given above. To start with let us point out that the first operation, which involves battery insertion, as well as the following two ones may actually be regarded as a stacking task, which, as mentioned in subsection 5.2.2.2 on page 143, can be viewed basically as sequences of guarded motions (cf. figure 5.19 on page 143). As regards the last step, we have to observe that it involves a screw fastening task which requires a more extended discussion.

The rest of this section is dedicated to discuss first screw fastening operations (cf. subsection 5.4.1 on page 168), then to present a reliable strategy to perform a general assembly of our torch family (cf. subsection 5.4.2 on page 173), and finally to describe the experiments performed in order to test such a strategy (cf. subsection 5.4.3 on page 182).

5.4.1 screw Module

As discussed in subsection 2.3.1 on page 47, screw fastening operations are by a large factor one of the most common parts joining processes in manufacturing industry. Because of high occurrence, special screwdriver tools have been developed to perform it rapidly and reliably. In this respect, we have to point out that our aim here in this subsection is not to develop a theoretical substitute for these tools, nor a mathematical analysis of the task. Indeed, what we want to achieve by investigating the different facets of such a task within the behaviour-based assembly approach is simply to propose a practical strategy allowing a reliable fastening of a large nut to a screw thread, in this case complicated by forming of the mating threads from pressed steel sheet, which makes it quite tricky for people to accomplish.

There is not much relevant literature for screw fastening performed without specialized end-effectors. In this respect, it is still today a research topic largely unexplored in robotics and the few works done in the area have just tackled simple aspects of it. In this regard, the work pursued in [Tao *et al.* 90] implemented a bolt threading operation with generalized stiffness controlled manipulator emulating a remote center compliance (RCC) type device. Incidentally, it is worth pointing out that a stiffness solution for threaded fasteners was initially suggested without however any experimental verification in [Loncaric 87]. A method for constructing a damping matrix for an insertion problem applicable in certain canonical configurations was then presented in [Schimmels & Pushkin 90]. A dynamic simulation of threaded insertion based on Euler's equations, impulsive forces and geometric description for threaded parts was developed in [Nicolson & Fearing 91].

As seen above, not much has been done to resolve screwing and a general and robust control solution has not as yet been presented. In this regard, we describe in the rest of this subsection a practical technique (section 5.4.1.1 here below) which shows an interesting practical reliability confirmed in experimental tests (section 5.4.1.2 on page 172) whose results will be summarized at the end of this subsection (section 5.4.1.3 on page 172).

5.4.1.1 Strategy for Screwing

Examining carefully the way in which a human operator performs an awkward nut screwing on to a bolt, we notice that he tends to make three actions: first he aligns nut and screw thread axes with each other, then, exploiting gravity and his hands' compliance, he mates the parts so that he senses the starting of the screw thread by means of his hands, and finally he performs partial twisting rotations until he sees the end of the thread or senses that the nut cannot be further tightened.

This pattern of actions outlined above works fine for a human operator because he can rely on the his hands' dexterity, however, general-purpose robot end-effectors, although being accurate tools, are more limited and less versatile than human hands.

The manipulator agent with which we want to perform screw fastening consists of a robot equipped with two grippers (right and left hands) and a differential touch sensor wrapped as a skin around the two finger of each gripper (cf. subsection 3.2.4 on page 78). Given such a hardware, we need to find a more suitable pattern of actions on which to model a behavioural module for accomplishing it. To this end let us suppose to perform nut screw fastening on a bolt by using a pair of pliers to hold a nut and another pair to hold a bolt. The first step we perform is mating the two parts together so that they are coaxial and in contact with each other. At this point we actually screw the nut on the bolt but, since we are assuming not to use any automatic screwing tool, we have to do it by twisting the nut, releasing it, rotating back and repeating the process until the end of the thread is reached. However, before doing so, it is safer, in order to avoid accidental misalignments, to make sure that the screw thread is started by performing an entire simultaneous rotation of the two hands twisting the two parts in opposite directions about the common axis. Once the thread is started, we can actually proceed to release the nut and rotate back to the position where the nut was initially laid on top of the screw. At that point, we repeat this twisting process over and over again until the force to tighten the nut exceeds a certain threshold and makes the sensor trigger. The sensor we assumed to employ (cf. page 80) is not capable of measuring the size of a force but it can still very reliably sense the presence of a force opposing the twisting rotation, which after all is what we need to detect.

The second pattern of actions outlined above for accomplishing screw fastening is much more suitable to our agent than the first one. Thus, assuming to adopt it as our screwing strategy, we need to define clearly which operations it involves.

Analyzing in more details the aforementioned strategy, we notice that we have first to start the screw thread and then performing repetitive twisting until an opposing force makes our touch sensor trigger an inhibitory signal to our manipulator agent. In order to initiate a screw thread, we observe that a complete revolution of the nut to be fastened about the screw thread axis guarantees that the thread is started. However, in order to be completely sure of that, whatever the initial position of nut and threaded bolt with respect to each other, we have to perform two revolutions. In this regard, recalling that we are employing a two-gripped manipulator agent, we basically perform simply one rotation clockwise with one gripper and at the same time a rotation anti-clockwise with a second one. These two rotations, which are performed in parallel, are monitored by a touch sensor that remain active until they complete their rotation, or it triggers. The end of this phase may have two outcomes: either the rotations have been completed and the thread is started, or the sensor has triggered a signal before the end of the rotations. If the latter is the case, then the whole fastening process has to quit with an exit state coding the failure. If the former is instead the case, then we have to start a sequence of repetitive twistings. This process consists in first ungripping the part currently held by the right gripper, second rotating it back of one revolution to the initial position, third gripping the part now partially screwed, and finally rotating the nut one revolution forward. At the end of this stage, if none of the steps along the process has failed and the sensor has not triggered, the whole twisting process is repeated. Summarizing, we can synthesize the algorithm discussed above in the following schema:

- simultaneous opposite rotations of left and right grippers for starting the thread,
- repeat
 - release of the nut held by the right gripper,
 - rotation of right gripper to its initial position,
 - nut gripping,

- nut rotation
- until a force opposing the rotation triggers a signal.

We have to remark that all the rotations involving the manipulator right gripper have to be regarded as instantiations of the same guarded motion used for the benchmark assembly. As regards gripping and releasing operations³⁴, they too have to be implemented as guarded operation in a way similar to our `guardedmove`.

Notice that the algorithm above does not require any input parameter in order to be performed. As regards the possible outcomes of the task, we observe that they are essentially three:

- 1) nut successfully fastened,
- 2) part to be fastened not found,
- 3) screw thread not started.

A complete diagram summarizing the description of the screw algorithm outlined in this subsection is reported below in figure 5.29. Notice that we have not included in it the initial mating of nut and screw thread because it is not strictly speaking part of the fastening task. Such a mating can always be performed as an opportune instantiation of `guardedmove`.

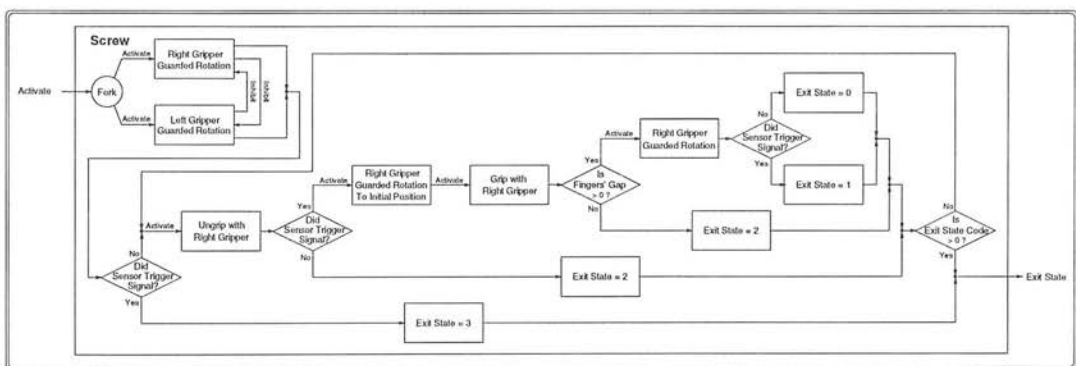


Figure 5.29: Screw Diagram.

³⁴ Releasing and ungripping are intended here as synonyms.

5.4.1.2 Screwing Experiments

Following the algorithm discussed above, we developed one of the possible implementation of screw fastening, which we labeled as screw, employing the same modules developed for the benchmark assembly family (cf. section 5.2 on page 115). In order to test such a module, we considered the largest kit in the torch family pictured in figure 3.5 on page 77. After having clamped the torch tube with the left gripper³⁵, we gripped the matching ring retainer³⁶ with the right one. At this point, we ran a series of 50 experimental tests consisting in driving the parts involved in contact and then performing screw. After the 50 trials, we recorded 48 successful fastenings (96% success rate). The two failures were due to spurious sensor readings caused by misalignments of the ring retainer with the screw thread which, under the push of the right gripper, slipped out of axis through the left gripper small fingers.

After having substituted for these small fingers customized V-shaped ones with larger gripping surface, we repeated the same series of 50 tests and this time recorded no more failures.

5.4.1.3 Summary of Screwing

As pointed out at the beginning of this subsection, tool-less screw fastening is still today quite an unexplored research topic as a robotic application. In this regard, we presented an interesting strategy for screwing modeled on the human nut twisting operation (cf. subsection 5.4.1.1 on page 169) which employed a very simple form of differential touch sensing. The algorithm implementing such a strategy did not require any input parameter (cf. page 171). As regards the possible outcomes of the screwing task, we observed that they are essentially three: nut successfully fastened, nut to be fastened not found, and screw thread not started.

Experimental tests using normal rectangular shaped fingers showed 48 successful fastenings out of 50 trials (96% success rate). The two failures recorded were due to the little gripping area actually clamped by the rectangular fingers. The employment of

³⁵ The left gripper is here used as a flexible fixture.

³⁶ Since we tested screw with a torch tube and its ring retainer, we considered this last as the nut mentioned in the algorithm.

customized V-shaped larger fingers in a new series of 50 experimental tests showed 50 successful fastenings (100% success rate)

5.4.2 Torch Assembly Plan

The plan outlined in section 5.4 on page 166 (cf. figure 5.28 on page 167) is a very good description of the torch assembly easily understandable by any human operator. The fact that at the bottom of the tube there is a spring which pushes up the batteries, and as a consequence all the other parts stacked on top of them, does not affect the description power of the above mentioned diagram: any human thanks to his powerful compliant hands can still overcome the difficulty of mating the different parts under the pressure of the spring. Unfortunately, we can not say the same for a manipulator agent. Once a part is stacked and released by the right gripper on the torch tube, the pressure of the spring makes the part slightly move from that position. This introduces extra uncertainty which a robot, relying only on a simple differential touch sensor, can not cope with. A better way to get around the problem of having this extra uncertainty is by dividing the assembly task in two independent subassemblies, which can be reliably built in separate stages, so that they do not introduce further uncertainty when they are joined together. In this regard, we noticed by examining the different parts of a torch kit that there are two sequences of operations in their assembly which are completely independent from each other:

- the insertion of the two batteries inside the torch tube, and
- the mating of the reflector with its matching glass and ring retainer.

Since these sequences can be easily and safely performed so that the different components do not come apart during their assembly, we decided to adopt them as the two aforementioned independent subassemblies with which we wanted to divide the entire torch assembly (cf. figure 5.30 on page 174).

Examining carefully each of these subassemblies, we observed that both of them require similar two-step processes (pick-and-place operation) which for the former consists in collecting the ring retainer and clamping it with the left gripper, collecting the glass and laying it down on top of the ring retainer, and finally collecting the reflector and placing

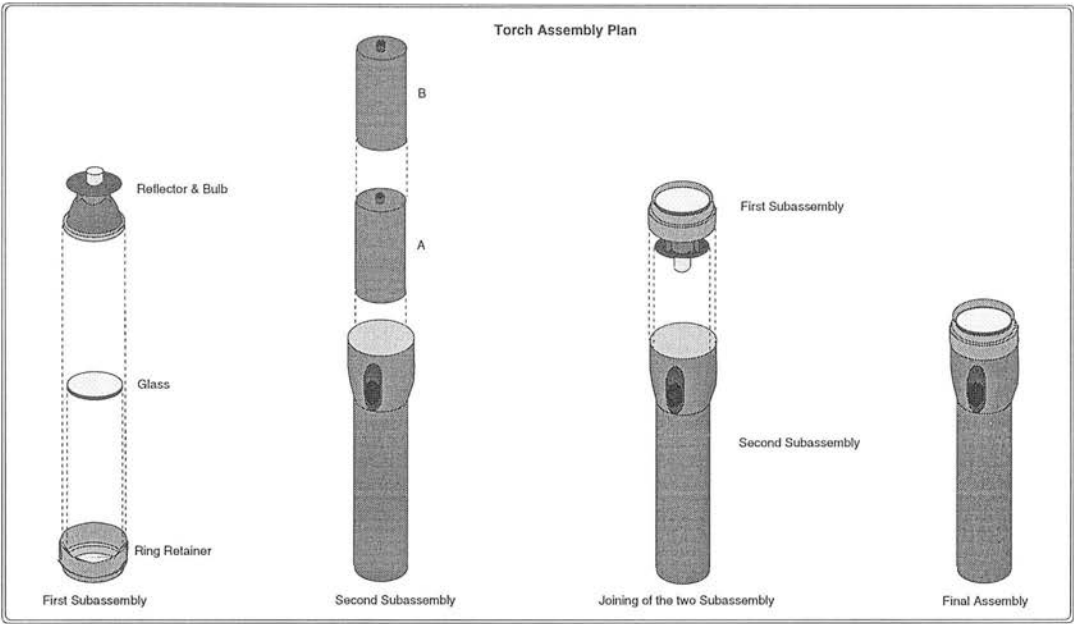


Figure 5.30: Torch Subassemblies.

it on top of the subassembly just built up, whereas for the latter consists in collecting the torch tube and clamping it with the left gripper, and collecting in turn the two batteries and letting them drop inside the torch tube. Thus, we need in both cases a sequence of one pick-up and one stack (cf. subsection 5.2.2.1 and 5.2.2.2 on page 139 and 143, respectively), which we label as pick-and-place module (cf. figure 5.31 here below). Notice that in the case of the tube-batteries subassembly we have used two

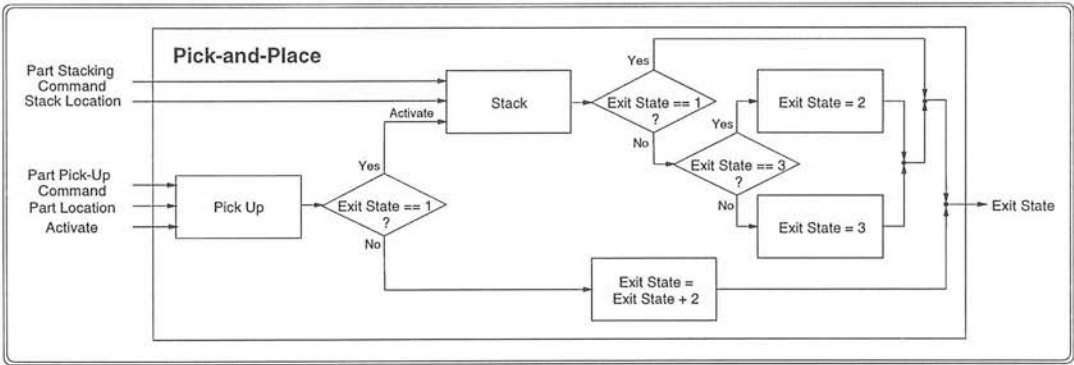


Figure 5.31: Pick-and-Place Operation Diagram.

stacking operations to assemble the two batteries. This was possible because of the retaining side effect of the tube which keeps the two of them one on top of the other

pick-up	
Code	Exit States
1	Part successfully collected
2	obstacle obstructs part lifting
3	part to collect not found
4	obstacle obstructs part approaching
5	obstacle encountered on the way to the location above part

Table 5.7: pick-up Outcomes.

stack	
Code	Exit States
1	part successfully stacked
2	no part to be stacked
3	obstacle obstructs stacking
4	location approaching
	obstacle encountered on the way to the location above stack

Table 5.8: stack Outcomes.

despite their inherent instability of remaining as such if considered by themselves.

We have to point out that, in order to perform one of these pick-and-place processes, we require four input parameters: a part collecting command, the location of the part to be collected, a part stacking command, and finally the location where the part has to be stacked.

As regards the outcomes of such a process, they depend on the exit state resulting from the composition of pick-up and stack. In order to determine thoroughly all the possible outcomes deriving from such a composition, we have to recall the exit states of pick-up and those of stack (cf. page 142 and 144, respectively), which for convenience we state again above in tables 5.7 and tables 5.8, respectively. We have to observe that we activate the execution of a stacking module only if a part was successfully collected (Exit state = 1). Thus, the only outcome which cannot occur out of the composition of a pick-up with a stack is that there is no part to be stacked. Therefore, considering the stacking of a part, we may have three possible outcomes (part successfully stacked and process succeeded, obstacle obstructed stacking, and obstacle obstructed stacking approach). These ones, combined with the other 4 possible outcomes from pick-up, bring the total number of outcomes out of the pick-and-place sequence to 7 (cf. table 5.9 on page 176). Recalling that both subassemblies mentioned earlier (cf. page 173) involve the collection and stacking of two parts onto a third one, we observe that each of them may be modeled with a composition of three pick-and-place operations: one for placing the base, and two for stacking the following two parts (cf. figure 5.32 on page 176).

Pick-and-place Module	
Code	Exit States
1	Both parts successfully stacked
2	Obstacle obstructed part stacking
3	Obstacle obstructed part approaching to stacking location
4	Obstacle obstructed lifting of part from its home location
5	Part not found in its home location
6	Obstacle obstructed approaching to part home location
7	Obstacle obstructed the way to location above part

Table 5.9: Pick-and-place Operation Outcomes.

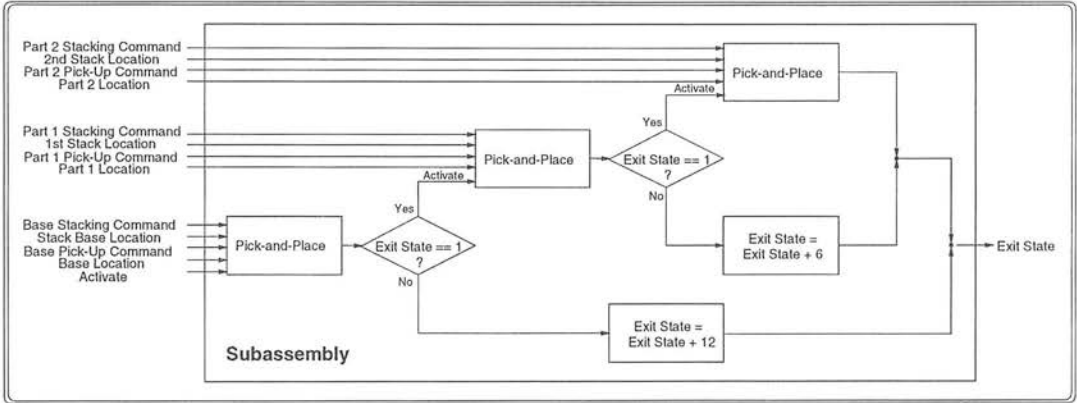


Figure 5.32: Diagram for both Subassemblies.

We have to remark at this point that the two pick-and-place operations mentioned above require four input parameters each. Thus, we require in total 12 inputs: three couples of collecting and stacking commands, three locations of parts, and three stacking sites. In this respect, assuming to stack the two parts involved on the same place where the base was laid down (left gripper site), we can actually feed just one stacking location and use it for the three pick-and-place operations.

As regards the outcomes of this process, they depend on the composition of the 7 exit states of each pick-and-place module. In this regard, we have to observe that, after having executed one pick-and-place, we proceed to the next one only if the exit state of the previous one is 1. Thus, taking into account the combination of the three operations, the total number of outcomes which may result from their execution sums up to 19 exit states (cf. table 5.10 on page 177).

Subassembly	
Code	Exit States
1	Both parts successfully stacked
2	Obstacle obstructed stacking of part 2
3	Obstacle obstructed approaching of part 2 to stacking location
4	Obstacle obstructed lifting of part 2 from its home location
5	Part 2 not found at its home location
6	Obstacle obstructed approaching to part 2 home location
7	Obstacle obstructed the way to location above part 2
8	Obstacle obstructed stacking of part 1
9	Obstacle obstructed approaching of part 1 to stacking location
10	Obstacle obstructed lifting of part 1 from its home location
11	Part 1 not found at its home location
12	Obstacle obstructed approaching to part 1 home location
13	Obstacle obstructed the way to location above part 1
14	Obstacle obstructed placing of the base
15	Obstacle obstructed approaching of the base to stacking location
16	Obstacle obstructed lifting of the base from its home location
17	Base not found at its home location
18	obstacle obstructed approaching to the base home location
19	obstacle obstructed the way to location above base

Table 5.10: Subassembly Outcomes.

At this point, in order to perform the two subassemblies mentioned at the beginning (ring retainer/glass/reflector and batteries/torch tube), we have to instantiate the subassembly process outlined above in each case with different input parameters. The former requires in particular the locations of ring retainer, glass and reflector (base, part 1 and 2), the location of the stacking site (left gripper location), one command to pick up the ring retainer and one to place it on the stacking site, one command to collect the glass and one to stack it on the ring retainer, and finally another command to pick up the reflector and yet another one to stack it on both glass and ring retainer. As regards the second subassembly, it requires tube and batteries home locations (base, part 1 and 2), the place where they have to be stacked (left gripper location), and finally six commands for picking up and stacking tube and each battery. In this respect, we have to remark that the two parts are equal in this case, thus we actually need just one couple of collecting and stacking commands for the two batteries.

It is worth pointing out that we assume to perform each subassembly in the same site (left gripper location). Thus, as soon as the first one is successfully completed, it is

necessary to move it away to some known work-cell location in order to make room for the other. Once this last has been successfully accomplished, the next stage is joining it with the first one. In order to do so, we observe that the torch tube is still firmly held by the left gripper³⁷ at the end of the second subassembly. Thus, we can perform its joining with the first one by keeping it clamped with the left gripper and by going with the right one first to fetch the other back from where it was temporarily laid down (temporary site) and then to fasten it on top of the torch tube (cf. figure 5.33 here below). In this regard, we have to remark that the orientation of the first subassembly

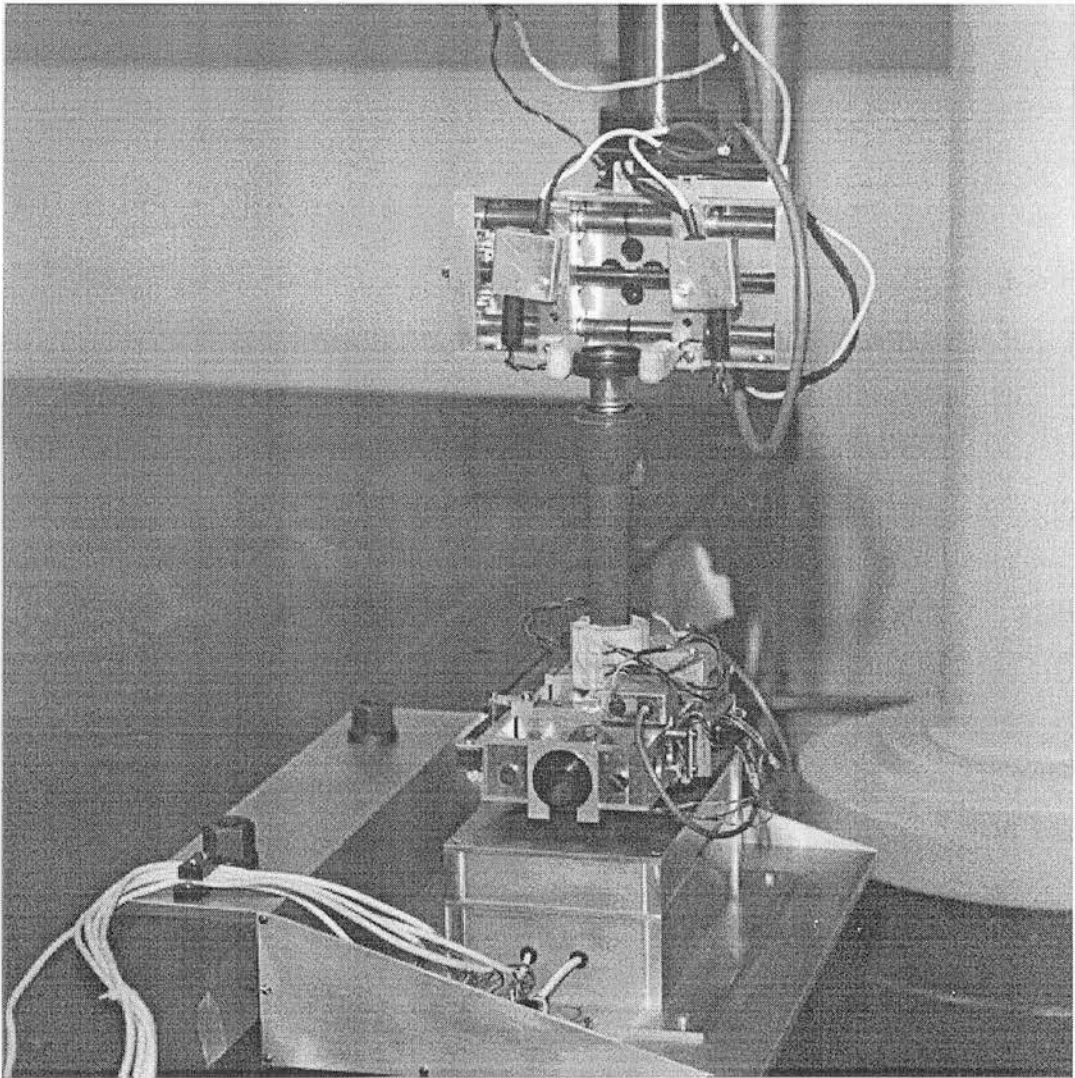


Figure 5.33: Screw Fastening of the Two Torch Subassemblies.

³⁷ This plays the role of a flexible fixture.

is actually upside-down with respect to the position it should have in order to be assembled with the torch tube (cf. figure 5.30 on page 174). Thus, before stacking it on top of the tube, we need to flip it so that it assumes the right orientation. Once such a position is achieved, the second subassembly can then be stacked and fastened.

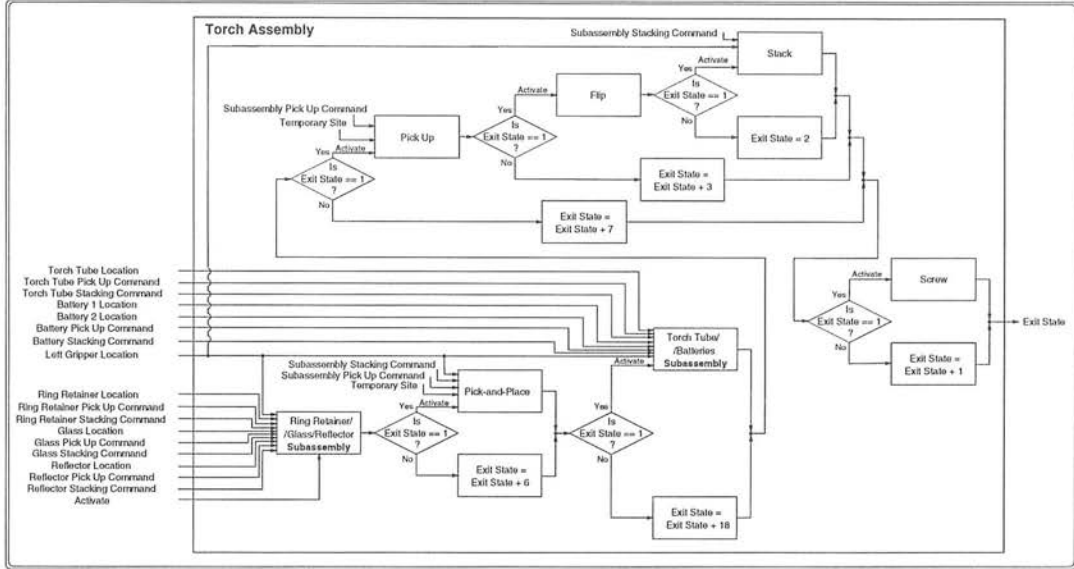


Figure 5.34: Behavioural Decomposition of the Torch Assembly.

Summarizing the discussion above, we can divide the plan to assemble a torch kit in five stages (cf. figure 5.34 above):

- subassembly of ring retainer with glass and reflector,
- saving of first subassembly at a temporary location,
- subassembly of torch tube with the two batteries,
- retrieval of first subassembly from its temporary site and its reorientation and stacking on the torch tube,
- screw fastening of the first subassembly to the second one.

The parameters which have to be fed as input are basically those required by the two partial subassemblies. We have to point out, though, that the assembly plan outlined above makes also use of some local knowledge (cf. subsection 4.1.1 on page 89) which in this case consists in a couple of pickup and stacking commands, and in a safe temporary site where to save the first subassembly.

As regards the possible outcomes of the torch assembly, since they depend on the combination of the modules composing the plan, we have to consider in turn the outcomes of each one of them. In this respect, we have 19 exit states out of the first subassembly, 6 more out of the pick-and-place module, further 18 out of the second subassembly, 7 more out of the retrieval stage, and finally 1 more out of the screw fastening, in total 51 exit states (cf. table 5.11 here below).

Table 5.11: Torch Assembly Plan Outcomes.

Torch Assembly Plan	
Code	Exit States
1	Ring retainer successfully fastened
2	Screw thread not started
3	Obstacle sensed during flipping of the first subassembly
4	Obstacle obstructs left gripper location approaching
5	Obstacle encountered on the way to left gripper location
6	Obstacle obstructs subassembly lifting from its temporary location
7	Subassembly to be collected not found at temporary location
8	Obstacle obstructs approaching to temporary location
9	Obstacle encountered on the way to go above temporary location
10	Obstacle obstructed stacking of battery 2
11	Obstacle obstructed approaching of battery 2 to stacking location
12	Obstacle obstructed lifting of battery 2 from its home location
13	Battery 2 not found at its home location
14	Obstacle obstructed approaching to battery 2 home location
15	Obstacle obstructed the way to the location above battery 2
16	Obstacle obstructed stacking of battery 1
17	Obstacle obstructed approaching of battery 1 to stacking location
18	Obstacle obstructed lifting of battery 1 from its home location
19	Battery 1 not found at its home location
20	Obstacle obstructed approaching to battery 1 home location
21	Obstacle obstructed the way to the location above battery 1
22	Obstacle obstructed placing of torch tube on the left gripper
23	Obstacle obstructed approaching of torch tube to stacking location
24	Obstacle obstructed lifting of torch tube from its home location
25	Torch tube not found at its home location
26	Obstacle obstructed approaching to torch tube home location
27	Obstacle obstructed the way to location above torch tube
28	Obstacle obstructed subassembly placing at temporary location
29	Obstacle obstructed subassembly approaching to temporary location
30	Obstacle obstructed lifting of subassembly from left gripper location
31	Subassembly to be saved not found in left gripper
32	Obstacle obstructed approaching to left gripper location
33	Obstacle obstructed the way to location above left gripper
34	Obstacle obstructed stacking of reflector

Continued on next page

<i>Continued from previous page</i>	
Code	Exit States
35	Obstacle obstructed approaching of reflector to left gripper location
36	Obstacle obstructed lifting of reflector from its home location
37	Reflector not found at its home location
38	Obstacle obstructed approaching to reflector home location
39	Obstacle obstructed the way to location above reflector
40	Obstacle obstructed stacking of glass
41	Obstacle obstructed approaching of glass to left gripper location
42	Obstacle obstructed lifting of glass from its home location
43	Glass not found at its home location
44	Obstacle obstructed approaching to glass home location
45	Obstacle obstructed the way to location above glass
46	Obstacle obstructed placing of ring retainer on left gripper
47	Obstacle obstructed approaching of ring retainer to left gripper location
48	Obstacle obstructed lifting of ring retainer from its home location
49	Ring retainer not found at its home location
50	Obstacle obstructed approaching to ring retainer home location
51	Obstacle obstructed the way to location above ring retainer

As regards screw fastening, it is worth pointing out that screw, although having three possible outcomes (cf. page 171), can in this case terminate with either 1 or 3, because this module is performed only if the second subassembly is successfully collected, thus the outcome of *part to be fastened not found* cannot occur. A similar line of reasoning applies to stack in the subassembly retrieval stage: such a module is in fact performed only if the subassembly is successfully fetched, thus its second outcome *part to be stacked not found* cannot occur either.

At this point, before concluding the subsection, we have to give a brief mention to a new module which appears in the plan discussed in this subsection: flip. In this regard, recalling the results of the survey on manufacturing operations (cf. subsection 2.3.1 on page 47), we notice that flipping objects is a common assembly task in manufacturing industry. Basically, it consists of manipulating the object currently held by the right gripper so that at the end of the manipulation its orientation is reversed. The behavioural module accomplishing this task, which we label as flip, is closely related to our guarded motion (cf. section 5.1 on page 102) but cannot be regarded fully as such, because this last affects all the robot joints during its execution. Thus, even though it is capable of roto-translating an object from one position-orientation to its reversed one, when the robot has to work in a location near its physical joint limits, our guarded mo-

tion may force the manipulator to exceed one or more of its joint limits in the attempt to accomplish its task causing a failure. Moreover, since we based it on a predefined VAL II Move command, we cannot rely that the robot rotates its wrist always in the same directions in all possible arm configurations. Our guarded motion can cope with the first problem in most situations but cannot do the same at all with the second one. By virtue of these observations we concluded that we needed a different motion command. In this respect, flip has to be regarded as a special sensor monitored motion command affecting only the joints at the wrist level. Furthermore, since it is a simple and convenient task occurring quite often in manufacturing industry, we consider it appropriate to include it in the basic set of elementary behavioural modules which we are looking for.

5.4.3 Torch Assembly Family Experiments

We have now reached the stage of describing the experiments run for the torch kit assembly. As done with the benchmark assembly family, our first concern was to set the work-cell and develop the assembly program implementing the plan discussed above in the previous subsection (cf. figure 5.34 on page 179) which makes use of the modules developed earlier for the benchmark assembly.

The goal which we aimed to reach with one torch kit was not just to assemble that particular kit, but was to develop a program capable of achieving the assembly of that kit as well as the assembly of other similar ones belonging to the same family. In order to do so, we set three assembly experiments: the largest torch kit, the medium one, and the smallest one. The following three subsections are dedicated to describe each of them.

5.4.3.1 Big Torch Kit

The first experiment we describe here involves the assembly of the largest kit. As done with the benchmark parts, our first step was to place the six parts of the torch kit (ring retainer, glass, reflector, torch tube, battery 1, and battery 2) at specific locations within the work-cell. Then, by employing a teach pendant, we recorded them as their home locations. Similarly, we recorded the locations of both left gripper and

temporary site. Once all these locations were set, we implemented the program as outlined in figure 5.34 by using the same pick-up and stack modules employed for the benchmark.

As discussed in the previous subsection, we divided the assembly plan in five different parts: ring retainer/glass/reflector subassembly, its saving at a temporary location, torch tube/batteries subassembly, retrieval of first subassembly from its temporary site and consequent reorientation and stacking on top of the torch tube, and finally screw fastening of the first subassembly to the torch tube (cf. page 179). In this regard, considering singularly the first four of them, we did not experience any particular problem. Indeed, testing them separately by means of one sequence of 60 trials, we recorded the full success in each case (100% success rate).

As regards the last stage of the assembly plan (the screw fastening of the ring retainer to the torch tube), we recorded a slightly different result: 57 successful fastenings out of 60 trials (95% success rate). The three failures (5% failure rate) were all due to a wrong start of the screw thread caused by the spring at the bottom of the tube which indirectly pushed the reflector out of axis³⁸. It is interesting to point out that all these failures occurred within the first 20 trials: afterwards, we did not record any more of them. We explain this outcome with the fact that the spring after many repeated experimental trials started wearing and losing part of its force.

After having tested all the different parts of the torch assembly plan, we experimented with the full program. In this respect, after having run a sequence of 60 complete assemblies of the torch kit, we recorded 58 successful ones (*ca.* 97% success rate). The two failures were again due to the spring push.

5.4.3.2 Medium Torch kit

The second experiment we are describing here concerns the medium torch assembly kit. After having also in this case placed the different parts at specific locations and recorded them as their home locations, we took the same assembly program used for the largest torch kit and instantiated it with the new home locations. As regards left

³⁸ Notice that this time we are testing the screw fastening in presence of the batteries inside the tube which are pushed upwards by the spring at the bottom of the tube.

gripper and temporary site locations, and pickup and stacking commands, we adopted for them the same ones used before.

In order to test the assembly program, we carried out the same pattern of tests outlined above. Thus, we performed a sequence of 60 assembly trials of the full medium kit, and recorded 59 successes (ca. 98% success rate). The only failure was also in this case due to a wrong start of the screw thread caused by the spring push. The higher success rate with respect to the previous assembly kit can be explained by the small physical size of the spring which, because of its smaller coefficient of elasticity, generates smaller forces opposing the screw fastening.

5.4.3.3 Smallest Torch kit

The third and last experiment we report here concerns the smallest torch kit. We repeated the same steps as done with the medium kit and recorded the new home locations of the parts involved. At this point, we instantiated the same assembly program by using these new locations and by keeping the rest of the parameters unchanged.

After having prepared the program, we ran a sequence of 60 experimental trials and recorded 60 successful assemblies (100% success rate). The higher success rate with respect to the previous kits is mainly due to the very small size of the spring involved which is capable of producing only very tiny push disturbing the successful screw fastening.

5.4.4 Torch Assembly Family Summary

The torch assembly family was our last experimental set up in our quest for the basic set of elementary behavioural modules with which to program manufacturing assembly tasks. As mentioned at the beginning of this section, the important reason which led us to opt for this kind of industrial product is that its assembly requires a very common task: screw fastening. We observed that in order to accomplish reliably such a task we need first to start the thread and then keep twisting the ring retainer until an opposing force triggers our sensor. In this regard, although both starting a screw thread and twisting can be performed by our guarded motion, as pointed out for insert-peg-with-

spring-retainer and snapfit, the behavioural module developed to accomplish it is general enough to be added to the set of the elementary behaviours we are looking for.

As regards the torch assembly itself (cf. subsection 5.4.2 on page 173), we decomposed the plan for accomplishing it in five parts: ring retainer-glass-reflector subassembly, its placing at a temporary location, torch tube-batteries subassembly, retrieval and reorientation of the first subassembly on top of the torch tube, and finally screwing it to the tube. This decomposition showed that the assembly plan was mainly made by an opportune composition of the same pick-up and stack modules developed for the benchmark assemblies. The two extra modules which emerged by this decomposition were flip and screw. The former reveals to be an important module when part reorientations at the wrist level are concerned. The latter is instead important for fastening matching threaded parts.

The aforementioned assembly plan was implemented and tested on three torch kits. Experiments showed that the manipulator agent was capable of quite reliably assembling all the kits by using the same program opportunely instantiated with the locations of the different parts (cf. table 5.12 here below).

Kit Type	Successes	Rate
Largest	58/60	97%
Medium	59/60	98%
Smallest	60/60	100%

Table 5.12: Torch Kit Experimental Results.

It is important to remark that the main result we achieved by experimenting with the torch assembly family was the development of two very useful behavioural modules (flip and screw) which, given their high occurrence in manufacturing industry, need to be included in the set of elementary modules which we are building in this document.

5.5 Summary

Let us now summarize the main results achieved in this chapter. Recalling that our final goal is to build a basic set of elementary behavioural modules with which to program 80% of the assembly tasks, we developed a sensor-based guarded motion module

(guardedmove) defined in terms of the robot basic motion command (cf. section 5.1 on page 102). Such a module allows our manipulator agent to correct and adapt its motion to the environment constraints without however performing any form of collision avoidance. We observed that such a module may be regarded as one of the most primitive behavioural modules in the sense given in definition 4.11 on page 95.

Analyzing such a module, we noticed that the motion might be achieved by straight lines or by joint interpolated ones. We opted for the former because it is the simplest between the two of them from the user point of view. We implemented this module so that it requires as input just the final destination and the speed to reach it. This last is necessary because we wanted to use guardedmove as a fine motion as well as a gross one. However, experiments showed that, mostly because of the compliance in our robot and gripper, its use in a fine motion application has to limit the speed to 3% of the full rate. As regards its outcomes, we observed that the two possible states in which the manipulator agent may be left at the end of its execution are either target location reached (1), or obstacle detected along the motion path before reaching the target destination (2).

guardedmove, as recalled above, may be fully regarded as the basic unit in terms of which most of the modules involving motions can be expressed. Even so, it is too low level both from a human operator and planner's point of view to be of any practical use.

In order to investigate which set of modules might be elementary and general enough to be used as a basis for a general purpose behaviour-based robot language, we considered three different and significant assemblies:

- a family of two benchmarks,
- the STRASS assembly,
- a family of three torches.

As regards the first one, we decomposed the assembly plan so that the final program was independent from any particular set of benchmark parts. Analysing such an assembly task, we isolated a first important module for performing round peg insertions which

we labeled as *peg-in-hole*. A close examination of this module showed its two main components: hole search and peg insertion. In this regard, we implemented the former using a hopping spiral search (cf. subsection 5.2.1.1 on page 126), and the latter using a thrust and correcting strategy (cf. subsection 5.2.1.2 on page 132). Both components were developed and tested showing encouraging level of robustness and reliability. Other two modules emerged from the benchmark assembly task were *stack* and *pick-up*. Given the limitation of our end-effector (two-fingered jaw gripper), we pointed out that the domain of objects we can deal with has to be limited to prismatic polyhedrons with parallel faces and to cylinders.

The first important result which we achieved by experimenting with the benchmark family was the fact that the assembly program designed for the metal kit had simply to be instantiated again with new input parameters in order to accomplish the assembly of the other kit. The other result was the development of *peg-in-hole*, *stack*, and *pick-up* which, because of their elementariness and high occurrence rate in manufacturing tasks, need to be included in the basic set of behavioural modules which we aimed to discover.

As regards STRASS, we pointed out that the assembly task involved in it could be viewed both as a particular form of *insert peg with retainer* and as a typical snapfit operation. In order to resolve both aspects of STRASS, we developed a *insert-peg-with-spring-retainer* module based on our *guardedmove* and a *snapfit* module based on a typical unguarded motion. Both modules, although being very close to *peg-in-hole*, required to insert a peg by overcoming an opposing force exerted by a spring retainer. This was quite a big problem for our robot which is equipped just with a simple differential touch sensor. However, thanks to the strategy outlined in subsection 5.3.1 on page 157, we managed anyway to express the first one of them in terms of our *guardedmove* by exploiting the mechanics of the particular parts involved. In this respect, though, the implementation we developed solves this particular assembly but its applicability is limited to the domain of parts we have considered. The other module (*snapfit*) is in this sense more general, but is limited by the fact that it has no means to control unpredicted situations during the insertion. However, despite their limitations, they accomplish quite useful elementary operations, thus we included both in our set

of elementary modules.

As regards the last assembly experiment, it concerned a family of real industrial products: a set of similar electric torches. As done with the benchmarks, we analyzed the task and, by decomposing it in its basic components, we isolated other two modules: flip and screw. The important result which we achieved was the development of a parameterized assembly program capable of accomplishing each member of the torch family with an opportune instantiation. In this regard, we have to stress the point that the input parameters did not concern any specific torch kit but simply the locations where the parts had to be collected. Thus, the program was actually independent of any particular set (cf. page 77).

The high success rate of the experimental tests showed in each case that the set of modules found so far was not only elementary enough but also general enough to be applied in many situations. The next chapter will analyze further this matter and will define more precisely all of the components of the basic set.

Chapter 6

Analysis of the Results

With this chapter we have finally arrived at the main point of our research: the discussion of the existence of a limited, basic and convenient set of behavioural modules with which to program 80% of the manufacturing tasks on our assembly domain (cf. subsection 3.2.3 on page 73 for the derivation and discussion of this figure).

The experiments described and discussed in the previous chapter represented for us a tool with which to conduct our investigation. However, we would like to stress the point that they were not meant to be a theoretical proof of the existence of such a set, nor to exhaust all the possible manufacturing assembly tasks. Indeed, they just set up good experimental grounds whereon to base and support our speculation of its existence. The particular assemblies we selected were chosen because in order to be accomplished they required some operations which, according to the survey discussed in subsection 2.3.1 on page 47, occurred very often in a large number of manufactured products.

The aim which we want to achieve in this chapter is first to argue in favour of the existence of the aforementioned set of elementary behavioural modules, second to synthesize the results gathered by the assembly experiments described throughout chapter 5, and finally to propose the foundations of a behaviour-based robot assembly language. To this end we divide the chapter in three sections: the discussion about the existence of a set of basic general-purpose modules (section 6.1 on page 190), the synthesis of the modules emerging from the experimental results (section 6.2 on page 192), and the proposal of a behavioural language which may be built on top of them.

6.1 Existence of the Basic Set

As mentioned in section 3.1 on page 66, although it is always theoretically possible to have a large variety of geometrically different manufacturing tasks, the range of those which actually occur in practice is considerably smaller. Indeed, thanks to the survey results presented in subsection 2.3.1 on page 47, we know that 95% of them cluster in just seven classes (cf. figure 2.10 on page 53).

According to the behaviour-based assembly paradigm, every assembly task may be viewed as a single grand-behaviour implemented as an opportune composition of several behavioural modules. In this regard, we do not know as yet how many modules we would need in order to accomplish each of the aforementioned 7 classes on the domain of real industrial parts. It may happen that, although the different assembly tasks are practically limited to a very small number, the set of modules required to perform them is very big or even infinite considering all task varieties. Thus, the problem we aimed to tackle was to investigate first whether there exists a set of general-purpose modules, and second whether such a set is limited (cf. subsection 3.1 on page 66). In this respect, we have to point out that the assembly world on which we carried out our research is considerably smaller than the vast domain of all the possible industrial assembly parts. However, if modules cannot be designed in order to have general applicability even on this small domain, then there would be no sense wondering how many of them we need for larger ones, because we would have to define *ad hoc* solutions for every new assembly task. If, instead, they do exist and a set of them is capable of covering at least 80% of the manufacturing tasks (cf. subsection 3.2.3 on page 73), then the question at issue would be whether a higher level programming language could be based on them.

Arguments in favour of the existence of such a set of general-purpose modules come from the results of other previous works. Balch, as summarized in subsection 2.3.2 on page 53, developed within our same paradigm a set of 23 low-level highly parameterized software modules whose generality, which was however tested on just one assembly test bed, derived from the fact that they did not have one specific purpose. Nnaji in his assembly programming system (RALPH) proposed instead a set of 14 general-purpose commands following the more conventional classic approach to robot control

(cf. subsection 2.3.3 on page 59). In both cases, a generic assembly was viewed as decomposed in terms of a limited set of elementary units.

Working within the framework of the behaviour-based assembly approach as Balch did and experimenting with some selected assemblies, we managed to isolate a small group of behavioural modules which showed an interesting level of generality within the scope of their applicability on the assembly domain considered. This was demonstrated by the fact that the assembly program based on them and developed in order to accomplish one benchmark kit was capable of performing a similar one simply by using a new instantiation of it (cf. section 5.2 on page 115). The same result was also reconfirmed by the torch assembly program which was capable of achieving the assembly of three similar electric torches by using opportune parameter instantiations. Indeed, the interesting result gathered here was the fact that the two assembly program, meant for very different assemblies, were actually making use of common modules (stack and pick-up).

These results pointed to the fact that it is actually possible to define behavioural modules which have a more general applicability than the simple one which they are initially designed for. In this regard, though, we have to remark that our investigations were conducted on a restricted domain of assembly parts¹ due to hardware constraints. Thus, although we implemented modules accomplishing some assembly tasks which cover about 80% of the manufacturing processes, the generality has to be limited to such a domain. However, we have to stress that employing a more sophisticated gripper, such as a multi-fingered one, can greatly enlarge such a domain. Thus, we argue similarly to Hopkins (cf. section 3.1 on page 66) that the existence of a limited set of general-purpose modules is a viable proposition.

An interesting observation which we ought to mention at this point is that we can program most of the fine motions of our manipulator agent simply by resorting directly to our `guardedmove` (cf. section 5.1 on page 102), which has to be thought of as one of the most primitive behavioural modules (cf. definition 4.11 on page 95), or to combinations of it. The availability of an exit state coding within its limits the outcome of the motion makes such a module more powerful than a simple robot motion

¹ Polyhedrons with parallel faces and cylinders.

command, because it allows the module, as opposed for instance to a VAL II motion command, to be used combined with others to accomplish many different tasks and hence achieve different goals, such as a nut fastening or a peg insertion. In this respect, we have to say that move in VAL II can be used to program very diverse assemblies². However, a program viewed in these terms would be nothing more than a mere sequence of point-to-point motions of the end-effector with no control on error situations unless these are specifically programmed in it with great detail.

Considering our experimental results and the observations drawn above, we conclude that we can define behavioural modules which are of more general use as shown by benchmark and torch assembly programs. As a consequence, we can define a limited set of them for programming many different kinds of assembly tasks. However, there may exist more than one set to be used for such a purpose. Thus, we need to select one which is general enough to cope with as many assembly tasks as possible, and easily used or learnt by human operators or automatic planning systems.

Since our experimental results support the argument about the existence of a limited set of general-purpose behaviours for programming a large number of manufacturing tasks within our assembly domain, we may wonder how many and which elements it is made of. If their number is too big, then such a set would not be of great help. If it is instead small enough, we might actually base an entire general-purpose behavioural language on top of it. In this regard, we dedicate the next section to synthesize a set of modules from the assembly experiments described in chapter 5 and the section after to discuss how it compares with the set of basic manufacturing tasks found by Kondoleon and with the set of general-purpose modules found by Balch.

6.2 Synthesis of the Experimental Results

As outlined in section 3.2 on page 68, our project consisted in a three-step process: selecting some significant assemblies, decomposing them up to their minimal behavioural terms, and finally synthesizing out of them those modules which are not only of

² The language designers considered obvious, and it is very widely accepted, that almost all assembly tasks can be expressed as a sequence of point-to-point motions, plus gripper, speed, and acceleration control. Consequently, "guarded moves" should share this generality of "move", and being more sophisticated, even exceeding it.

general applicability but also ergonomic.

As regards generality, we have to premise that our assembly domain because of hardware constraints is quite restricted, thus the modules we developed in the previous chapter cannot claim applicability beyond it. However, the fact that a small group of them is actually capable of coping with 80% of the assembly processes on such a domain points the fact that a similar limited set may be developed for a larger assembly world. We argue elsewhere (cf. subsection 3.2.3 on page 73) that the 20% we omit from our experimental implementation is simply more of the same, and contains no obstacles to implementation. As is the way with exceptions, however, the remaining 20% of tasks would require a great deal of extra implementation, and so was omitted from this exploration.

Looking back at the experiments we discussed in the previous chapter, let us now briefly summarize the main results we gathered from them.

6.2.1 Development of a Guarded Motion

The first result we recall concerns the development of a guarded motion which allows our manipulator agent (Adept 1 SCARA robot) to correct and adapt its motion to the environment constraints during the travel towards a target point within the work-cell. The module implementing it makes use of a piezo-film wrapped around the jaws of a two-fingered gripper. Such a film acts like a simple differential touch sensor which can detect the occurrence of event contacts only (cf. subsection 3.2.4 on page 80). Thus, our `guardedmove` does not perform any whole arm collision avoidance like in [Strenn *et al.* 94] and in [Feddema & Novak 94]. Nevertheless, it shows the interesting characteristic of reacting immediately on the occurrence of an event, a feature which is similar to the reflexive behaviour proposed in [Wikman & Newman 92] and in [Wikman *et al.* 93] but with the difference of being presented as a packaged module.

Our `guardedmove` is implemented so that it can be used as a fine motion as well as a gross motion. In order to do so, we have to provide as input the target location together with the arm speed rate. However, because of delays within the control loop of our manipulator agent, a fine motion application requires to limit such a rate within 3% of

the agent full velocity. This guarantees errors to be contained within the compliance of the finger rubber pads around which our sensor is wrapped.

Another interesting result that we want to draw attention to concerns the outcome of our guarded motion. Reprising an idea from Balch, we implemented it so that information about the result of the motion is returned as output. However, differently from Balch, we encoded the information concerning each outcome with a number and returned at the end of its execution in just one variable which we labeled with exit state. This characteristic makes our guardedmove a crucial brick on top of which we can build more complex modules.

The module viewed within the behaviour-based assembly perspective may therefore be regarded as one of the most primitive modules (cf. definition 4.11 on page 95). Indeed, it may be viewed as one of the most important, though we have to observe that it is still a low-level unit which may not be very convenient to be used from a human operator's or an automated planner's point of view.

6.2.2 One Assembly Program for Similar Assemblies

The second important result which we have to consider concerns the development of assembly programs. As mentioned back in section 6.1 on page 191, experimenting with two similar assembly kits (metal and wooden benchmarks), we managed to define a program in terms of modular units (behavioural modules) which was capable of achieving their assembly simply by instantiating it with new input parameters. In this regard, we have to point out that the program developed for a simplified version of the assembly (cf. partial benchmark assembly in subsection 5.2.2 on page 136) required very little effort to be generalized to achieve the complete one (cf. full benchmark assembly in subsection 5.2.3 on page 147).

Of course, the original modules and programs were designed with this generalization in mind. However, given this intention, the interesting point is that the generality was easily achieved, which lends support to the thesis contention that such generality is well within the scope of the technological resources and human ingenuity.

A similar result emerged from the assembly of a family of three electric torches. The

program designed to assemble one member of such a group was actually capable of accomplishing the other two members by its instantiation with new input parameters typical of the particular torch to be considered. We have to remark that this program was not only defined in terms of modular units as the benchmark one but some of these units were actually those defined for the benchmark assembly.

We have to observe that the two results recalled above were obtained by testing the two assembly programs just on a small group of similar assemblies. However, considering them globally, they point to the fact that it is possible to define a robot program of more general use than one single assembly. The relative ease with which this was accomplished for these two families (torches and benchmarks) suggests that, at least for families of assembly, this kind of generalization is well within the scope of current resources.

6.2.3 Synthesis of our Basic Set

Recalling that our final goal is to investigate the existence of a small set of elementary general-purpose modules capable of coping with 80% of the manufacturing processes (cf. section 3.1 on page 66), we have now reached the stage of synthesizing such a set out of our experiments. Before starting, though, it is worth also recalling that, because of hardware constraints, our assembly domain is restricted just to polyhedrons with parallel faces and cylinders. Thus, the generality and applicability of the set which we synthesize in this subsection has to be limited to such a domain.

The following subsections will discuss the various modules emerged from the experiments carried out in chapter 5 on page 101.

6.2.3.1 peg-in-hole as a Basic Module

Analyzing the benchmark assembly problem, we observed that it involved a peg-in-hole operation which is a very common manufacturing task (cf. subsection 2.3.1 on page 52). We did not attempt in our investigation to give a solution to the general form of such a problem. Nevertheless, although restricting ourselves to examine just round single peg-in-hole, we tackled a more general form of it involving the insertion of a peg in

two coaxial holes which, following the terminology introduced in [Wu & Hopkins 90], we labelled as tandem peg-in-hole.

A close examination of the task revealed that it is actually made of two main parts: search for a hole, and peg insertion. Thus, the module we developed in order to accomplish it, which we called peg-in-hole, was making use of these two components which were both based on our guardedmove (cf. subsection 6.2.1 on page 193).

As regards the first component (search for a hole), we did not use any vision or optical system as in [Martínez & Llario 87] or [Paulos & Canny 93]). We adopted instead a technique consisting in tilting a peg³ with respect to the normal to the surface where the hole is located and in hopping on such a surface along a square spiral path: as soon as a hop does not detect a contact between peg tip and surface, a hole is detected. Notice that the idea of a tilted peg is similar to the one presented in [Bruyninckx *et al.* 95], however, differently from us, Bruyninckx does not perform any search: he uses it simply to locate the hole boundary and hence the insertion axis, assuming of course that the hole lies beneath the peg. This is an important difference because the information about the outcome of a search for a hole can in our case be used for accomplishing a different task than peg insertion only, as for instance checking the presence of a step-wall⁴.

As regards the second component (peg insertion), we did not use (as opposed to [Strip 88]) any complicated sensing device to control the insertion. Indeed, the sensor employed was quite crude but the data we gathered out of it revealed to be very informative for controlling our agent. As regards the insertion strategy, we implemented a technique which allowed a manipulator agent to insert a peg through two coaxial holes belonging to detached parts. Such a technique is capable of coping within certain limits with axes misalignments (max. $\frac{1}{2}mm$). However, we have to point out that its applicability is restricted just to round peg-in-hole tasks. Nevertheless, our strategy as opposed to the one presented in [Caine *et al.* 89] works for both chamfered and chamferless pegs in tilted and non-tilted holes with respect to the X-Y plane of

³ Notice that this method applies for both chamfered and chamferless peg.

⁴ The search in this case would terminate with the outcome of *obstacle detected during search* (cf. subsection 5.2.1 on page 120).

the robot frame system.

Summarizing our analysis of peg-in-hole, we can say that, although being restricted to particular form of pegs, the module is general enough to have wide applicability within our assembly domain. Thus, we include it in our set of basic behavioural modules. We ought to remark, though, that a further development of the insertion component is required in order to extend the range of peg insertions it can deal with. In this regard, we expect to provide peg-in-hole with an extra parameter carrying the information on the particular insertion required. However, our insertion implementation can still be used as a basis for this module enhancement for some orientation dependent insertions such as square, rectangular, or prismatic pegs. In any case, we expect that very awkward insertions such as a key in a lock may require a dedicated module. Though, considering that nowadays industrial products are designed so that to facilitate automatic assembly, this kind of tasks are not very common. Thus, a peg-in-hole enhanced to cope with round pegs as well as prismatic ones may well cover the great majority of insertion tasks.

As last remark, we have to observe that the case of a very long peg shaft, although being in principle a peg-in-hole operation, should not be viewed as such, because gravity and center of mass play a more crucial role with it than with a usual peg-in-hole task. As will be discussed shortly, a case like this one would be better suited to a stacking module.

6.2.3.2 pick-up as a Basic Module

Considering once again the benchmark assembly experiment, we observed that it involved another quite common operation: parts collection. In this regard, we developed a module based on guardedmove which we called pick-up. Such a module is the one that constrains most the assembly domain which we can cope with. In this respect, we use a simple two-fingered jaw gripper which restricts our assembly world to just polyhedrons with parallel faces and cylindric parts. A multi-fingered end-effector enlarges the range of objects our agent can manipulate, thus we expect that the development of a more complex manipulator would extend the applicability of this module to a wider domain and therefore extend its generality.

Our pick-up as it is developed now does not perform any grasp planning as for instance in [Caselli *et al.* 93], [vanBruggen *et al.* 93], and [Joukhandar *et al.* 94]. This is in line with our choice to work with the hybrid approach of the behaviour-based assembly paradigm, which assumes the use of a planner to compute this information. Thus, pick-up is developed so that a suitable grasping position is opportunely instantiated by a planner or a human operator.

The module performs always the same pattern of actions and this makes it independent from any particular part⁵. Indeed, although we experimented just with vertically down collections, our module is capable of performing horizontal collections. However, we have to observe that this is made possible by the fact that our agent is capable of computing relative transformations which enable it to reason in terms of the gripper Z-axis and not just in terms of the robot one.

Parts collection is an important operation which, although not being strictly speaking an assembly process, needs to be considered because many manufacturing tasks require at one stage or another to collect a part in order to proceed with the assembly. Thus, we decided to include the behavioural module implementing it in our basic set.

6.2.3.3 stack as a Basic Module

Another common operation emerged from the benchmark assembly experiment is stacking. In order to cope with it, we developed a stack module which, likewise pick-up, is based on our guardedmove. The module, although recalling a form of placing command in RALPH (cf. subsection 2.3.3 on page 59), does not perform any planning. It simply assumes a planner or a human operator to provide as input the appropriate information to carry out the task.

Our stack was developed and tested for our assembly world which consists basically of polyhedrons with planar parallel faces and with cylindric prisms. However, we have to point out that its applicability extends beyond such a domain because we can actually stack objects which need not necessarily have a planar base. This feature is possible because stack does not depend on the particular object our agent is holding. In other

⁵ Notice that the module expects the part to be collected at the location specified as input.

words, since it has not got any knowledge of the shape of the object, it is not bound to it. Thus, we can still use our stack module to place, for instance, a spoon in its slot of a cutlery case.

Another important point regarding our stack module is the fact that it can perform side stacking within of course the limits dictated by parts stability. This applies well to our restricted assembly world of polyhedrons and cylinders⁶ but it may show problems if we try to use it to make a side stacking of a spoon in its slot as in the aforementioned example.

As pointed out at the end of section 6.2.3.1 on page 197, peg insertions with long peg shafts are not well suited for our peg-in-hole. However, provided that such a peg is held at the level of its centre of mass along its shaft, we can still attempt the insertion by employing our stack which would first move the part above the hole and then let it drop inside by taking advantage of gravity and mechanics. This strategy, though, does not give any guarantee of a successful insertion unless the hole is considerably larger than the shaft.

As a final comment, we observe that stack, although developed for our limited assembly domain, showed interesting characteristics of generality. Besides, this module used together with pick-up forms the basis of a pick-and-place operation which as pointed out earlier for pick-up, is a very important task. Thus, considering its elementariness, usefulness, and generality within our domain, we decided to include stack in our basic set.

6.2.3.4 grasp and ungrasp as Basic Modules

The last two operations which emerged from the benchmark assembly experiment were grasping and ungrasping. The two modules implementing them (grasp and ungrasp) are both based on the same kind of sensor as guardedmove. Thus, a part is successfully gripped if a contact between fingers and part is detected.

Both modules constrain the range of objects which our manipulator agent can deal with. In this respect, because we are using a two-fingered jaw-gripper, our assembly

⁶ Notice that cylinders are intended to be stacked on their base.

domain results limited just to rigid prisms with parallel faces or cylinders. However, it is interesting to point out that we implemented our grasp and ungrasp modules in order to deal also with hollow parts. Thus, we can for instance collect a rigid box by gripping it either from outside, in this case by closing the gripper fingers, or from inside, in this case by opening them. Similarly if a part is held from inside, it has to be ungripped by closing fingers, and *vice-versa* if it is held from outside, it has to be released by opening fingers. Notice, though, that ungrasp does not keep track of how a part was collected, because it expects this information to be provided as input.

Both modules show general applicability on our assembly domain, thus decided to include both in our basic set.

6.2.3.5 insert-peg-with-spring-retainer and snapfit as Basic Modules

Analyzing the STRASS assembly experiment, there emerged another interesting operation consisting of mating a peg into a hole by overcoming the opposing force of a spring retainer at the rim of a hole. Such an operation showed similarities to the retaining operation studied by Kondoleon (cf. subsection 2.3.1 on page 47), however it differs from his because the STRASS retainer is not inserted at the end in order to fix the mating of the parts involved but is actually an integral component of the assembly itself. Indeed, the presence of a spring retainer, which exerts only a temporary opposing force to the insertion, makes this operation more similar to a snapfit.

In order to accomplish the assembly, we developed a behavioural module called insert-peg-with-spring-retainer which is based on our guardedmove. Such a module solves the task by exploiting the mechanics of the particular parts involved and shows an interesting reliability for a large variety of opposing forces determined by the compression of the spring retaining device. However, we have to observe that the strategy adopted was devised for the specific parts of STRASS. Thus, its applicability may not extend beyond our restricted assembly world. Nevertheless, we think that it may still be useful for tasks where the retainer is not rigidly fixed but can slide along its axis like snap rings. Therefore, insert-peg-with-spring-retainer, although not having extensive application beyond our assembly domain, may still be useful for this kind of situations.

Examining the snapfit aspect of STRASS, we proposed a different answer by developing a module, which we called snapfit, based on the more common unguarded motion. This is not bound to a particular shape of the peg as the previous one, and performs the mating by a direct thrust of the peg inside the retaining hole. However, we have to point out that such a module, although showing a considerable degree of reliability and robustness, has no control over unpredicted situations during the mating such as the presence inside the hole of an obstacle besides the retainer. Moreover, it relies on the assumption to know the depth of the hole as an input parameter provided either by a planner or by a human operator. This may not always be available, however, we can still determine it with our limited sensor by employing the technique for measuring the length of the peg shaft (cf. subsection 5.2.2 on page 137). This time, though, we touch first the area near the rim of the hole and then its bottom by using a long thin stick as a probe.

6.2.3.6 screw as a Basic Module

Considering our last assembly experiment, namely a family of electric torches made of a torch tube, a ring retainer, a reflector, a plastic glass, and two batteries, we noticed that the assembly plan for accomplishing the task involved a very common parts joining operation: screw fastening (cf. subsection 2.3.1 on page 52). We solved the problem by developing a dedicated behavioural module called screw allowing our manipulator agent to fasten nut-like parts to matching threaded ones.

The method implemented in the module for performing the operation is modeled on a human-like nut fastening which consists in consecutive twistings of the nut. However, differently from the approach taken in [Loncaric 87] and [Tao *et al.* 90], we did not develop it by exploiting a stiffness controlled manipulator. This is due to the more limited sensing capability of our agent which can just detect variation of forces greater than an internal threshold.

The module assumed to have available a second gripper placed on the table to be used as a flexible jig. As explained in subsection 3.2.4 on page 78, this hardware set up allowed our agent to start the thread more easily and reliably. However, we have to stress that our screw relies on it just for starting the thread and not for the actual

twistings of the nut-like part. Thus, the presence of the second gripper should not be viewed as a crucial limitation, since the module can be easily redefined without making use of it. A limitation comes instead from our agent sensing capability: it cannot in fact distinguish between a nut just partly fastened because of a wrong start of the thread and nut completely fastened. The use of V-shaped jaws allowed us to reduce this problem, and, indeed, we did not record wrong fastening after having employed them.

It is interesting to point out that the module does not rely on the parts to be placed in a specific orientation, because, thanks to the one-turn twists performed by both manipulator and table gripper in opposite directions, the thread is guaranteed to start whatever the initial position of the two parts is. If the module is redefined without making use of the left gripper⁷, then this feature may still be implemented by letting our agent perform two twists in the same direction.

Our screw module, although tested just on a group of torch ring retainers, can also be applied for long and short externally-shaped prismatic nuts. However, it cannot be used for externally threaded large hollow parts when these are gripped from inside.

Another interesting remark concerns its use in combination with stack. In this regard, we have to point out that we can accomplish an operation of push-and-twist (cf. figure 2.8 on page 47) by performing stack and screw in sequence.

Taking into account the observations drawn above, we conclude that our screw module, although developed and tested for a rather limited assembly world, shows interesting features which may extend its applicability beyond the tested world. Thus, considering its usefulness and generality over our domain and compared them with its limitations, we decided to include it in our set of basic modules.

6.2.3.7 flip as a Basic Module

Analyzing once again the torch assembly experiment, we noticed that it involved an operation consisting in a complete reversing of a part. This operation, although not as common as those previously examined, does still occur and is usually intended as a

⁷ Table gripper.

preparatory step preceding a specific parts joining process.

The operation is achieved first by holding a part with a gripper so that gripper and robot Z-axes are orthogonal, and then by reversing the orientation of this last. In this respect, we have to observe that it is actually closely related to a particular form of motion where both start and target point are spatially coincident but with the pitch reversed. Our `guardedmove`, although being capable of achieving such a particular motion, cannot be used to accomplish it because it gives no guarantees that it would rotate the manipulator wrist always in the same way in all possible arm configurations. This is due to a limitation of many robot control languages which specify rotations in terms of end positions and not directly, leaving in this way the choice of rotation to the interpreter depending on the particular joint angles configuration which the manipulator arm is in. Thus, we developed a dedicated module, which we called `flip`, based on the same outline of `guardedmove` but with the difference of directly driving the robot joints at the wrist level.

The module acts on the pitch and roll angles of the robot wrist, therefore it does not necessarily require to have gripper and robot Z-axes orthogonal to each other. We can flip any wrist position simply by computing the supplementary angle of the wrist pitch with respect to the robot Z-axis and then by rolling the wrist 180° .

We have to observe that the module is applicable in many arm configurations but may fail to perform in regions close to the manipulator main body.

As a conclusion we have to say that the task does not occur so often, however a module accomplishing it becomes very handy, and indeed sometimes indispensable for avoiding further introduction of uncertainty, when parts reorientations are required. Thus, considering its utility and elementariness with respect to our assembly domain, we decided to include it in our basic set.

6.2.4 Basic Set

The analysis carried out in the previous subsections outlined a group of elementary modules which may constitute an embryo of a basic set. In this respect, we have to point out that all these modules are intended to be applied just on our rather restricted

assembly world.

At this point let us summarize the results emerged from our assembly experiments.

The first test bed (benchmark) brought to light two very common assembly operations which we accomplished by developing two behavioural modules (peg-in-hole and stack). We noticed that during such an assembly another further operation was quite frequently recalled: parts collection, which was accomplished by developing a dedicated pick-up module. In this regard, we also developed two more modules for the actual parts gripping and parts releasing (grasp and ungrasp, respectively). It is important to point out that none of these five modules performed any planning.

In the second test bed (STRASS) emerged two further operations (retaining and snapfit) which we accomplished by implementing two modules (insert-peg-with-spring-retainer and snapfit) for accomplishing the particular form of retaining defined by STRASS. The former was more bound to the particular parts involved, however we decided to keep it because it turns out to be useful for performing some kinds of snapping operations, such as for example snap ring.

The third and final test bed (torch) brought to light a very common parts joining process: screw fastening. However, this was not of simple nut-and-bolt type. Indeed, it consisted of very large diameter parts with their threads pressed from steel sheet. This is more difficult, and it is in fact quite tricky also for a human being too do. We accomplished this operation without resorting to a dedicated tool by implementing a module modeled on a human-like nut screwing (screw). However, we have to stress the point that such a module is not meant to represent a substitute of the aforementioned specialized screwing tools. As final remark, we have to say that from the torch assembly emerged also another operation involving parts reorientation. We accomplished it by developing a dedicated module (flip), similar to guardedmove, and capable of acting directly on the robot joints at the wrist level.

As a conclusion, the group of modules we synthesized out of our experiments is composed of 9 elements:

- peg-in-hole for peg insertions,

- pick-up for parts collections,
- stack for laying parts down,
- grasp for gripping parts,
- ungrasp for ungripping parts,
- insert-peg-with-spring-retainer for retaining and snap ring operations,
- snapfit for snap fit tasks,
- screw for nuts screw fastenings, and
- flip for parts reorientations.

All these modules are general enough in our domain that they can be used in many different situations, and at the same time they are easy to use by human operators or automatic planners. Anyhow, we have to stress the point that such a group represents just the embryo of a basic set of elementary behavioural modules which are general-purpose with respect to the assembly domain on which they are applicable. This assembly world may be extended, however this requires further hardware and software development. By virtue of these observations, we conclude that if we want to create a behavioural robot language based on the above listed modules powerful enough to deal with 80% of the most common manufacturing assembly processes, we need to include them all in the set of its behavioural commands.

An important observation which we have to draw at this point concerns their reliability. In this regard, although they reached in average about 98% success rate, we have to say that for the purpose of our thesis we considered that acceptable, since what we were investigating was the development of a toolkit of modules and not their specific reliability performance. However, we expect them to require some further development in order to reach the stage of industrial acceptability.

As final remark, we have to point out that the set listed above should be viewed as a prototype which is able to cope with 80% of manufacturing assembly tasks on our rather restricted assembly domain. Enhancing this domain may require some more modules to be added to the set. Nevertheless, based on the grounds of our

experience, we expect their number to be rather limited because we can actually define assembly programs capable of dealing with similar assemblies simply by instantiating them with appropriate parameters. Anyhow, our work should not be interpreted as a theoretical proof of the existence of a basic set of behavioural modules, but it does set up experimental grounds to support such a thesis.

At this point, it is worth comparing our basic set first with Kondoleon's assembly processes (subsubsection 6.2.4.1 here below) and then with Balch's general-purpose modules (subsubsection 6.2.4.2 on page 208).

6.2.4.1 Comparison with Kondoleon's Set

The set of basic manufacturing operations found by Kondoleon (cf. subsection 2.3.1 on page 47) was composed of twelve elements. Comparing it with the set of modules we experimentally synthesized, we notice that ours does not cope with eight tasks of the former: push-and-twist, force-fit, multiple-peg-insertion, remove-location-peg, crimp, provide-temporary-support, remove-temporary-support, and weld. However, we have to point out that the scope of his work was the set of all the manufacturing processes which include feeding and shaping, assembly, and fixing, whereas ours is instead concerned just with assembly processes. Anyhow, it is worth making some observations for each operation considered by Kondoleon.

To start with, let us point out that we can easily perform Kondoleon's push-and-twist task by using a sequence of our stack and screw modules. In this regard, though, we need to remark that this is possible as long as the hole is large enough to let a part drop in. Kondoleon's force-fit can then be accomplished within the limits of the robot strength by employing our snapfit module.

As regards multiple-peg-insertion, we have to observe that our modules cannot cope with it because we assumed at the beginning of our research not to consider it. Thus, we would expect a dedicated module to be developed and added to our set in order to deal with such a process. However, differently from Kondoleon, we considered a more general form of peg-in-hole operation which involves tandem insertions through coaxial holes located in detached parts. As regards then remove-location-peg, provide-

temporary-support, and remove-temporary-support, we have to remark that if we assume to use a flexible jig or, as in our case, a two-handed robot system like the one we have developed ([Pettinaro & Malcolm 95b]), we can perform them by employing our pick-up module together with the two gripper commands grasp and ungrasp.

A special comment is required for the operation which Kondoleon labelled as crimp⁸. Such a parts joining process is nowadays no longer common due to the coming of cheap and reliable plastic materials during the eighties which were capable of achieving similar results more cheaply. Since their use made uneconomic the employment of metal crimping, plastic processing operations are today far more common than they were in seventies when Kondoleon's survey was carried out. As a consequence, we have to consider the fact that some more modules may need to be included in our basic set in order to make it more complete and capable of coping also with this kind of parts joining process. However, we have to observe that plastic fastenings, although being common, are strictly speaking manufacturing fixing tasks and not assembly ones. Besides, a behavioural module accomplishing them would not be so general-purpose to be used in other situations not requiring plastic processing. Moreover, such a module would always require a manipulator agent capable of grasping or attaching a dedicated tool in order to accomplish the operation. Taken into account that we restricted ourselves to deal just with simple assembly processes, we think that dedicated modules performing them should be added to our set only if we decide to deal also with parts fixing processes.

The last comment concerns Kondoleon's weld task. As emerged from his statistics (cf. subsection 2.3.1 on page 47), it is a very important operation largely used in industry, but, as said for metal crimping, it is actually a manufacturing fixing process and thus out of our scope which concerns just assembly operations. Therefore, although a more complete basic set should include some dedicated modules for dealing with it, we think that our set is not too limited because of its inability to cope with it.

The conclusion we draw out of the comparison is that our group of basic modules is capable of performing on our rather restricted assembly domain the great majority of the manufacturing assembly tasks considered by Kondoleon. However, we have to

⁸ Operation which consists of joining thin steel sheets by crimping them together.

stress that such a group of modules should be viewed as an embryo of a basic set and that some more modules may be required in order to deal with the other manufacturing processes considered by Kondoleon. In this regard, we argue that defining modules with a one-to-one mapping on his entire list of manufacturing tasks, or on other similar ones, is a viable proposition.

At this point, having compared our set of elementary modules with Kondoleon's basic manufacturing tasks, what we are left to discuss is how our set compares with the general-purpose modules isolated by Balch.

6.2.4.2 Comparison with Balch's General Purpose Modules

Based on grounds of our experimental results, we reached the conclusion back in section 6.1 on page 190 that it is possible to define a limited set of ergonomic and elementary modules to be used for programming assembly robots.

As discussed in subsection 2.3.2 on page 53, a similar conclusion was serendipitously reached by Balch and Malcolm in an earlier project which consisted in developing software routines general enough to handle large spatial misalignments and easy enough to be adapted to other similar assembly tasks.

The equipment he used to carry out such a project was slightly different from ours. He employed a simple RTX manipulator agent with a workstation as its controller as opposed to our compact and more accurate Adept system, and employed an expensive force-torque sensor as opposed to our relatively crude and simple touch sensor.

Nevertheless, the result he reached was that the program for accomplishing his test assembly was expressed in terms of a small group of 23 parameterized modules (cf. page 53). He noticed that these modules were quite general to be employed as off-the-shelf software packages and he argued that they could be used as predefined general-purpose procedures for many other assembly applications. In this regard, we have to observe that the way in which he synthesized such a group of modules was very different from ours. Since the beginning he aimed to reach generality in a bottom-up fashion by developing useful routines to apply to his unique experimental test assembly. In other words, his initial aim was not to define a set of basic software modules with which

to program assembly robots, but to retain as much generality as possible with which to handle variations in dimension and location. Since the beginning our approach has been instead much more focussed: we used the behavioural decompositions of our assemblies to look for convenient elementary modules general enough to be employed in as many situations as possible within our assembly domain. Anyhow, despite the difference of our initial aims, we both reached a similar conclusion: a set of general-purpose elementary assembly modules was easily within the scope of current robot technology and programming languages.

Comparing our basic set with Balch's one, we have to observe that most of his modules are at a very low level. It has been argued that, since behavioural modules are defined as having a predetermined purpose, they cannot be as general as his modules with disjunctive purposes. He maintained that his software modules, by having more than one specific purpose are instead at a lower and more general level than behavioural modules (if these are defined as having pre-defined single purposes).

We consider that the argument against multi-purpose behavioural modules is unnecessarily purist and restrictive, and based on our experimental results (cf. subsection 6.2.2 and 6.2.4 on page 194 and 203, respectively), we managed in fact to define a small group of elementary behavioural modules which may assume different purposes according to how they are wrapped into more complex units.

A close examination of his set shows an important difference compared with ours: its size. In this regard, it is logical to wonder whether the higher number of software modules he found with respect to ours is due to the kind of sensor he employed, which is far more sophisticated than ours, or to the lower level of modularization at which he worked. In this respect, it is true that replacing our sensor with his would change the implementation of many low-level submodules. However, our behavioral modules should always accomplish the same task on our assembly domain and return the same exit state codes whatever the implementation of the supporting submodules. This means that modules such as `stack` or `peg-in-hole`, for instance, should always stack parts or insert a peg into a hole, respectively, whatever their low-level submodules' implementation is. Therefore, even if we use other kinds of sensors, such as a camera, we would not need to add any new modules to our set to deal with them, because

this would simply affect the low-level implementations and not the addition of new members to our basic set. Thus, the higher number of modules found by Balch is basically due to the lower level of modularization at which he worked, and not to the more sophisticated sensor he used. Indeed, if he had synthesized his routines at a higher level, he would have probably reduced his set to the one we have found.

Observing more carefully his 23 modules and comparing them one by one with our basic set, we notice that under the hardware and software assumptions outlined in subsection 3.2.1.1 on page 70 and in subsection 3.2.4 on page 78, we can actually find for several of his modules a corresponding one in our set which simulates it. However, because of the lower level at which his software is based, many of them are redundant for us.

The list here below shows briefly how Balch's modules compare with our set.

Moves Modules, Check-Lock, Dab, Lift and Press can not be performed directly by our modules because they are too low level. We have to point out, though, that one of their most frequent uses is to implement pick-and-place operations which we perform with a combination of our pick-up and stack. Thus, although our modules are unable to simulate them singularly taken, we do not think that this is a disadvantage, because their use should be hidden anyhow in a task-level system.

Move Gripper, which is used for controlling the robot end-effector, may be easily performed by our low-level grasp and ungrasp modules.

Hopping-Follow and Hopping-Spiral-Search may be achieved within certain limits by our peg-in-hole which has the feature of search for hole by hopping built-in.

Sensor Routines for interacting with a force-torque sensor, namely *Get-Force*, *Get-Force-Vector*, *Check-Force*, and *Reset-Sensor* are in our case redundant given the simplicity of the sensor we are employing, whose interface is already built-in in most of our modules.

Position Routines are also redundant for us because we assume that

this kind of information is available at a very low-level which should be invisible to the end-user.

Relieve-Stress is redundant too, because our touch sensor triggers just if there are variation of forces. Thus, if a robot exerts a stress with one of its gripper fingers, then the reacting behaviour of our guarded-move, which most of our modules are based on, would trigger a signal allowing our manipulator to stop itself and hence the stress it causes.

Peg-in-Hole and *Put-Flat* can finally be easily performed by our peg-in-hole and stack modules, respectively.

All of these remarks are summarized here below in table 6.1 where we list which Balch's routines can be simulated with our higher-level modules.

Balch's Modules	Simulation with our Modules
<i>Move-Absolute, Move-Relative, Guarded-Move, Guarded-Move-Relative, Check-Lock, Lift, Dab, Press</i>	Too low-level, cannot be singularly simulated
<i>Hopping-Follow, Hopping-Spiral-Search</i>	peg-in-hole
<i>Move-Gripper</i>	grasp, ungrasp
<i>Get-Force, Get-Force-Vector, Check-Force, Reset-Sensor, Get-Position, Get-Position-Vector, Relieve-Stress</i>	Redundant
<i>Contact-Position, Follow, Spiral-Search</i>	Cannot be simulated
<i>Peg-in-Hole</i>	peg-in-hole
<i>Put-Flat</i>	stack

Table 6.1: Comparison between Balch's set and ours.

Observing the table listed above, we notice that the command *Contact-Position*, selected by Balch as one of the general purpose modules, is supposed to determine the location of a possible contact between a part and the table underneath which the force-torque sensor is located. Such a command as well as *follow* and *spiral-search* cannot be reformulated in terms of any of our modules because they rely on the possibility of having a continuous reading out of a sensor which we initially supposed not to have. However, we can approximate *Follow* and *Spiral-Search* with their corresponding hopping commands which, as pointed out above, can be performed by our peg-in-hole.

As regards *Contact-Position*, *Check-Lock*, *Lift*, *Dab*, *Press*, and all the *Moves Modules*, we have to observe that they are too low level from the end-user point of view who should not be concerned with such a degree of detail. Thus, as long as a task is successfully accomplished, he should not be aware of the low-level implementations which allow him to do it. Our set does not contemplate specific behavioural modules performing them because they deal with our assembly domain at a higher level of abstraction. Nevertheless, this should not be regarded as a weakness of the description power of our set because it does allow our manipulator agent to perform on our restricted assembly domain 80% of the most common manufacturing tasks (cf. subsection 6.2.4.1 on page 206).

Summarizing, we can say that our modules can simulate some elements of Balch's set but not those which directly depend on the kind of sensor employed, nor those which involve elementary motion of the manipulator arm. However, these particular ones should be invisible at our level of abstraction because they involve very technical details. Thus, the lack of dedicated modules to perform them should not be viewed as a limitation of the description power of our set which, as experiment showed, can reliably cope with our assembly domain.

6.3 Behavioural Robot Language

Summarizing the main achievements gathered by our experiments which we discussed in the previous two sections, we can say first that it is possible to define a set of behavioural modules with which to express the great majority of the assembly tasks on our assembly world and second that such a set should include: *peg-in-hole*, *pick-up*, *stack*, *screw*, *flip*, *insert-peg-with-spring-retainer*, *snapfit*, *grasp*, *ungrasp*. Using these modules directly or opportunely composing them together, we are actually able to represent many complex assembly tasks on our assembly domain.

At this point, given the aforementioned group of behavioural modules listed before, we have to outline a possible behavioural language which may be built on top of them. Incidentally, it is interesting to notice how different is our approach from that taken for the definition of knowledge-based languages (cf. [Gruver *et al.* 84], [Gini 87],

[vanAken & vanBrussel 88], [Rock 89], and [ElMaraghy & Rondeau 92]). The designers are in such a case mainly concerned with the definition of opportune control and data structures, with the definition of functions to handle the data, with the specification of commands to operate the robot manipulator, and finally with the specification of low-level sensor and signal interface. Our approach is instead limited in defining just the basic elements in order to compose our modules together without getting involved with low-level considerations about the hardware of the particular robot system used. In this regard it is worth briefly synthesizing the main features which a successful and widespread robot language such as VAL II has.

This language is very simple compared with the great majority of computer languages on the market, nonetheless it has many interesting characteristics which may be of interest to us. We give here below a brief description of the most relevant ones.

Robot Location is a data structure made of six components: the first three identify the Cartesian coordinates of the end-effector, and the other three the rotation about each axis, namely *yaw*, *pitch* and *roll*. Locations can be accessed by name and represented as a **transformation** which is a robot-independent representation of the position and orientation of the robot end-effector.

A location can also be defined as a precision point which is a representation in terms of the exact positions of the individual robot joints.

Numeric Values allow the programmer to deal with four data types: integer, real, logical, and ASCII. Values can be accessed by name and can be grouped in arrays. Numeric values and variables can also be composed together into expressions by means of four kinds of operators:

- mathematical operators,
- relational operators,
- logical operators, and
- binary operators.

String Values allow instead the programmer to define sequences of ASCII character enclosed in quotes. Each string can be accessed by name, which are preceded

by a \$ character in order to distinguish them from other data types. Strings can also be grouped together in arrays of strings and can be composed together with string variables into string expressions by means of a concatenation operator (“+”).

Monitor Commands allow the programmer to handle the manipulation of programs and data, to control the program execution and in general the system status. They can be classified into:

- program editing,
- program and data storage,
- program and data manipulation,
- program control,
- location definition,
- system control, and
- binary signals.

Program Instructions allow an end-user to program the robot to perform tasks. We can distinguish the following classes of commands:

- robot control which deals with the robot motion and with the hand, trajectory, and configuration control;
- program control which deals with the program control structures;
- input/output which deals with the user and disk file input/output, with external signals, and with peripheral device control;
- assignment which deals with the setting of program variables and system parameters.

Functions allow the programmer to make more sophisticated computations or tests and to make their results available within user-defined expressions. Among the many categories defined by the language we list the following functions:

- numerical functions (mathematical, robot control, robot location, constant values, input/output, and string manipulation),

- location functions (transformations and precision point),
- string functions.

Looking back at the characteristics of VAL II, we have to acknowledge that the availability of robot locations is very important also for us, indeed we may claim that it is fundamental for most of our modules. As regards numeric and string values, we think that they should also be included in our behavioural language because they can greatly simplify the composition of behaviours into more complex entities.

The operation of composition is a fundamental feature of a behavioural language based on our basic modules, thus it is important at this point to define clearly how to achieve it. To this end we introduced in subsection 4.1.2 on page 91 six operators: sequence, fork, select, and while-repeat, repeat-until, and for-to-repeat. By opportunely applying them to our set of modules, we can actually program many complex assembly tasks.

Disregarding sequence which does not require any format to be specified, since it is a mere list of modules run one after the other, we need now to define possible formats for the remaining ones.

fork This operation may be represented as a function which takes all the behaviours to be performed in parallel as parameters.

selection This operation may instead be modelled either on the usual if-then-else format, shared by many high-level computer language, or on the more powerful case-of format. Although the latter can always be reformulated in terms of opportune sequences of the former, it is better for a question of convenience to retain both of them.

while-repetition This operation may be represented with the while-do construct which subordinates the repetition to the truth of a predicate tested before each loop.

repetition-until This operation may be represented with the repeat-until structure which differently from the previous one subordinates the repetition to the falsity of a predicate tested after each loop.

for-to-repetition This operation may be finally represented with the for-to

construct which subordinates the repetition to a counter which ranges from a starting value to an end one, both specified at the beginning.

Figure 6.1 here below reports a diagram summarizing the discussion above.

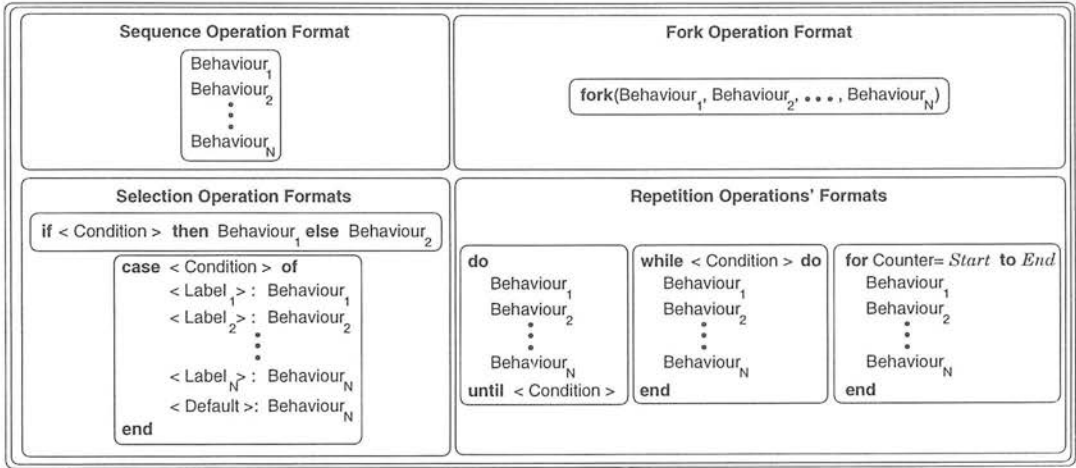


Figure 6.1: Proposed Formats for our Composition Operations.

At this point having proposed some formats for our operations, we need to talk about other two elements of a programming system: the variables and the constants. We know that any high-level programming language needs to provide them in order to be able to make any computations. Since we are outlining the definition of a possible language based on our set of basic modules, we think that such a language should allow the use of programming variables and the definition of constants. If this is not the case, the operations discussed before could not be even implemented.

Looking back again to the VAL II characteristics listed before, we think that the monitor commands available in VAL II are not really required by our language because they are more related to an operating system than to a robot language. Thus, we may assume to resort to the underlying system commands to perform them. As regards VAL II program instructions, we have to point out that they are at a very low-level and because of this it is very tedious to write programs with them. The modules of our set (cf. subsection 6.2.4 on page 204) allow us to program, instead, at a task-level in a more user-friendly fashion, and, as discussed in section 6.2 on page 192, we can actually accomplish with them 80% of the most common assembly tasks on our assembly

domain.

A final point to be analyzed is concerned with mathematical functions and operations. These are not essential features of a programming language, and indeed their presence is strongly dependent on which domain the language is designed for. In Computer Science in fact there are examples of programming languages, such as Prolog, wherein mathematical functions are not part of the language: they are just added for a question of convenience for the end-user. In pretty much the same way, considering that our programming domain is the class of assembly tasks which our set of modules can cope with, and considered that a language should be convenient to be used, we think that a behavioural language built on top of our set should at least provide the bare arithmetic operations. However, nowadays it is relatively very simple to implement higher mathematical functions such as logarithmic or trigonometric functions, and their availability becomes crucial when sophisticated sensors such as a 6-axis force sensor or vision are employed. Therefore, in order to avoid unnecessary limitations, we think that a behavioural language should provide, besides the simple arithmetics, a large set of predefined mathematical functions.

Location and string functions may instead be both more useful to us. The former because they allow complex location manipulation, such as calculation of Euclidean distance between points or composition of a location out of numerical components. The latter because they allow us to manipulate and process commands written in a more natural way closer to the end-user than to the robot, such as extractions of substrings out of a string or compositions of them. This in particular allows us to raise further the level of robot programming.

As a conclusion, we can say that a behavioural language built on top of our set should first provide the bare arithmetics together with an opportune set of high mathematical functions, and second include completely location and string functions.

At this point, we can summarize the main results achieved in this chapter.

6.4 Summary

As pointed out in section 6.1 on page 190, we observed that, even if it is always possible to find a large variety of assembly tasks, the range of those which most frequently occur in manufacturing industry is rather limited. In this respect, our research, although carried out on a small assembly domain, showed that it is possible to define programs which can be used for similar assemblies simply by instantiating them with opportune parameters characteristic of the particular test bed on which they are applied (cf. subsection 6.2.2 on page 194). Moreover, since these programs are defined in terms of software units (behavioural modules) which have disjunctive multiple purposes, the second main achievement gathered by our experiments was that we can define software modules which can have a larger applicability than the one for which they are initially designed. In this regard, we synthesized out of our experiments 9 software modules (peg-in-hole, stack, pick-up, grasp, ungrasp, insert-peg-with-spring-retainer, snapfit, screw, and flip), which are capable of coping with 80% of the assembly tasks on our restricted domain (cf. subsection 6.2.4 on page 203). However, the two aforementioned results should not be interpreted as a proof of the necessary existence of a set of general-purpose behavioural modules. Nevertheless, they do set experimental grounds supporting such a thesis because, although the few test beds examined were not meant to exhaust all the possible assembly tasks, they show that we do not require to define a new module for every slightly different task.

Comparing our set of 9 modules with Kondoleon's set of twelve manufacturing processes (cf. subsubsection 6.2.4.1 on page 206), we observed that some of them cannot be performed by any combinations of our group of modules because their scope is much larger than our assembly domain. Moreover, although some of his tasks such as force-fit or crimp might be simulated within certain limits by our modules, we think that a few more dedicated ones might be required in order to cope with all of them. Comparing then our set with the group of 23 software modules found by Balch, we noticed that our set was capable of simulating those which were at a higher level of abstraction but not the low level ones which directly controlled the robot arm or directly accessed sensory data (cf. subsubsection 6.2.4.2 on page 208). However, this should not be viewed as a weakness of our set, because we synthesized it so that to hide most of the technical

details underlying the specific workcell considered.

As a conclusion, our set is capable of programming on our assembly domain 80% of the most common manufacturing assembly tasks.

As a final point, we dedicated the last section of the chapter to outline the basic features which a behavioural language based on our set should have (cf. section 6.3 on page 212). In this regard, we proposed briefly first the kind of formats the composition operations should possess, then the use of programming variables and constants, and finally the functions which we should consider to include in such a language, namely at least location, string manipulation and mathematical functions. Once again, the point is not to make a specific proposal for a language, but to show, by means of an illustrative example, that it would not be difficult to do this.

Chapter 7

Conclusions and Further Work

This chapter, which represents the final stage of this thesis, is dedicated to summarizing the evaluation of the results obtained from our investigations (cf. chapter 6 on page 189), to synthesizing the scientific contributions brought in by our work, and finally to outlining possible extensions of this research. Thus, we organize the discussion in three sections: a summary of the analysis of the experimental results (section 7.1 on page 220), a discussion of what we have achieved (section 7.2 on page 222), and a discussion of how our project can be expanded (section 7.3 on page 224). A final section will draw conclusions about our research (section 7.4 on page 226).

7.1 Summary of the Results

The problem tackled by our project aimed at investigating the existence of elementary behavioural modules with which to program the most common assembly operations (cf. subsection 2.3.1 on page 52). In order to carry out such an investigation, we restricted ourselves to deal just with 80% of the assembly tasks, since the rest were either variations or specialized operations, and posed no extra problems, but, as is the way with exceptions, would have taken a great deal of further implementation to handle. Moreover, due to hardware constraints, we limited our assembly world just to polyhedrons with parallel surfaces and to cylinders.

The first important result gathered was the development of a simple form of guarded move which enabled our manipulator agent to adapt its motion to environment con-

straints by exploiting event contact information (cf. subsection 6.2.1 on page 193). In this regard, we have to say that the module implementing it does not perform any kind of collision avoidance planning and may be viewed as a crude extension of the common unguarded move available basically in one way or another in all robot systems. Thus, the interesting point of it is not what it can do on its own, but the fact that, despite the limited information it provides, it allows more complex capabilities to be erected on its top by opportunely combining it with low-level robot commands.

The second result obtained in our research consisted of the fact that we managed to write a program for achieving the assembly of two similarly shaped benchmarks and another program for achieving the assembly of a small group of similar electric torches (cf. subsection 6.2.2 on page 194). The observation drawn out of this result was that it is possible to define assembly programs which are capable of coping with groups of similar assemblies. However, the interesting point is not that we can write a program capable of coping with a selected number of test beds, but is the fact that we can do it in terms of behavioural units having disjunctive multiple purposes. This feature, which allowed some of these units to be employed for both benchmark and torch programs, is the key to generality and is what makes them valuable programming tools, because they assume a purpose, and hence achieve a particular goal, according to how they are composed together. In this regard, we have to observe that we achieved it by encoding the different purposes with a numerical code returned at the end of the execution of the module in the form of an exit state. It is worth pointing out that such a code does not represent a success or a failure but simply the outcome of the task. Such a characteristic makes a module more general, and indeed supports our idea that general-purpose ones resolving within certain limits classes of assembly tasks may exist.

The third result achieved by our investigation was the synthesis of a group of 9 elementary behavioural modules (flip, grasp, peg-in-hole, pick-up, insert-peg-with-spring-retainer, screw, snapfit, stack, ungrasp) which showed a characteristic of general applicability on our limited assembly domain (cf. subsection 6.2.3 on page 195). Such a group can be interpreted as an embryo of a basic set which is capable of coping with 80% of the most common manufacturing tasks (cf. diagram in figure 2.10 on page 53). We have to remark, though, that the modules of this set should not be viewed as final solutions,

because they are so far applicable just on a small domain. In this regard, we think that some of them may require further development in order to cope with a larger assembly world and with some more different task varieties. We think in this respect to add a few more modules to our set, however we expect based on the grounds of our experience that their number is limited.

Comparing our set with the group of the 12 most common manufacturing processes emerged by Kondoleon's survey, we realized that several of them can be reformulated in terms of the modules of our set. Indeed, we argued that it is possible to define modules with a one-to-one mapping with his list or with other similar ones (cf. page 208). However, we have to stress the point that this applies in our case only as far as our domain is concerned which is far smaller than that considered by Kondoleon. He included in fact also manufacturing fixing processes such as weld which our modules cannot cope with, since their scope is restricted to some assembly processes only. As regards other tasks such as force-fit, we showed that we can use some of our modules to simulate them within certain limits. Nevertheless, we expect, as mentioned before, a few more modules to be added to our set in order to let it cope with larger task domains.

Comparing then our set of 9 behavioural modules with the set of 23 general-purpose software modules found by Balch, we noticed that ours was smaller because it was synthesized at a higher level of abstraction. In this regard, we have to point out that our modules, although being able to simulate some of his, cannot directly perform the great majority of them because they are too low level. Nonetheless, our set proved to be as general as his on the same assembly domain but with the difference of dealing with the tasks at a higher level.

7.2 Scientific Contributions

As discussed in chapter 2 on page 13, one of the main reasons to which we ascribe the limited development of robotic systems in manufacturing assembly industry is due to the difficulties of robot programming. As showed there, many solutions have been proposed, but none of the robot-level or task-level languages developed, or even just

proposed, has been good enough to become a paradigm.

Our research tackled this problem within the behaviour-based assembly paradigm which may be an interesting alternative answer to the problem of reliably and conveniently programming assembly robots. However, in order to get this status, we have to establish whether we can actually define a general-purpose behaviour-based assembly language. To this end, the research we carried out in our thesis aimed at two main targets:

- making sure of the existence of a convenient basic set of behaviours with which to program 80% of the assembly tasks,
- building an embryo of a possible set of them.

Looking back at the results obtained by our project, we observe that the first achievement of our research is the fact that we can write an assembly program expressed in terms of behavioural modules for performing similar assemblies simply by instantiating it with appropriate input parameters. A second achievement is the implementation of a behavioural module with a single exit state variable encoding the different outcomes of the module. Such a value is returned at the end of the module execution and, according to how a module is composed with others, it enables a manipulator agent to achieve a certain goal instead of another. The third and final achievement is the definition on a rather restricted domain of a possible group of behavioural modules which may constitute an embryo of a basic set. In this regard, although their scope is quite limited and their reliability is globally about 98%, we think that their small number is a sign that a limited general-purpose set of modules can be defined on a larger domain. However, further development of our modules would be required to extend their range of application. Moreover, a few more may be needed in order to enlarge the range of tasks which such a set can cope with.

Comparing our work with the related research projects mentioned in subsection 2.3.3 on page 56, we have to point out that the results we have achieved are an advance of the current state of the art. In particular the absence of probabilistic models to estimate uncertainty or any models of the world (cf. [Krogh & Sanderson 86], [Archibald & Petriu 93], and [Nnaji 93]) reduces considerably the complexity of the

overall system. Moreover, the fact that the architecture within which our modules are designed to operate does not allow them to compete with each other for the control of a robot is better suited to the programming of assembly agents than competitive parallel schemes (cf. [Noreils & Chatila 89]). As regards the level of abstraction at which our modules are synthesized, we have to remark that it is closer to task level programming than to the machine one. Thus, our modules as opposed to [Nilsson & Nielsen 92] do actually facilitate on our rather restricted assembly domain the programming of assembly agents.

Summarizing, we can say that the scientific contributions we achieved in this thesis are:

- the possibility of using one interpreted program, parameterized at run-time, for performing similar assemblies, the architecture naturally tending to simplify and make significant in human terms the meaning of the parameters;
- the implementation of generality in behavioural modules by means of an exit state encoding the possible outcomes of the task performed by the module;
- the definition of an embryo of a basic set of elementary modules.

7.3 Further Work

The last point we need to discuss concerns possible extensions of our research. In this regard, the first thing we have to observe is the fact that the set of 9 modules synthesized out of our experimental work is applicable just to a very limited assembly world, namely polyhedrons with parallel faces and cylinders. This was mainly due to the rather crude end-effector (two-fingered gripper) with which we equipped our manipulator agent. Thus, we think that the first point which requires to be developed is a pick-up module capable of dealing with a more flexible end-effector (multi-fingered hand). In this respect, also other modules such as `grasp` and `ungrasp` require to be developed accordingly in order to cope with an enlarged assembly domain.

Another important point of our research which requires further development is reliability. As emerged from the experiments carried out in chapter 5 on page 101, the 9

modules of our set achieved globally at most 96%-98% reliability rate which is still low compared with 99.9% required by industrial standards. For the purpose of this thesis this is acceptable because what we have investigated concerned the development of a toolkit of general modules (cf. page 205). However, they need to be enhanced in order to be acceptable to industry and to be a viable alternative to specialized hard automation equipment. One way of doing it may be devising better strategies to perform them. A special remark in this respect concerns our peg-in-hole whose strategy let it deal just with round tandem peg-in-hole. However, we would considerably extend the applicability of such a module, if we could make it deal also with some form of orientation dependent insertions. In this regard, our strategy of thrust and correct may still be used as an outline in order to implement an enhanced peg-in-hole which is capable of dealing also with limited misaligned prismatic insertions. Indeed, since the current implementation showed encouraging experimental results, we may use the current strategy alongside another specialized in orientation dependent insertions. This can be achieved by redesigning our peg-in-hole so that to accept an extra input parameter acting as a selector for choosing the appropriate strategy according to the kind of peg insertion required. Anyhow, it is important to point out that such a selector would always require to be instantiated by a planner or a human operator.

Another point which we expect to be further developed concerns the possible addition of other behavioural module to our basic set. In this regard, we observed that our group of modules cope with 80% of the most common assembly tasks on our rather small assembly world. However, expanding both task and world domains may cause some more modules to be added to our group. As pointed out earlier, though, we do expect at our level of abstraction that their number would be quite small, possibly within the range of 10-15 more.

Another line of further development of our research regards the definition and implementation of a behavioural language based on the modules of our basic set. In this respect, we gave an outline of what features we would expect such a language to have (cf. section 6.3 on page 212). Nevertheless, we think that a more formal definition of its programming elements is required. In this regard, an interesting extension would be the development of a user-friendly computer graphical interface (cf. [Balch 92b]).

Such a tool would greatly simplify the composition of our modules in the same way as a CAD system (to which it is closely related) facilitates the design of new products. This interface might allow an end-user with no particular knowledge or experience in the field to program a robot to make a reliable assembly. This particular line of research, though, is not straightforward, and indeed it may be considered as a long term project. However, if the behavioural language which we can build on top of our set of basic modules has such a front-end interface, the activity of programming a manipulator agent would be enormously facilitated, thus helping to extend the use of robots to a much larger public.

7.4 Conclusions

As recalled at the beginning of this chapter in section 7.1 on page 220, the problem we tackled in this thesis consisted in investigating the existence of a limited set of elementary behavioural modules with which to program 80% of the most common assembly operations.

This focussed attention on the behavioural decompositions of three significant assemblies selected because they involved very common assembly processes (cf. survey in subsection 2.3.1 on page 47). The experimental results achieved with this research, although not representing a theoretical proof and being valid on a rather restricted assembly world, showed that it was possible to define convenient basic modules which have general applicability on a certain domain. In this respect, we managed to synthesize 9 elementary modules showing such a characteristic at a task-level on a restricted assembly world:

- `flip`
- `grasp`
- `insert-peg-with-spring-retainer`
- `snapfit`
- `peg-in-hole`

- pick-up
- screw
- stack
- ungrasp

The final conclusion we draw is that this particular group of 9 basic behavioural modules is both representative and ergonomic enough on such a domain to be easily used either by human operators or automatic planners for programming 80% of the most common manufacturing assembly operations. However, it should be viewed as an embryo of a more complete basic set on top of which could be erected a high-level behavioural language loosely modelled on a successful robot language such as VAL II (cf. section 6.3 on page 212).

Appendix A

Adept Characteristics

The Adept robot is a SCARA five-axis robotic system designed mainly for production applications in light material handling, parts fitting, assembly and packaging. The Mean Time Between Failures is at least 2000 hours.

Its main characteristics are:

- user-friendly programming language and control system (VAL II),
- extensive process control capabilities,
- minimum cycle times for material handling operations,
- high accuracy, repeatability and resolution,
- real-time sensory control,
- large working envelope,
- expandable control system,
- library of available application programs,
- all DC servo-motor operation, and
- fail-safe brakes.

The particular Adept model here described has one more degree of freedom than its basic counterpart. The five joints are:

Joint 1 rotation about the vertical axis at the shoulder level (max. 300°),

Joint 2 rotation about the vertical axis at the elbow level (max. 294°),

Joint 3 linear displacement at the end of the arm (max. $195mm$),

Joint 4 rotation about the vertical axis at the wrist level (max. 554°),

Joint 5 rotation about one of the horizontal axis (max. 180°).

A Adept Characteristics

The first 2 joints are responsible for the horizontal positioning of the manipulator, the third joint for lifting and placing down objects, the fourth for orienting parts, and the fifth for rotating objects. In figure A.1 we report a diagram of the above mentioned degree of freedom.

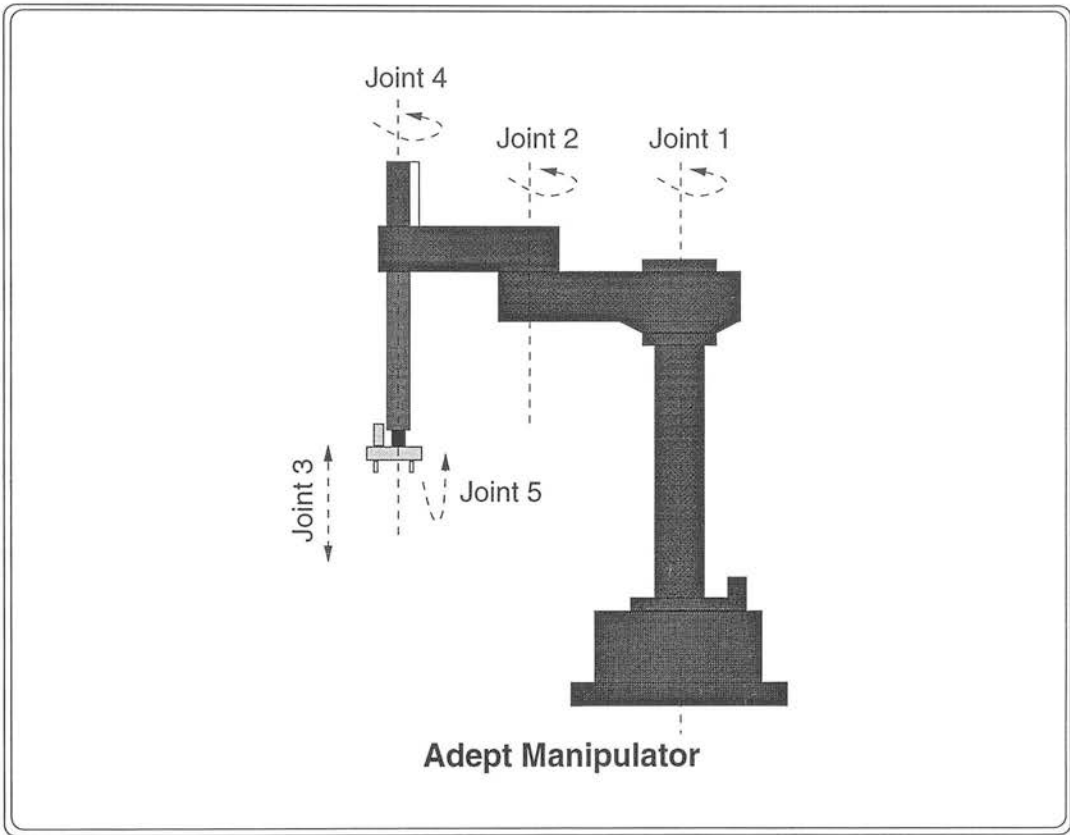


Figure A.1: AdeptRobot Joints.

The robot is equipped with 24 parallel ports available: 16 inputs (from address 1001 to 1016) and 8 outputs (from address 17 to 24). It is also equipped with 6 serial ports of which 3 are currently used: the terminal (ASCII TERMINAL), the Tesla server¹ (USER 1), and a PC (USER 2).

¹ This is the server to which the machines of the lab are connected.

Appendix B

Robot Electric Gripper

The two-fingered gripper reported here below was designed by Cameron and Wyse of the departmental mechanical workshop. It is made of two main parts (cf. figure B.1): a D.C. motor and a gearing system to move the fingers.

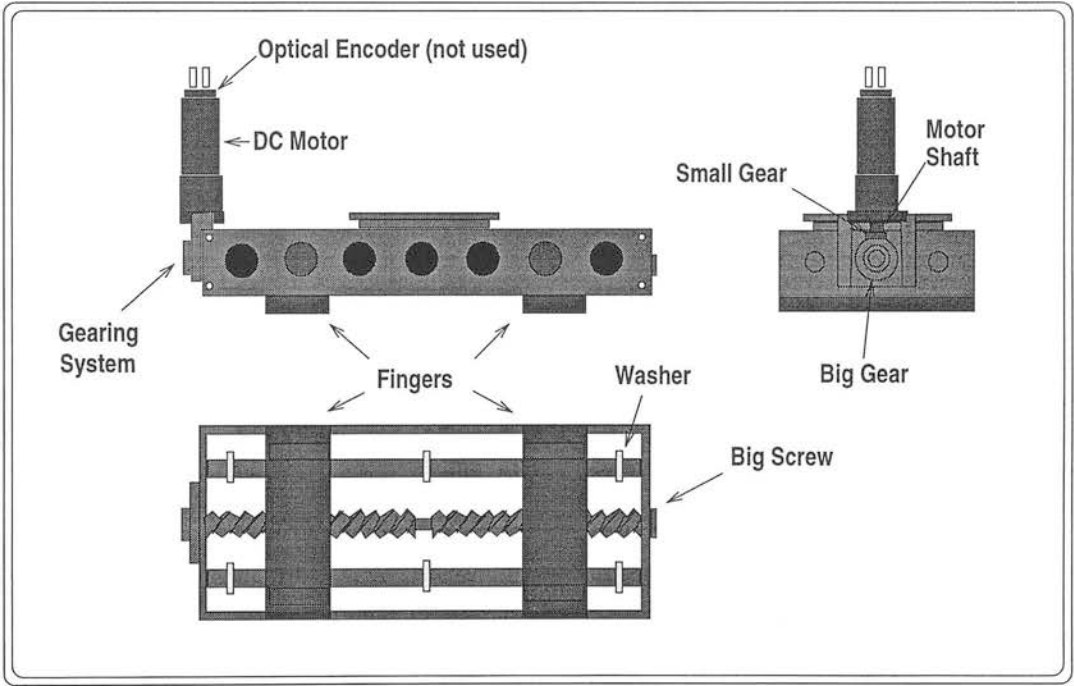


Figure B.1: DAI Electric Gripper.

The motor used is an escapTM 23D21-210E incorporating an optical encoder on board, however this last was not used. The technical characteristics of such a motor are listed in table B.1. The motor, as we can see from the table, has a considerable range of operating voltages. Experimental tests showed that increasing the voltage applied to the motor increases the final speed of the fingers. This is not an unexpected behaviour, let us see why. Since the back EMF of the motor is proportional to its speed, supplying

Measuring Voltage	V	24	
No-load speed	rpm	6600	
Stall torque	mNm	23	
	oz-in	3.3	
Power output	W	4	
Av. no-load current	mA	8	
Typical starting voltage	V	0.25	
Max. continuous current	A	0.39	
Max. recommended speed	rpm	8000	
Max. angular acceleration	10^3 rad/s^2	91	
Back-EMF constant	V/1000 rpm	3.6	
Rotor inductance	mH	1.7	
Motor regulation R/k^2	10^3 Nms	30	
Terminal resistance	Ω	36	
Torque constant	mNm/A	34.5	
	oz-in/A	4.89	
Rotor inertia	$\text{kgm}^2 \cdot 10^{-7}$	3.7	
Mechanical time constant	ms	11	
Thermal time constant	rotor	s	8
	stator	s	580
Thermal resistance	rotor-body	$^{\circ}\text{C/W}$	5
	body-ambient	$^{\circ}\text{C/W}$	12

Table B.1: Motor Characteristics.

the motor with a fixed voltage in effect produces a proportional control system which increases or decreases the torque to correct errors in the final speed.

Beside controlling the speed, though, the voltage applied to the motor has another important implication: it determines the strength with which the fingers grasp an object. The torque of the motor is proportional to the current. When the motor is stopped there is no back EMF and the voltage produces a proportional current and thus a proportional torque. Of course, the higher the voltage, the higher the grasping force. Since we have to choose an operating voltage, it is important to make a good trade between a firm grasping of objects and a low gear shock for sudden stops. By experimenting with different voltages, we reached the conclusion that a reasonable value should be not less than 20V. With such a power applied to the motor the total closing time from the position of fingers completely opened is around 4 sec¹.

The two parallel fingers are driven by means of a gearing system made of a small gear (32 teeth) and a big gear (64 teeth), see figure B.1. The smaller one is mounted onto the shaft of the electrical motor, whilst the other onto a big screw whose rotation makes the two fingers move (opening or closing).

The big and small gears and the big screw have diameters of 25mm, 13mm and 9.5mm respectively.

Because of the number of teeth of each gear, two turns of the smaller one make one turn of the bigger one. Every turn of the big gear, then, makes the fingers move of 3mm.

¹ The same applies for the opposite situation, i.e. from fingers completely closed to fingers completely opened.

B Robot Electric Gripper

Every time the fingers grasp an object, the gears are subjected to a shock caused by the impact which gets stronger as the voltage applied to the motor increases. In order to reduce such a shock, 6 washers are added to the gripper. They are placed in such a way that the fingers cannot crush against each other or against the screw ends (see figure B.1).

Appendix C

Two-Handed Robot System

This appendix is a summary of the two-handed robot system developed by the author in order to carry out the research reported in this thesis. The reader interested in the technical details may look in [Pettinaro & Malcolm 95b] for the full description, here below we give just a summary of its architecture.

The system is basically composed of three mechanical parts: right end-effector, left end-effector, and left wrist. Since each of them is driven directly by the robot controller, we may regard the three of them as independent subsystems. Thus, we identify:

- robot/right-gripper¹ subsystem,
- robot/left-gripper subsystem, and
- robot/left-wrist subsystem.

They are all similar to each other, indeed the first two ones are exact copies of each other, thus what we say for one of them is completely applicable to the others. Nevertheless, let us split their discussion in two different subsections: one for the two robot/two-grippers subsystems and one for the robot/left-wrist subsystem.

C.1 Robot/Two-Grippers Subsystems

A full description of these subsystems may be found in [Pettinaro & Malcolm 94], here we will limit ourselves to summarize the most salient characteristics.

The two main tasks that the gripper has to accomplish are driving the grippers' motors and reading the gap between the fingers.

In order to drive the motor of one gripper, the robot controller requires two signals:

- one to *activate/inactivate* the power to the motor, and

¹ From now on we will use the words *hand* and *gripper* as synonyms.

- one to *reverse* the motion of the fingers².

Because we have two grippers and because the robot used, which is an Adept, has got 8 parallel binary output lines available, it seems reasonable to use some of them to perform the driving. Indeed, if we assume that only one subsystem may be active at one time, we may use for both subsystems the same line to *activate/inactivate* the power and the same line to *reverse* the polarity.

In order then to distinguish between the two grippers, we may use two extra output lines. A diagram of the two subsystems is given in figure C.1.

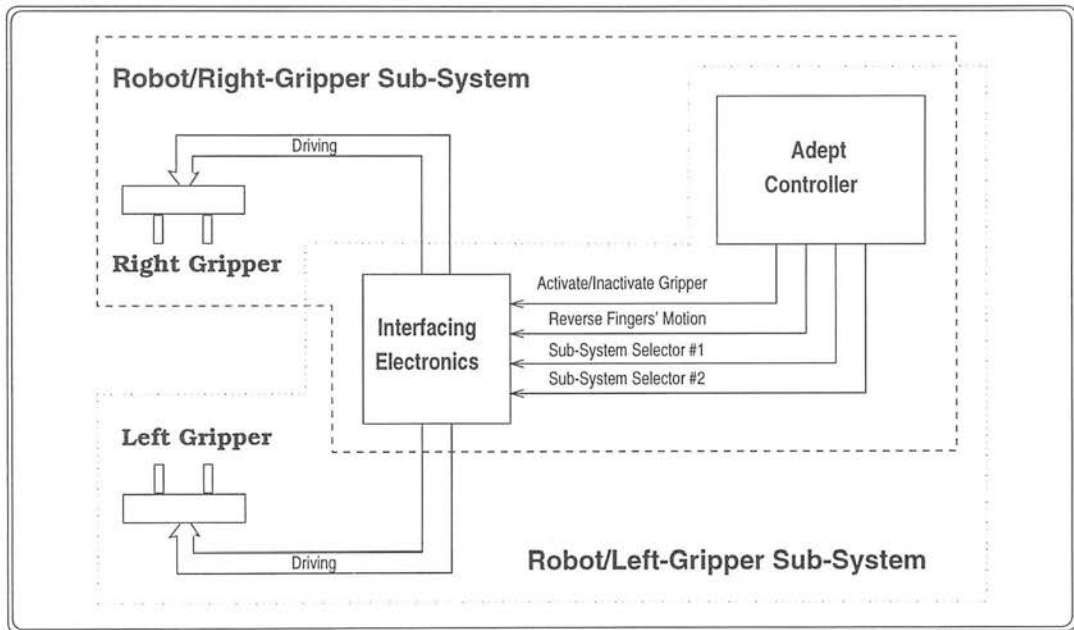


Figure C.1: Robot/Two-Grippers Sub-Systems.

The kind of gripper which we are dealing with is equipped with an electric motor geared onto a screw. Its spin direction makes the fingers mounted on the gripper slide towards each other or far away from each other. With this sort of mechanics there are basically two ways for reading the position of the fingers, and therefore their gap:

- using an optical shaft encoder, or
- using a potentiometer.

In the first case the encoder may be geared onto the motor and may code the gap by counting the revolutions of the motors. In the second case the potentiometer may be either geared onto the motor as before, or mounted directly onto the fingers: whatever way is chosen the gap is then converted into a voltage. For reasons explained in [Pettinaro & Malcolm 94] we chose a linear potentiometer attached to the fingers.

² This may be accomplished by reversing the polarity of the power to the motor.

Because a potentiometer is an analogue device, the gap is coded into an analogue voltage within a certain range. However, this implies that in order to read the gap we need another device on the other end to convert the analog signal into a digital one. The circuit capable of such an operation is an analogue-to-digital converter (A/D converter).

It is worth to mention at this point that our Adept controller is equipped with five serial ports (RS-232C) and 16 parallel binary input lines (cf. appendix A on page 228). Two of the serial ports, though, are already engaged: one with the terminal communication and one with the robotics lab server (Tesla). As regards the parallel input lines, four of them are used to monitor the touch sensors mounted onto the right and left grippers, whereas the other 12 are left free to the users.

In order to leave the maximum number of binary lines free for future developments, we decided to connect the converter to the robot controller by means of one of the three remaining serial lines available. To build this connection we used an AmstradPC equipped with an A/D converter and a serial port. In this case the PC acts as a pure slave device whose only task is to keep continuously polling the different potentiometers' lines and sending the corresponding readings to the robot controller. In order to do so, we can choose between basically two solutions:

- either reading one line at one time, sending immediately its value to the controller and then switching to another line,
- or keeping reading one line until the robot tells to switch to another.

The first solution implies the definition of a synchronous communication, and hence of a protocol. Since the delays caused by the use of a protocol may be too big to make a fast reliable reading of the fingers' position, we opted for the second solution. However, this last requires a way to understand the different values appearing in input at the serial port, and also a way to tell the PC to switch the readings to another line. In order to do so, we need to provide extra lines from the robot to the PC. Indeed,

Line #2	Line #1	Result
Off	Off	Keep reading current line
Off	On	Reading Right Gripper Pot.
On	On	Reading Left Gripper Pot.

Table C.1: Slide Potentiometers' Codes.

since we have two potentiometers, we need two lines to discriminate between them (cf. table C.1 for the discrimination codes). Thus, the PC still sends data through the serial line asynchronously, because the robot controller tells directly which line has to be monitored. Figure C.2 shows a diagram of the whole fingers' gap reading sub-system.

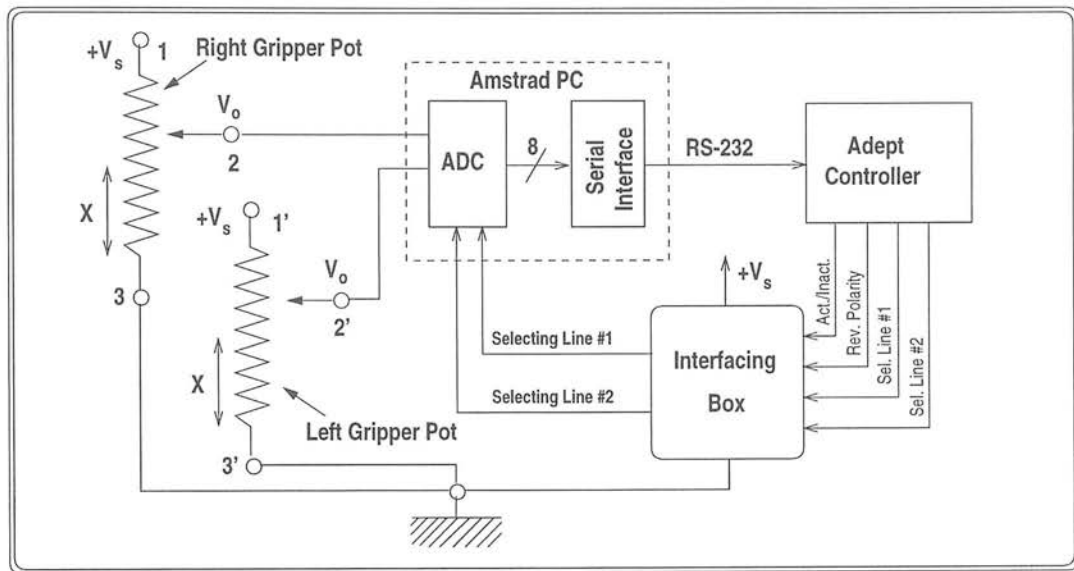


Figure C.2: Fingers' Gap Reading Subsystem.

C.2 Robot/Left-Wrist Subsystem

The architecture of this subsystem is similar to the grippers' one. However, the operations which it performs are slightly different:

- driving the wrist to spin for at most one turn clockwise or one anti-clockwise, and
- reading the angle about which the wrist has actually rotated with respect to a central position.

As in the case of the grippers' driving, we need one signal to *activate/inactivate* the power to the motor, and one to reverse the polarity. The way in which this subsystem works is exactly analogous to the grippers' one. In order to drive the two grippers' and turntable motors, we need to feed into an interfacing circuitry one binary output line to power the motors, one to reverse the polarity (cf. figure C.1). However, in order to discriminate among the three devices, we need also two extra lines: selector #1 and #2 which enables to select the appropriate device to be activated (*right* or *left* gripper, or turntable). Given the close analogy with the grippers' driving subsystems, we can easily draw diagram of the architecture as reported in figure C.3.

The mechanics of the left wrist is very different from the grippers' one, yet the way in which we chose to read its rotation angles about the vertical axis is very similar to that to read the fingers' gap of the gripper. The only difference consists just in the fact that this time we are dealing with a rotary potentiometer capable of many turns. In order to limit the complexity of the software which will accomplish the readings, we allow the wrist to rotate just one turn clockwise or one anti-clockwise with respect from a starting position. The rotation angle about the z-axis may therefore be computed

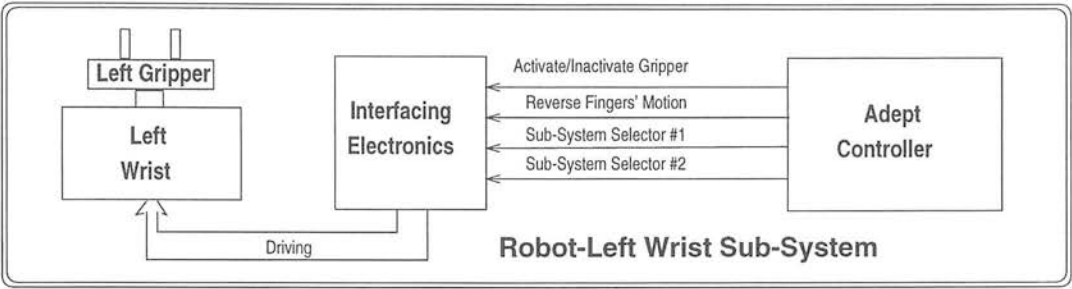


Figure C.3: Left Wrist Driving Subsystem.

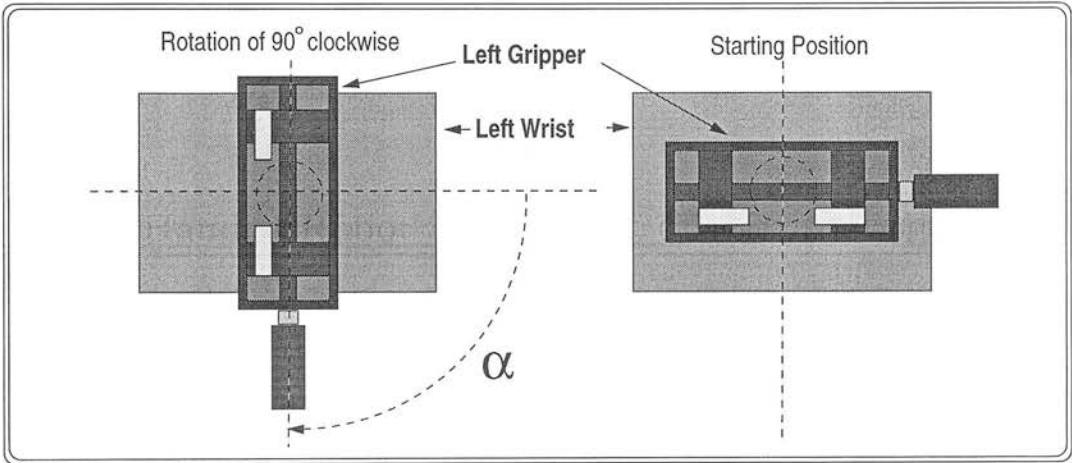


Figure C.4: Example of rotation anti-clockwise.

with respect to this last (cf. figure C.4). Now since from the Adeptcontroller point of

Line #2	Line#1	Result
Off	Off	Reading Current Line
Off	On	Reading Right Gripper Slide Pot.
On	Off	Reading Rotating Pot.
On	On	Reading Left Gripper Slide Pot.

Table C.2: Potentiometers' Discrimination Codes.

view the left wrist pot is just another device to be read, we can use the same reading system used for the grippers' pots: two parallel lines from the interfacing circuit to notify the PC which potentiometer line has to be read and one serial line to send the corresponding pot reading to the Adept. In order to realize the pot discrimination, we can extend the codes presented in table C.1 as shown in table C.2. Because of the high similarities between the pot reading of the gripper and of the left wrist, we can draw the diagram of the architecture of this subsystem as reported in figure C.5.

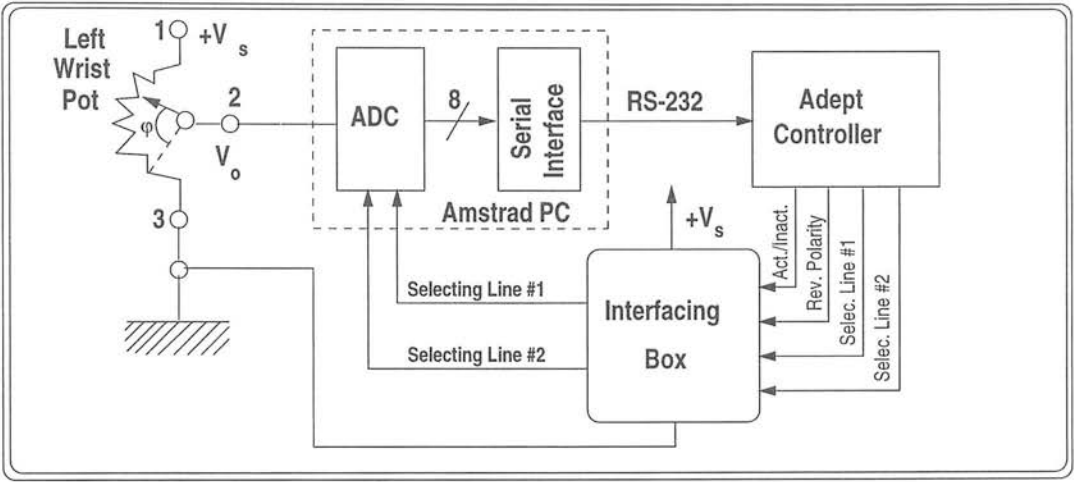


Figure C.5: Wrist Potentiometer Reading Subsystem.

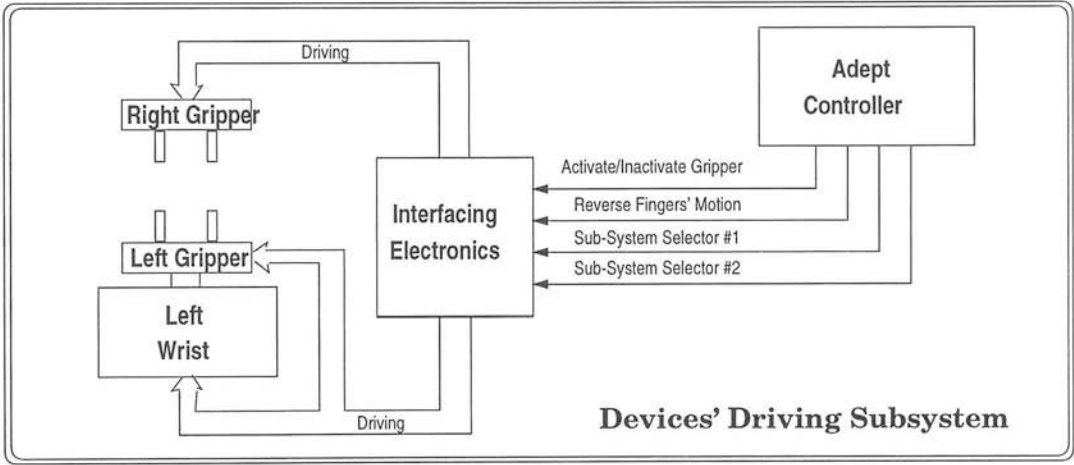


Figure C.6: Grippers and wrist driving sub-system.

At this point merging the devices' driving and potentiometers' reading subsystems (cf. figure C.6 and C.7 respectively), we obtain the complete architecture of the two-handed robotic system which we report in figure C.8.

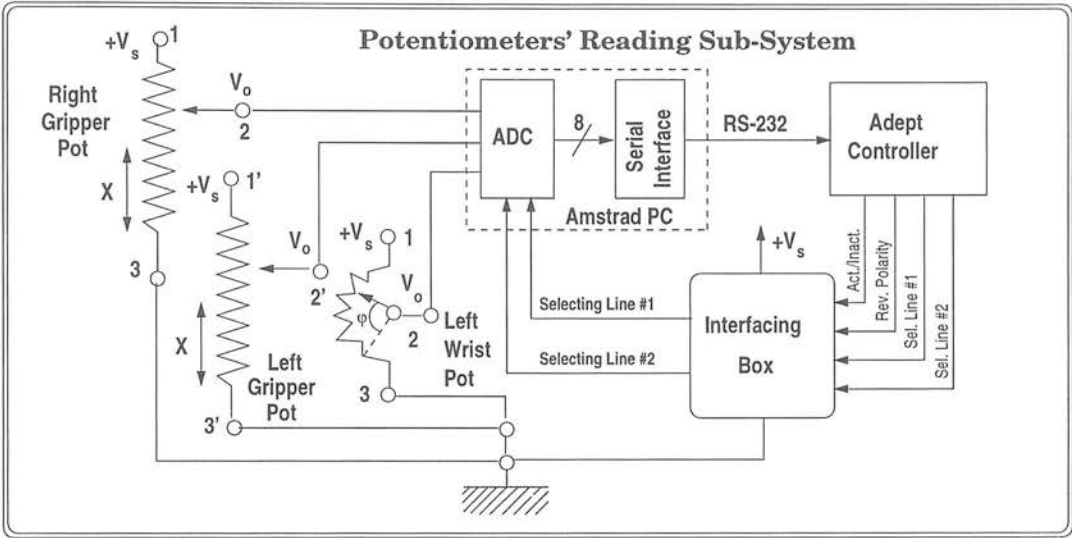


Figure C.7: Potentiometers' reading sub-system.

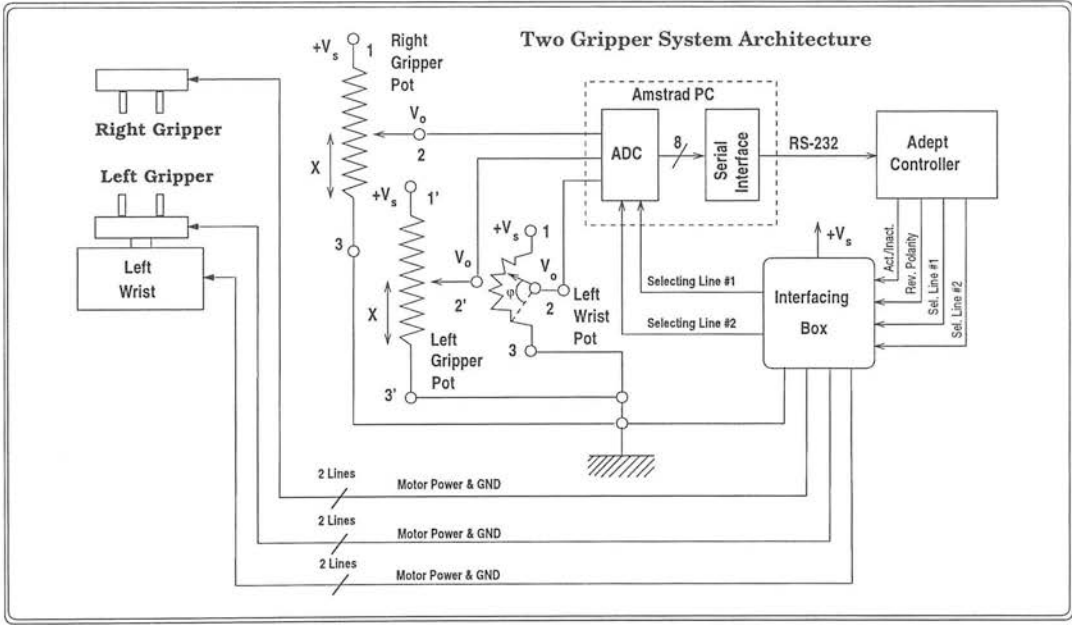


Figure C.8: Two-Handed System Architecture.

Index

A

Adept, 70, 105, 110, 192
Assembly operations, 47, 66
 current, 49
 Kondoleon, A. S., 47
 statistic, 52, 73, 78

B

Balch, P., 53, 63, 66, 98, 100, 189–191, 207–211
 general-purpose modules, 53, 67
Behaviour, 39, 55, 67, 84–86
 basic, 97
 Concept, 84
 definition, 85
 purpose, 39, 55, 85, 86
 scope, 85
Behaviour-based assembly, 67, 82
 definition, 87
Behaviour-task relationship, 67
Behavioural module, 149
 properties, 86
Behavioural modules, 86
 basic, 97
 basic set, 194
 complexity, 95
 fork, 92, 105
 hierarchical level, 96
 hierarchy, 95
 most primitive, 95, 193
 operators, 92
 repetition, 93
 selection, 92
 sequence of, 92
Brooks, R. A., 39
Bruynickx, H., 195
Byrne, C., 63, 69

C

Cartesian robot, 70
Chongstitvatana, P., 40

vision servoing algorithm, 41

Church, A., 46

D

Deacon, G. E., 43
 pushing motion, 43

E

Electric gripper, 71, 79
 table, 78
End-effector, 71
Exit state, 89, 91

F

Fine motion, 102
flip module, 180, 225
Full assembly benchmark
 exit states, 150

G

grasp module, 225
Grasping, 139
Gross motion, 102
Guarded move, 66, 82, 102–115, 160, 164, 170, 184, 190, 192–193, 219
 exit state, 106

H

Handey, 29–33, 42
Hole search, 123
 outcomes, 120
 spiral, 126
 zigzag, 125

I

insert-peg-with-spring-retainer module, 225

J

Joint interpolated motions, 103

K

Kim, T., 80, 83

Kondoleon, A. S., 47, 191, 205–207, 217
 statistic, 48

L

Latombe, J. C., 102
 Left gripper, 79, 177

M

Malcolm, C. A., 39, 53, 85, 86
 Matarić, M. J., 86, 97, 98
 McFarland, D., 85
 Motion control, 104

N

Nnaji, B. O., 58, 189

O

Obstacle avoidance, 104

P

Partial assembly benchmark, 135
 exit states, 138
 Peg in hole, 116–135, 154, 157
 decomposition, 117
 exit states, 120
 inputs, 121
 strategy, 119
 tandem, 8, 118
 Peg in hole module, 137
 Peg on hole, 121–127
 peg-in-hole module, 225
 Pick-and-place operation, 173
 exit states, 174–175
 inputs, 174
 pick-up module, 225
 Pick-Up module, 139
 inputs, 141
 Project, 68
 aims, 72
 description, 69
 objectives, 72

R

RALPH, 58–60, 63, 189, 197
 RAPT, 18
 Remote center compliance, 117, 167
 Right gripper, 79

S

SCARA robot configuration, 70, 102, 192

screw module, 165–172, 226
 algorithm, 169
 exit states, 170
 inputs, 170
 strategy, 168–170

Sensors, 104

Slide potentiometer, 71

Smithers, T., 39

snapfit module, 225

stack module, 142, 226

Steels, L., 85, 99

Straight line motion, 103

STRASS assembly, 156–165
 retaining, 157
 snapfit, 159
 strategy, 158

T

Tandem hole, 146
 Teach pendant, 144, 151, 162, 182
 Topic problem, 66
 definition, 68
 Torch assembly, 165–185
 inputs, 178
 parts, 165
 plan, 172
 Torch subassemblies, 173
 exit states, 176
 inputs, 175
 Touch sensor, 88, 105, 108, 113, 120,
 123, 129, 136, 139, 140, 147,
 154, 158, 161, 163, 169, 172,
 192
 Turing machine, 46
 universal, 46
 Turing, A. M., 7, 46
 thesis, 7, 46, 82
 Turntable, 79
 Two-gripped robot system, 169

U

ungrasp module, 226

V

VAL II, 70, 87, 90, 97, 102, 108, 111–
 113, 191, 212, 226

W

Whole arm collision avoidance, 104

work-cell, 70, 78

Y

Young, R., 98

Bibliography

- [Akella & Cutkosky 89] Prasad Akella and Mark Cutkosky. Manipulating with Soft Fingers: Modeling Contacts and Dynamics. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 764–769, Resort Scottsdale, Arizona, U.S.A., 14th–19th May 1989.
- [Archibald & Petriu 93] Colin Archibald and Emil Petriu. Skill-Oriented Robot Programming. In *Proceedings of the International Conference of Intelligent Autonomous Systems*, Pittsburgh, Pennsylvania, U.S.A., 15th–18th February 1993.
- [Armstrong 95] Edwin Armstrong. Using Virtual Sensors when Doing Compliant Motion. UWA-DCS-95 n° 9, Centre for Intelligent Systems, Department of Computer Science, University of Wales, Aberystwyth, 27th November 1995.
- [Aspragathos 91] N. A. Aspragathos. Assembly Strategies for Parts with a Plane of Symmetry. *Robotica*, 9:189–195, 1991.
- [Balch 92a] Peter Balch. Force Sensing in an Industrial Assembly. D.A.I. Working Paper n° 237, Department of Artificial Intelligence, University of Edinburgh, 1992.
- [Balch 92b] Peter Balch. A Software Architecture for Robot Assembly. D.A.I. Working Paper n° 235, Department of Artificial Intelligence, University of Edinburgh, 1992.
- [Berry *et al.* 87] Gerard Berry, Philippe Couronne, and Georges Gonthier. Synchronous Programming of Reactive Systems: an Introduction to *estere1*. Rapport de Recherche INRIA n° 647, Institut national de recherche en informatique et en automatique, 1987.
- [Bicchi *et al.* 93] Antonio Bicchi, Kenneth J. Salisbury, and David L. Brock. Contact Sensing from Force Measurement. *Journal of Robotics Research*, 12(3):249–262, 1993.

- [Blake & Taylor 93] Andrew Blake and Michael Taylor. Planning Planar Grasps of Smooth Contours. In *Proceedings of the IEEE International Conference on Robotics and Automation*, volume 2, pages 834–839, Atlanta, Georgia, U.S.A., 2nd-6th May 1993.
- [Bland 86] C. J. Bland. Peg-Hole Assembly: a Literature Survey. Cardiff Technical Note ARC n° 10, Department of Mechanical and Manufacturing Systems Engineering, University of Wales Institute of Science and Technology, January 1986.
- [Böhm & Jacopini 66] C. Böhm and G. Jacopini. Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules. *Communications of the ACM*, 9(5):366–371, May 1966.
- [Bonarini 94] Andrea Bonarini. Some Methodological Issues about Designing Autonomous Agents which Learn their Behaviours: the ELF Experience. In R. Trappl, editor, *Proceedings of the 12th European Meeting of Cybernetics and Systems Research*, pages 1435–1442, Singapore, 1994. World Scientific Publishing.
- [Borovac et al. 94] Branislav Borovac, Milan Nicolić, and Laszlo Nagy. A New Type of Force Sensor for Contact Tasks. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1791–1796, San Diego, California, U.S.A., 8th-13th May 1994.
- [Brooks & Stein 93] Rodney Allen Brooks and Lynn Andrea Stein. Building Brains for bodies. A.I. Memo 1439, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, August 1993.
- [Brooks 86a] Rodney Allen Brooks. Achieving Artificial Intelligence through Building Robots. A.I. Memo 899, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, May 1986.
- [Brooks 86b] Rodney Allen Brooks. A Robust Layered Control System for a Mobile Robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, April 1986. Published also as M.I.T. A.I. Memo 864.
- [Bruyninckx et al. 95] H. Bruyninckx, S. Dutré, and J. de Schutter. Peg-on-Hole: a Model Based Solution to Peg and Hole Alignment. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1919–1924, Nagoya, Aichi, Japan, 21st-27th May 1995.

- [Byrne & Hopkins 91] Carlton B. Byrne and S. H. Hopkins. An Expert System for the Identification of Assembly Tasks. In *Eurotech 91*, Birmingham, England, U.K., 1991.
- [Byrne 87] Carlton Byrne. A Literature Survey of Classification Systems for Assembly Processes. Cardiff Technical Note ARC n° 17, Department of Mechanical and Manufacturing Systems Engineering, University of Wales Institute of Science and Technology, October 1987.
- [Byrne 89] Carlton Byrne. A Survey of Mechanical and Electromechanical Assembly Tasks. Cardiff Technical Note ARC n° 32, Department of Mechanical and Manufacturing Systems Engineering, University of Wales Institute of Science and Technology, June 1989.
- [Caine *et al.* 89] Michael E. Caine, Tomás Lozano-Pérez, and Warren P. Seering. Assembly Strategies for Chamferless Parts. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 472–477, Resort Scottsdale, Arizona, U.S.A., 14th–19th May 1989.
- [Canny 87] John F. Canny. *The Complexity of Robot Motion Planning*. The M.I.T. Press, Cambridge, Massachusetts, U.S.A., 1st edition, 1987.
- [Caselli *et al.* 93] Stefano Caselli, Eugenio Faldella, Bruno Fringuelli, and Francesco Zanichelli. A Hybrid System for Knowledge-Based Synthesis of Robot Grasps. In *Proceedings of IROS '93 – the IEEE Conference on Intelligent Robots and Systems*, Yokohama, Japan, 1993.
- [Chongstitvatana & Conkie 92a] Prabhas Chongstitvatana and Alistair Conkie. Active Mobile Stereo Vision for Robotic Assembly. In *Proceedings of the 23rd International Symposium on Industrial Robots*, Barcelona, Spain, 6th–9th October 1992.
- [Chongstitvatana & Conkie 92b] Prabhas Chongstitvatana and Alistair Conkie. Behaviour Based Assembly Experiments Using Vision Sensing. In A. Colin and E. Emil, editors, *Advances in Machine Vision*, pages 329–342. World Scientific Press, Singapore, 1992. Published also as Edinburgh University DAI Research Paper n°466.
- [Chongstitvatana 92] Prabhas Chongstitvatana. *The Design and Implementation of Vision-Based Behavioural Modules for a Robotic Assembly System*. Unpublished PhD thesis,

- Department of Artificial Intelligence, University of Edinburgh, 1992.
- [Clegg *et al.* 93] A. C. Clegg, A. W. Quinn, M. W. Dunnigan, and D. M. Lane. Practical On-Line Motion Planning and Dynamic Control for Robotic Manipulators. In *Proceedings of the IEE Colloquium on Advances in practical Robot Controllers*, London, U.K., 26th November 1993.
- [Conkie & Chongstitvatana 90] Alistair Conkie and Prabhas Chongstitvatana. An Uncalibrated Stereo Visual Servo-System. In *Proceedings of the British Machine Vision Conference*, pages 277–280, Oxford, England, U.K., 1990. Published also as Edinburgh University DAI Research Paper n°475.
- [Coste-Manière *et al.* 92] Eve Coste-Manière, Bernard Espiau, and Eric Ruten. A Task-Level Robot Programming Language and its Reactive Execution. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 2751–2756, Nice, France, 12th–14th May 1992.
- [Craig 89] John J. Craig. *Introduction to Robotics: Mechanics and Control*. Electrical and Computer Engineering: Control Engineering. Addison-Wesley, 2nd edition, 1989.
- [Deacon & Malcolm 94] Graham E. Deacon and Chris Malcolm. A Robot System Designed for Task-Level Assembly. In *International Workshop on Advanced Robotics and Intelligent Machines*, Salford, England, U.K., 28th–29th March 1994.
- [Deacon 95] Graham Deacon. *Accomplishing Task-Invariant Assembly Strategies by means of an Inherently Accommodating Robot Arm*. Unpublished PhD thesis, Department of Artificial Intelligence, University of Edinburgh, 1995.
- [deFazio *et al.* 84] T. L. de Fazio, D. S. Seltzer, and D. E. Whitney. The Instrumented Remote Center Compliance. *Industrial Robot*, 11(4):238–242, December 1984.
- [deWit *et al.* 93] C. Canudas de Wit, R. Ortega, and S. I. Seleme. Robot Motion Control using Induction Motor Drives. In *Proceedings of the IEEE International Conference on Robotics and Automation*, volume 2, pages 533–538, Atlanta, Georgia, U.S.A., 2nd–6th May 1993.

- [Dijkstra 69] Edsger W. Dijkstra. Structured Programming. In P. Naur, B. Randell, and N. J. Buxton, editors, *Proceedings of the NATO Conference on Software Engineering, Concepts and Techniques*, pages 222–226, New York, New York, U.S.A., 1969. Petrocelli/Charter.
- [Dupont & Yamajako 94] Pierre E. Dupont and Serge P. Yamajako. Jamming and Wedging in Constrained Rigid Bodies. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 2349–2354, San Diego, California, U.S.A., 8th–13th May 1994.
- [ElMaraghy & Rondeau 92] H. A. ElMaraghy and J. M. Rondeau. Automatic Robot Programming Synthesis for Assembly. *Robotica*, 10:113–123, 1992.
- [Feddema & Novak 94] J. T. Feddema and J. L. Novak. Whole Arm Obstacle Avoidance for Teleoperated Robots. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 3303–3309, San Diego, California, U.S.A., 8th–13th May 1994.
- [Fleming 85a] Alan Duncan Fleming. Analysis of Uncertainties in a Structure of Parts: 1. D.A.I. Research Paper n° 271, Department of Artificial Intelligence, University of Edinburgh, 1985.
- [Fleming 85b] Alan Duncan Fleming. Analysis of Uncertainties in a Structure of Parts: 2. D.A.I. Research Paper n° 272, Department of Artificial Intelligence, University of Edinburgh, 1985.
- [Fleming 87] Alan Duncan Fleming. *Analysis of Uncertainties and Geometric Tolerances in Assemblies of Parts*. Unpublished PhD thesis, Department of Artificial Intelligence, University of Edinburgh, 1987.
- [Fu *et al.* 87] K. S. Fu, R. C. Gonzales, and C. S. G. Lee. *Robotics: Control, Sensing, Vision, and Intelligence*. Industrial Engineering Series. McGraw-Hill, 1st edition, 1987.
- [Gini 87] Maria Gini. The Future of Robot Programming. *Robotica*, 5:235–246, 1987.
- [Groover *et al.* 86] Mikell P. Groover, Mitchell Weiss, Roger N. Nagel, and Nicholas G. Odrey. *Industrial Robotics: Technology, Programming, and Applications*. Industrial Engineering Series. McGraw-Hill, 1st edition, 1986.
- [Grupe & Henderson 86] Roderic A. Grupe and Thomas C. Henderson. A Survey of Dextrous Manipulation. UUCS-86-00 7,

- Department of Computer Science, The University of Utah, July, 18th 1986.
- [Gruver *et al.* 84] William A. Gruver, Barry I. Suroka, John J. Craig, and Timothy L. Turner. Industrial Robot Programming Languages: a Comparative Evaluation. *IEEE Transactions on Systems, Man, and Cybernetics*, 14:565–570, July/August 1984.
- [Hayward & Paul 83] V. Hayward and R. P. Paul. Robot Manipulator Control under Unix. In *Thirteenth International Symposium on Industrial Robotics*, Chicago, Illinois, U.S.A., 1983.
- [Hodges 92] Bernard Hodges. *Industrial Robotics*. Newnes Butterworth Heineman Ltd., Oxford, England, U.K., 2nd edition, 1992.
- [Hollingum 94] Jack Hollingum. Simulation, Calibration and Off-Line Programming. *Industrial Robot*, 21(5):20–21, 1994.
- [Hopkins *et al.* 88] S. H. Hopkins, C. J. Bland, and C. B. Byrne. A Toolbox of Assembly Strategies. In *Proceedings of the International Symposium on Industrial Robots*, pages 145–156, Lausanne, Switzerland, 26th–28th April 1988.
- [Hörmann 92] Andreas Hörmann. On-Line Planning of Action Sequences for a Two-Arm Manipulator System. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1109–1114, Nice, France, May 1992.
- [Hwang & Ahuja 92] Y. K. Hwang and N. Ahuja. Gross Motion Planning - A Survey. *Computing Surveys*, 24:219–291, 1992.
- [Inoue 95] Hirochika Inoue. Vision-Based Robotics: a Challenge to Real World Artificial Intelligence. *Advanced Robotics*, 9(4):351–366, 1995.
- [Joseph & Rowland 95] R. M. Joseph and J. J. Rowland. Fusing Diverse Sensor Data by Processing Abstract Images. In *Proceedings of the International Conference on Intelligent Autonomous Systems (IAS-4)*, University of Karlsruhe, Germany, 27th–30th March 1995.
- [Joukhandar *et al.* 94] A. Joukhandar, C. Bard, and C. Laugier. Planning Dextrous Operations using Physical models. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 748–753, San Diego, California, U.S.A., 8th–13th May 1994.

- [Kempf 81] Karl G. Kempf. Robot Command Languages and Artificial Intelligence. Technical paper, Artificial Intelligence Group, Electronic Research Laboratory, Hirst Research Center, General Electric Company Limited, Wembley, Middlesex, England, November 1981.
- [Kim 96] Taehee Kim. *Development of PVDF Tactile Dynamic Sensing in a Behaviour-Based Assembly Robot*. Unpublished PhD thesis, Department of Artificial Intelligence, University of Edinburgh, 1996.
- [Kim *et al.* 93] Taehee Kim, Chris A. Malcolm, and John Hallam. Developing of Vibration Sensors as Event Signature Sensors in Assembly. In *Proceedings of the International Conference on Robotics and Manufacturing*, pages 39–41, Oxford, U.K., September 1993.
- [Kleimann *et al.* 95] Karl Kleimann, Dagmar M. Bettenhausen, and Matthias Seitz. A Modular Approach for Solving the Peg-In-Hole Problem with a Multifingered Gripper. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 758–763, Nagoya, Aichi, Japan, 21st-27th May 1995.
- [Kochan 95] A. Kochan. Off-Line Programming Essential to meet Roobot Demand. *Industrial Robot*, 22(3):27–28, 1995.
- [Kohn 91] Wolf Kohn. Declarative Control Architecture. *Communications of the ACM*, 8:64–79, August 1991.
- [Kondoleon 76] Anthony S. Kondoleon. Application of a Technological Economic Model of Assembly Techniques to Programmable Assembly Machine Configuration. Unpublished M.Sc. thesis, Department of Mechanical Engineering, Massachusetts Institute of Technology, May 1976.
- [Krogh & Sanderson 86] Bruce H. Krogh and Arthur C. Sanderson. Modeling and Control of Assembly Tasks and Systems. CMU-RI-TR-86 1, The Robotics Institute, Carnegie-Mellon University, July 1986.
- [Latombe & Mazer 81] J. C. Latombe and E. Mazer. LM: a High-Level Language for Controlling Assembly Robots. In *Proceedings of the 11th International Symposium on Industrial Robots*, pages 683–690, Tokyo, Japan, October 1981.
- [Latombe 91] Jean-Claude Latombe. *Robot Motion Planning*. Kluwer International Series in Engineering and Com-

- puter Science. Kluwer Academic Publishers, Cambridge, Massachusetts, 1st edition, 1991.
- [Laugier 88a] Christian Laugier. Les Apports Respectifs des Langues Symboliques et de la CAO en Programmation des Robots. *Robotica*, 6:243–253, 1988.
- [Laugier 88b] Christian Laugier. Traitement des Incertitudes en Programmation Automatique des Robots. Rapport de Recherche I-IMAG 68 LIFIA n° 695, Laboratoire d’Informatique Fondamentale et d’Intelligence Artificielle, Informatique et Mathématiques Appliquées de Grenoble, January 1988.
- [Lefebvre & Saridis 92] D. R. Lefebvre and G. N. Saridis. A Computer Architecture for Intelligent Machines. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 2745–2750, Nice, France, 12th-14th May 1992.
- [Lieberman & Wesley 77] L. I. Lieberman and M. A. Wesley. Autopass: an Automatic Programming System for Computer Controlled Mechanical Assembly. *IBM Journal of Research Development*, 21(4):321–333, 1977.
- [Lim *et al.* 91] William Lim, Harry Breul, and Alex Peck. High-Level Modes for Controlling Mobile Robots. In *Proceedings of the SPIE Mobile Robots VI*, volume 1613, Boston, Massachusetts, U.S.A., 14th-15th November 1991.
- [Lindstedt & Olsson 93] Gunnar Lindstedt and Gustaf Olsson. Using Ultrasonic for Sensing in a Robotic Environment. In *Proceedings of the IEEE International Conference on Robotics and Automation*, volume 3, pages 671–676, Atlanta, Georgia, U.S.A., 2nd-6th May 1993.
- [Loncaric 87] J. Loncaric. Normal Forms of Stiffness and Compliance Matrices. *IEEE International Journal of Robotics and Automation*, 3(6), December 1987.
- [Lozano-Pérez & Brooks 85] Tomás Lozano-Pérez and Rodney A. Brooks. An Approach to Automatic Robot Programming. A.I. Memo 842, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, April 1985.
- [Lozano-Pérez 82] Tomás Lozano-Pérez. Robot Programming. A.I. Memo 698, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, December 1982.
- [Lozano-Pérez *et al.* 83] Tomás Lozano-Pérez, Matthew T. Mason, and Russell H. Taylor. Automatic Synthesis of Fine-Motion

- strategies for Robotics. A.I. Memo 759, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, December 1983.
- [Lozano-Pérez *et al.* 92] Tomás Lozano-Pérez, Joseph L. Jones, Emmanuel Mazer, and Patrick A. O'Donnell. *Handey: a Robot Task Planner*. The M.I.T. Press, Cambridge, Massachusetts, U.S.A., 1st edition, 1992.
- [Lundstrom & Rooks 77] Goran Lundstrom and Brian W. Rooks. *Industrial Robots-Gripper Review*. International Fluidics Services, Bedford, 1st edition, 1977.
- [Malcolm & Fothergill 86] Chris A. Malcolm and A. P. Fothergill. Some Architectural Implications of the Use of Sensors. In *Proceedings of the NATO Conference on Robotics*, Pisa, Italy, September 1986. Published also as Edinburgh University DAI Research Paper n°294.
- [Malcolm & Smithers 88] Chris Malcolm and Tim Smithers. Programming Robotic Assembly in terms of Task Achieving Behavioural Modules. *Structural Learning*, 1988.
- [Malcolm 90] Chris A. Malcolm. Behavioural Modules: a new Approach to Robotic Assembly. Communication from author, Autumn 1990.
- [Malcolm 93] Chris A. Malcolm. Behaviour, Purpose, and Meaning. Communication from author, February 1993.
- [Malcolm *et al.* 89] Chris Malcolm, Tim Smithers, and John Hallam. An Emerging Paradigm in Robotics. In *Second Intelligent Autonomous Systems Conference*, Amsterdam, Netherlands, 13th-15th December 1989. Published also as Edinburgh University DAI Research Paper n°447.
- [Martínez & Llario 87] Antonio B. Martínez and Vincenç Llario. Real Time Holes Location: a Step Forward in Bin Picking Tasks. In *Proceedings of the NATO Advance Research Workshop on Sensor Devices and Systems for Robotics*, Barcelona, Catalonia, Spain, 13th-16th October 1987.
- [Mason 84] Matthew T. Mason. Automatic Planning of Fine Motions: Correctness and Completeness. In *Proceedings of the IEEE Computer Society International Conference on Robotics*, Atlanta, Georgia, U.S.A., 13th-15th March 1984. Published also as Carnegie Mellon University CMU-RI-TR-83, Technical Report n°18.

- [Mason 89] Matthew T. Mason. Robotic Manipulation: Mechanics and Planning. In Michael Brady, editor, *Robotics Science*, chapter 7, pages 262–288. The MIT Press, Cambridge, Massachusetts, U.S.A., 1989.
- [Mason *et al.* 88] Matthew T. Mason, Kenneth Y. Goldberg, and Russel H. Taylor. Planning Sequences of Squeeze-Grasps to Orient and Grasp Polygonal Objects. In *Proceedings of Ro.Man.Sy '88, 7th CISM-IFTOMM Symposium on Theory and Practice of Robots and Manipulators*, Udine, Italy, September 1988. Published also as Carnegie Mellon University CMU-CS-88, Technical Report n°127.
- [Masutani *et al.* 94] Yasuhiro Masutani, Takeshi Iwatsu, and Fumio Miyazaki. Motion Estimation of Unknown Rigid Body under no External Forces and Moments. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1066–1072, San Diego, California, U.S.A., 8th-13th May 1994.
- [Mataric 92a] Maja J. Mataric. Behaviour-Based Control: Main Properties and Implications. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 46–54, Nice, France, 12th-14th May 1992.
- [Mataric 92b] Maja J. Mataric. Designing Emergent Behaviors: from Local Interactions to Collective Intelligence. In *From Animals to Animats, Proceedings of the Second International Conference on Simulation of Adaptive Behavior*, pages 432–441, 1992.
- [Matarić 94] Maja J. Matarić. *Interaction and Intelligent Behavior*. Unpublished PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1994.
- [Mazer 83] E. Mazer. LM-Geo: Geometric Programming of Assembly Robots. In *Advanced Software in Robotics*, Liege, Belgium, 1983. Published also as Laboratoire IMAG Technical Paper (Grenoble, France, 1982).
- [McFarland & Bösser 93] David McFarland and Thomas Bösser. *Intelligent Behavior in Animals and Robots*. The M.I.T. Press, Cambridge, Massachusetts, U.S.A., 1st edition, 1993.
- [McLachlan *et al.* 92] D. S. McLachlan, B. R. Bannister, A. D. Joyce, and D. McManus. Process Control Parameters in Flexible Assembly. In *Proceedings of the 11th IASTED, Modelling, Identification, and Control*, Innsbruck, Austria, February 1992.

- [McManus *et al.* 92] D. McManus, K. Selke, B. R. Bannister, A. D. Joyce, D. S. McLachlan, and E. Nadarajah. Automatic Generation of Control Structures for Assembly Monitoring. In *Proceedings of ICARV 92 – 2nd International Conference on Automation, Robotics, and Computer Vision*, Singapore, 15th-18th September 1992.
- [Mehdian & Rahnejat 94] M. Mehdian and H. Rahnejat. A Dextrous Anthropomorphic Hand for Robotic and Prosthetic Applications. *Robotica*, 12:455–463, 1994.
- [Meitinger & Pfeiffer 94] Th. Meitinger and F. Pfeiffer. Automated Assembly with Compliant Mating Parts. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1462–1467, San Diego, California, U.S.A., 8th-13th May 1994.
- [Metta & Oddera 93] Giorgio Metta and Andrea Oddera. RCI-RCCL Introduzione al Sistema. Technical Report TR 5, LIRA-Lab-DIST, University of Genova, September 1993.
- [Miura & Ikeuchi 95] Jun Miura and Katsushi Ikeuchi. Task-Oriented Generation of Visual Sensing Strategies in Assembly Tasks. CMU-CS-95 116, School of Computer Science, Carnegie Mellon University, February 1995.
- [Montana 91] David J. Montana. The Condition for Contact Grasp Stability. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 412–417, Sacramento, California, U.S.A., 9th-11th April 1991.
- [Mukerjee & Mali 94] Amitabha Mukerjee and Amol D. Mali. Agent Models of Intelligence - Limitations and Prospects. I.I.T. Technical Report ME 24, Center for Robotics, Indian Institute of Technology, 1994.
- [Nakagaki *et al.* 95] Hirofumi Nakagaki, Kosei Kitagaki, and Hideo Tsukune. Study of Insertion Task of a Flexible Beam into a Hole. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 330–335, Nagoya, Aichi, Japan, 21st-27th May 1995.
- [Nakano *et al.* 94] M. Nakano, N. Sugiura, M. Tanaka, and T. Kuno. ROPSII: Manufacturing System Simulator with Object Oriented Simulation Language. In *Proceedings of New Directions in Simulation for Manufacturing and Communications*, pages 493–498, 1994.

- [Nevins & Whitney 78] James L. Nevins and Daniel E. Whitney. Computer-Controlled Assembly. *Scientific American*, pages 62–74, February 1978.
- [Nicolson & Fearing 91] Edward J. Nicolson and Ronald S. Fearing. Dynamic Modeling of a Part Mating Problem: Threaded Fastener Insertion. In *Proceedings of the IROS '91 - IEEE International Workshop on Intelligent Robots and Systems*, pages 30–37, Osaka, Japan, 3rd–5th November 1991.
- [Nilsson & Nielsen 92] Klas Nilsson and Lars Nielsen. An Architecture for Application Oriented Robot Programming. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1115–1121, Nice, France, 12th–14th May 1992.
- [Nnaji 93] Bartholomew O. Nnaji. *Theory of Automatic Robot Assembly and Programming*. Chapman & Hall, 2-6 Boundary Row, London SE1 8HN, U.K., 1st edition, 1993.
- [Noreils & Chatila 89] Fabrice R. Noreils and Raja G. Chatila. Control of Mobile Robot Actions. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 701–707, Resort Scottsdale, Arizona, U.S.A., 14th–19th May 1989.
- [Norman 94] T. J. Norman. Motivated Goal and Action Selection. In *AISB94 Workshop Series*, University of Leeds, 11th–13th April 1994.
- [Owen 84] Anthony E. Owen. *Flexible Assembly Systems: Assembly by Robots and Computerized Integrated Systems*. Plenum Press, 1st edition, 1984.
- [Owen 85] Anthony E. Owen. *Assembly with Robots*. Kogan Page, 1st edition, 1985.
- [Owen 86] Tony Owen. Robotics: the startegic issues. *Robotica*, 4:117–122, 1986.
- [Paetsch & vonWichert 93] W. Paetsch and G. von Wichert. Solving Insertion Tasks with a Multifingered Gripper by Fumbling. In *Proceedings of the IEEE International Conference on Robotics and Automation*, volume 3, pages 173–179, Atlanta, Georgia, U.S.A., 2nd–6th May 1993.
- [Park 77] W. T. Park. Minicomputer Software Organization for Control of Industrial Robots. In *Joint Automatic Control Conference*, San Francisco, California, U.S.A., 1977.

- [Paulos & Canny 93] Eric Paulos and John Canny. Informed Peg-In-Hole Insertion Using Optical Sensors. In *Proceedings of the SPIE Sensor Fusion VI*, volume 2059, Boston, Massachusetts, U.S.A., 7th-8th September 1993.
- [Paulos & Canny 94] Eric Paulos and John Canny. Accurate Insertion Strategies Using Simple Optical Sensors. In *Proceedings of the IEEE International Conference on Robotics and Automation*, San Diego, California, U.S.A., 8th-13th May 1994.
- [Paulos 93] Eric Paulos. Trends in Grasping. Robotics Grasping: Literature Overview, 1993.
- [Petropoulakis & Malcolm 90] Lykourgos Petropoulakis and Chris Malcolm. Programming Autonomous Assembly Agents: Functionality and Robustness. *Mechatronic Systems Engineering*, 1:107–113, 1990.
- [Pettinaro & Malcolm 94] Giovanni Cosimo Pettinaro and Chris Malcolm. Electric Gropper Development. D.A.I. Technical Report n° 28, Department of Artificial Intelligence, University of Edinburgh, September 1994.
- [Pettinaro & Malcolm 95a] Giovanni Cosimo Pettinaro and Chris Malcolm. A Program for Describing Two Similar Assemblies. D.A.I. Technical Report n° 35, Department of Artificial Intelligence, University of Edinburgh, December 1995.
- [Pettinaro & Malcolm 95b] Giovanni Cosimo Pettinaro and Chris Malcolm. Two-Handed Robotic System. D.A.I. Technical Report n° 34, Department of Artificial Intelligence, University of Edinburgh, May 1995.
- [Popplestone & Ambler 83] Robin J. Popplestone and Pat A. Ambler. A Language for specifying Robot Manipulations. In Alan Pugh, editor, *Robotic Technology*, pages 125–141. Peter Peregrinus, 1983.
- [Popplestone 83] Robin J. Popplestone. Group Theory and Robotics. In *Proceedings of the 1st International Symposium of Robotics Research*, M.I.T., Massachusetts, U.S.A., September 1983. Published also as Edinburgh University DAI Research Paper n°196.
- [Popplestone et al. 78] R. J. Popplestone, A. P. Ambler, and I. Bellos. RAPT: a Language for Describing Assemblies. *Industrial Robot*, 5(3):131–137, 1978.

- [Popplestone *et al.* 80] R. J. Popplestone, A. P. Ambler, and I. Bellos. An Interpreter for a Language for Describing Assemblies. *Artificial Intelligence*, 14(1):79–107, 1980.
- [Powers 73] William T. Powers. *Behaviour: the Control of Perception*. Wildwood House, 1st edition, 1973.
- [Regev 95] Yoram Regev. The Evolution of Off-Line Programming. *Industrial Robot*, 22(3):3, 1995.
- [Reynaerts & vanBrussel 95] Dominiek Reynaerts and Hendrik van Brussel. Whole Finger Manipulation with a Two-Fingered Robot Hand. *Advanced Robotics*, 9(5):505–518, 1995.
- [Rock 89] S. T. Rock. Developing Robot Programming Languages using an existing Language as a Base - A Viewpoint. *Robotica*, 7:71–77, 1989.
- [Rodrigues *et al.* 95] M. A. Rodrigues, Y. F. Li, M. H. Lee, J. J. Rowland, and C. King. Robotic Grasping of Complex Objects without Full Geometrical Knowledge of the Shape. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 737–742, Nagoya, Aichi, Japan, 21st-27th May 1995.
- [Rogalinski 94] Pawel Rogalinski. An Approach to Automatic Robots Programming in the Flexible Manufacturing Cell. *Robotica*, 12:263–279, 1994.
- [Sato *et al.* 95] Shuichi Sato, Masaru Nagako, Fumiko Kubota, Norio Sugiura, Minoru Tanaka, Toshihiko Koyama, and Seiya Nakayama. Task-Level Teaching System with Trajectory Planning for Industrial Robots. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 2394–2400, Nagoya, Aichi, Japan, 21st-27th May 1995.
- [Schimmels & Pushkin 90] J. M. Schimmels and M. A. Pushkin. Synthesis and Validation of Non-Diagonal Accommodation Matrices for Error-Corrective Assembly. In *Proceedings of the IEEE International Conference on Robotics and Automation*, Hyatt Regency, Cincinnati, Ohio, U.S.A., 13th-18th March 1990.
- [Shahinpoor & Zohoor 91] M. Shahinpoor and H. Zohoor. Analysis of Dynamic Insertion Type Assembly for Manufacturing Automation. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 2458–2464, Sacramento, California, U.S.A., 9th-11th April 1991.

- [Sinha & Goldenberg 93] Pramath R. Sinha and Andrew A. Goldenberg. A Unified Theory for Hybrid Control of Manipulators. In *Proceedings of the IEEE International Conference on Robotics and Automation*, volume 1, pages 343–348, Atlanta, Georgia, U.S.A., 2nd-6th May 1993.
- [Söderquist & Wernersson 92] Bertill A. T. Söderquist and Åke Wernersson. Information for Assembly from Impacts. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 2012–2017, Nice, France, 12th-14th May 1992.
- [Steels 93] Luc Steels. The Artificial Life Roots of Artificial Intelligence. *Artificial Life Journal*, 1:89–125, 1993.
- [Steels 94a] Luc Steels. A Case Study in the Behaviour-Oriented Design of Autonomous Agents. In *Proceedings of the Conference on Simulations and Adaptive Behaviour*, Cambridge, Massachusetts, U.S.A., 1994.
- [Steels 94b] Luc Steels. Mathematical Analysis of Behavior Systems. In *Proceedings of the PERARC Conference*, Lausanne, Switzerland, 1994.
- [Stobart 87] R. K. Stobart. Geometric Tools for the Off-Line Programming of Robots. *Robotica*, 5:273–280, 1987.
- [Strenn *et al.* 94] Stephen Strenn, T. C. Hsia, and Karl Wilhelmsen. A Collision Avoidance Algorithm for Telerobotics Applications. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 359–365, San Diego, California, U.S.A., 8th-13th May 1994.
- [Strip 88] David R. Strip. Insertions Using Geometric Analysis and Hybrid Force-Position Control Method and Analysis. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1744–1751, Franklin Plaza Hotel, Philadelphia, Pennsylvania, U.S.A., 24th-29th April 1988.
- [Tao *et al.* 90] J. M. Tao, J. Y. S. Luh, and Y. F. Zheng. Compliant Coordination Control of Two Moving Industrial Robots. *IEEE Transactions of Robotics and Automation*, 6(3), June 1990.
- [Taylor *et al.* 94] Michael Taylor, Andrew Blake, and Adrian Cox. Visually Guided Grasping in 3D. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 761–766, San Diego, California, U.S.A., 8th-13th May 1994.

- [Tempelmeier & Kuhn 93] Horst Tempelmeier and Heinrich Kuhn. *Flexible Manufacturing Systems: Decision Support for Design and Operation*. Systems Engineering Series. Wiley, 1st edition, 1993.
- [Torrance 94] Steve Torrance. The Robots New Mind. In *AISB94 Workshop Series*, University of Leeds, 11th-13th April 1994.
- [Tung & Kak 94] C. P. Tung and A. C. Kak. Integrating Sensing, Task Planning and Execution. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 2030–2037, San Diego, California, U.S.A., 8th-13th May 1994.
- [Turing 37] Alan M. Turing. On Computable Numbers with an Application to the *entscheidungsproblem*. *Proc. London Mathematical Society*, 42, 1937.
- [vanAken & vanBrussel 88] L. van Aken and H. van Brussel. Robot Programming Languages: the Statement of the Problem. *Robotica*, 6:141–148, 1988.
- [vanBruggen *et al.* 93] Marnix van Bruggen, Jan Peter Baartman, and Willem F. Bronswoort. Grips on Parts. In *Proceedings of the IEEE International Conference on Robotics and Automation*, volume 2, pages 828–833, Atlanta, Georgia, U.S.A., 2nd-6th May 1993.
- [Watson 78] P. C. Watson. Remote Center Compliance System. U.S. Patent 4098001, July 1978.
- [Whitney & Gilbert 93] Daniel E. Whitney and Olivier L. Gilbert. Representation of Geometric Variations Using Matrix Transforms for Statistical Tolerance Analysis in Assemblies. In *Proceedings of the IEEE International Conference on Robotics and Automation*, volume 2, pages 314–321, Atlanta, Georgia, U.S.A., 2nd-6th May 1993.
- [Whitney 85] Daniel E. Whitney. Part Mating in Assembly. In Shimon Y. Nof, editor, *Handbook of Industrial Robotics*, chapter 64, pages 1084–1116. John Wiley & Sons, 1985.
- [Whitney 89] Daniel E. Whitney. A Survey of Manipulation and Assembly: Development of the Field and Open Research Issues. In Michael Brady, editor, *Robotics Science*, chapter 8, pages 291–348. The MIT Press, Cambridge, Massachusetts, U.S.A., 1989.

- [Wikman & Newman 92] Thomas S. Wikman and Wyatt S. Newman. A Fast, On-Line Collision Avoidance Method for a Kinetically Redundant Manipulator Based on Reflex Control. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 261–266, Nice, France, 12th-14th May 1992.
- [Wikman *et al.* 93] Thomas S. Wikman, Michael S. Branicky, and Wyatt S. Newman. Reflexive Collision Avoidance: a Generalized Approach. In *Proceedings of the IEEE International Conference on Robotics and Automation*, volume 3, pages 31–36, Atlanta, Georgia, U.S.A., 2nd-6th May 1993.
- [Williams 95] Tomos G. T. Williams. A Proposal for the Investigation of Force/Torque Sensing in Assembly Robot Behavioural Modules. UWA-DCS-95 n° 19, Centre for Intelligent Systems, Department of Computer Science, University of Wales, Aberystwyth, 27th November 1995.
- [Wilson & Latombe 92] Randall H. Wilson and Jean-Claude Latombe. Reasoning about Mechanical Assembly. In *Proceedings of the International Symposium on Artificial Intelligence*, Cancún, Mexico, 7th-11th December 1992.
- [Wilson 92] Myra Scott Wilson. *Achieving Reliability using Behavioural Modules in a Robotic Assembly System*. Unpublished PhD thesis, Department of Artificial Intelligence, University of Edinburgh, 1992.
- [Wilson 94] Myra S. Wilson. Behaviour-Based Robotic Assembly. In *AISB94 Workshop Series*, University of Leeds, 11th-13th April 1994.
- [Wittenberg 95] G. Wittenberg. Developments in Off-Line Programming: an Overview. *Industrial Robot*, 22(3):21–23, 1995.
- [Wu & Hopkins 90] M. H. Wu and S. H. Hopkins. Assembly Strategies for Tandem Peg/Hole and In-Phase Parallel Peg/Hole. UWCC Technical Note n° 37, School of Electrical, Electronic and Systems Engineering, University of Wales College of Cardiff, March 1990.
- [Yin *et al.* 84] Baolin Yin, Pat A. Ambler, and Robin J. Popplestone. Combining Vision Verification with a High Level Robot Programming Language. In *Proceedings of the 4th International Conference ROVISEC*, pages 371–379, 1984.

- [Young 94] Roger Young. Robots and Intentionality. In *AISB94 Workshop Series*, University of Leeds, 11th-13th April 1994.
- [Zheng *et al.* 91] Y. F. Zheng, R. Pei, and C. Chen. Strategies for Automatic Assembly of Deformable Objects. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 2458–2464, Sacramento, California, U.S.A., 9th-11th April 1991.