$\mathcal{RS}$: A Formal Model of Distributed Computation For Sensory-Based Robot Control[1]

Damian Lyons

COINS Technical Report 86-43

September 1st, 1986

Laboratory for Perceptual Robotics
Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

[1]Preparation of this paper was supported in part by grants ECS-8108818 and DMC-8511959 from NSF

$\mathcal{RS}$: A Formal Model of Distributed Computation For Sensory-Based Robot Control

A Dissertation Presented

By

Damian Martin Lyons

Submitted to the Graduate School of the
University of Massachusetts in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 1986

Department of Computer and Information Science

© Copyright by Damian Martin Lyons 1986

All Rights Reserved.

This research was supported in part by grant numbers ECS-8108818 and DMC-8511959 from the National Science Foundation.

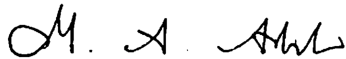
$\mathcal{RS}$: A FORMAL MODEL OF DISTRIBUTED COMPUTATION FOR SENSORY-BASED ROBOT CONTROL

A Dissertation Presented

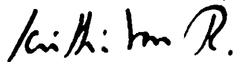
By

DAMIAN MARTIN LYONS

Approved as to style and content by:

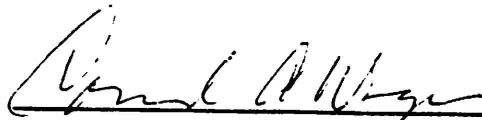


Prof. Michael A. Arbib, Chairperson of Committee



Prof. Krithivasan Ramamritham, Member

Prof. Theodore Djaferis, Outside Member



Prof. Conrad A. Wogrin, Department Chair
Computer and Information Science

Acknowledgement

I wish to thank my committee chairman, Michael Arbib, for the freedom and encouragement he gave me in developing the ideas in this dissertation, and to acknowledge the debt that this work owes to his theory of schemas. My thanks are also due to Krithi Ramamritham and Ted Djaferis, the other members of my dissertation committee, their feedback was essential.

I am especially indebted to Jeanie Shippey Lyons, my wife. Her support and friendship saw me through all of this, and her help with the final drafts of this document were invaluable. I also owe a tremendous debt of gratitude to my parents Patrick and Catherine Lyons, who taught me to work hard at what I wanted to do, even if it took me far from home.

Finally, thanks are due to the members of the Laboratory for Perceptual Robotics (LPR) with whom I have spent many hours of fruitful discussion; especially the members of the 'Hand Group', where all this started: Thea Iberall and Michael Arbib, and also Gerry, Judy, TV, Ruki, Gordon, Randy, and Miles.

ABSTRACT

$\mathcal{RS}$: A Formal Model of Distributed Computation

For Sensory-Based Robot Control

September 1986

Damian Martin Lyons, B.A.I.,B.A., Trinity College, Dublin

M.Sc., Trinity College, Dublin

Ph.D., University of Massachusetts

Directed by: Professor Michael A. Arbib

Robot systems are becoming more and more complex, both in terms of available degrees of freedom and in terms of sensors. It is no longer possible to continue to regard robots as peripheral devices of a computer system, and to program them by adapting general-purpose programming languages. This dissertation analyzes the inherent computing characteristics of the robot programming domain, and formally constructs an appropriate model of computation. The programming of a dextrous robot hand is the example domain for the development of the model.

This model, called $\mathcal{RS}$, is a model of distributed computation: The basic mode of computation is the interaction of concurrent computing agents. A schema in $\mathcal{RS}$ describes a class of computing agents. Schemas are instantiated to produce computing agents, called SIs, which can communicate with each other via input and output ports. A network of SIs can be grouped atomically together in an Assemblage, and appears externally identical to a single SI. The sensory and motor interface to $\mathcal{RS}$ is a set of primitive, predefined schemas. These can be grouped arbitrarily with built-in knowledge in assemblages to form task-specific

object models. A special kind of assemblage called a task-unit is used to structure the way robot programs are built.

The formal semantics of $\mathcal{RS}$ is automata theoretic; the semantics of an SI is a mathematical object, a Port Automaton. Communication, port connections, and assemblage formation are among the $\mathcal{RS}$ concepts whose semantics can be expressed formally and precisely. A temporal logic specification and verification method is constructed using the automata semantics as a model. While the automata semantics allows the analysis of the model of computation, the temporal logic method allows the top-down synthesis of programs in the model.

A computer implementation of the $\mathcal{RS}$ model has been constructed, and used in conjunction with a graphic robot simulation, to formulate and test dextrous hand control programs. In general, $\mathcal{RS}$ facilitates the formulation and verification of versatile robot programs, and is an ideal tool with which to introduce AI constructs to the robot domain.

TABLE OF CONTENTS

LIST OF FIGURES xii

LIST OF TABLES xiv

CHAPTER

I. Introduction 1

§1. Motivation for Developing a Model of Computation for Robots . . . 3

§2. Strategy for the Development of the Model 4

§3. Overview of the Dissertation 5

II. Example Domain: Grasping and Manipulation with a Dextrous Hand 7

§1. Previous Work in this Area 8

§2. A General Model of Grasping and Manipulation 10

§2.1 Example Task 11

§2.2 A Framework for Grasping 14

§3. A Simple Set of Grasps 20

§3.1 Outline of the SSG Grasps 21

§3.2 Kinematic Details of the SSG grasps 22

§3.3 Encompass Grasp: (P_e, A_e, M_e) 30

§3.4 Lateral Grasp: (P_l, A_l, M_l) 35

§3.5 Precision Grasp: (P_p, A_p, M_p) 38

§4. Selection of Grasps 41

§5. Summary and Discussion 44

III. Characteristics of the Robot Domain 45

§1. Observation One: Inherent Parallelism 45

§2. Observation Two: Formal Verification 47

§3. Observation Three: Perception and Action 48

§4. Observation Four: Parameterizable Action Plans and Object Models 50

§5. Summary 51

IV. Structure of the Computational Model 53

§1. Part I: An Introduction to $\mathcal{RS}$ 53

§2. Part II: The $\mathcal{RS}$ Model 62

§2.1 Distributed Property 62

§2.2 Instantiation Property 63

§2.3 Primitive Schemas 69

§2.4 Aggregation Property: 72

§2.5 Fan property: 76

§2.6 Classification Property: 79

§2.7 The Task-Unit Assemblage 83

V. Formal Semantics 87

§1. Introduction 87

§2. The Port Automaton Model 88

§3. Construction of the Formal Semantics 101

VI. Verification in $\mathcal{RS}$ 133

§1. Network and SI Verification 134

§2. Temporal Logic 134

§3. The Model 138

§4. SI Assertions 144

§4.1 Proof Rules 148

§4.2 Example 149

§5. Reasoning about Schema Behavior 153

§5.1 Extension of Behavior 154

§5.2 Transition Axioms 154

§5.3 Example 158

§6. Task-Unit Assertions 160

§7. Limitations of the Verification System 161

VII. Representational Issues 162

§1. Recursion and Iteration 162

§2. Some Common Data Structures 165

§3. AI Representations 171

VIII. Implementation of the SSG System 174

§1. Example Assembly Program 174

§1.1 The Example Task 175

§1.2 Sequencing of Robot Programs 175

§2. The SSG System 177

§2.1 Design of the Implementation 179

§2.2 SSG Schema Specifications 195

IX. Comparisons 199

§1. Robot and AI Models 199

§1.1 Previous Schema Work 199

§1.2 Robot Computation Models 202

§1.3 The NBS Robot System 202

§1.4 The Actor Model of Computation 207

§2. General Purpose Models 208

§2.1 Communicating Sequential Processes 208

§2.2 OCCAM 212

X. Conclusion 214

§1. Summary 214

§2. Implementation 216

§3. Future Research 216

APPENDIX

A. Additional $\mathcal{RS}$ Programs 218

B. Glossary 226

§1. The Grasping and Manipulation Terminology 226

§2. The $\mathcal{RS}$ model Terminology 228

§3. The Temporal Logic Terminology 230

BIBLIOGRAPHY 232

LIST OF FIGURES

1. The Components of the Example Assembly Task. 12

2. The Completed Example Assembly Task. 12

3. Wrist (a) and Object (b) Centered Frames. 15

4. The Kinematics of the Reach. 16

5. Hand Models: DOF and Link Numbering. 24

6. Graphs for Finger-tip/Thumb-tip Separation Equations. 26

7. Geometrically-Reasoned Inverse Kinematic Solution. 29

8. The Encompass Grasp. 31

9. Encompass Acquisition Strategy. 34

10. The Lateral Grasp. 36

11. Compliance of Lateral Grasp to Shape (a) and Limited Manipulation of the Grasped Object (b). 38

12. The Precision Grasp. 39

13. Recursive Network of Factorial. 68

14. The Counter Assemblage. 75

15. Fan-in (a) and Fan-out (b). 77

16. The **Center** Assemblage. 79

17. The **Obj?** Precondition. 81

18. The Port Connection Map. 89

19. Construction of the Network Automaton. 92

20. Network Described by Port Connections (a) Versus Network Map (b). 93

21. The Semantics of the Forall Statement. 124

22. The Sequential Version of the **Forall** Operation. 126

23. The **move$joint** Assemblage. 150

24. The *ith* Level of Recursion of **Sumfromzero**. 164

25. Internals of **stack**. 166

26. Internals of **array**. 169

27. Internal Structure of the **grasp** Schema. 184

28. The Reach, Preshape and Acquisition Sequence. 186

29. The **preshape^g** Assemblage. 189

30. NBS Hierarchical Robot Controller. 203

31. A Level of the NBS System. 205

32. $\mathcal{RS}$ and CSP Composition Operations: Part I. 211

33. $\mathcal{RS}$ and CSP Composition Operations: Part II. 211

34. $\mathcal{RS}$ and CSP Composition Operations: Part III. 212

LIST OF TABLES

1. Linear Angular Separation Equations for Both Hand Models. 27

2. Object Characteristics for the Example Assembly Task. 42

3. $\mathcal{RS}$-TL1 Assembly Program for the Example Assembly Task. 176

CHAPTER I

INTRODUCTION

There is a strong link between the characteristic called intelligence and the ability to use tools. It is quite certain that much of man's success is due to his ability to extend his mind and body by the tools which he has created. At the dawn of civilization, these tools were simple and designed to extend human physical powers, e.g., the jawbone of an ass, a flint knife, etc. The advance of civilization has been matched by the development of more complex and general-purpose tools. The *robot* concept could possibly yield the ultimate general-purpose tool - a 'machine' which can be instructed to carry out any task a human can, and which will accomplish it at least as well as a human. However, the current state of robot research is quite primitive, in contrast to this ultimate scenario.

One of the most challenging problems today is the attempt to build a robot which has some measure of the versatility of a human. As generally defined, a *robot system* consists of a *manipulator*, a set of *sensors* and a *controller*. The manipulator is that part of the system (normally mechanical) which is used to effect changes in the robot's external environment. Each sensor 'measures' some property of the external world. In the most common class of robot system, the *Industrial Robot*, the manipulator is usually a single "arm", a series of rigid links connected by controllable joints. The controller is that element of the robot system which determines the behavior of the manipulator. Simple controllers just store a sequence of joint position values which they transmit to the joint actuators in some fixed sequence. More complex controllers use *sensory input* and some *internal task description* to make the response of the manipulator appropriate to the external environment of the robot. This dissertation addresses the structure of such a controller.

In almost all cases, this controller is a general-purpose computer. The programs which are executed on this computer are not, however, general-purpose programs — they can be placed in one of the following classes:

1. Servo-Control: These programs usually implement a control-theoretic approach to ensuring that parameters of the manipulation structure, such as joint positions, velocities, torques, stiffnesses, etc., can be reliably maintained [52,70].

2. Sensory Processing: Programs which refine or combine the raw data produced by the robot's sensors [7,69,86]. These may include programs for object recognition — classifying objects in the robot's (immediate) environment into parameterized instances of some set of object classes.

3. Tactical Control/Planning: Programs which break down the description of a single atomic operation to be carried out on an object (one of a fixed class of such operations), into a series of commands to the servo-control and sensory processing programs [48,49]. Operations which are referenced against objects, as opposed to parts of the manipulator, are called *task-level* actions.

4. Strategic Planning: Programs which take a high-level, goal-oriented description of some complex task and determine a sequence of atomic task-level actions which will result in the robot carrying out the task [25]. A *Dynamic Planner* [89] is a strategic planner which can determine from sensory information if some part of its planned sequence of operations has failed, and how to resolve this failure.

The vast majority of robots today are *Industrial Robots*; these occupy the lowest rung of the robot hierarchy. Usually, they have only primitive position sensors, a rudimentary servo-control system, and not much else. Increasingly, the trend in robot construction is to build robots with multiple, possibly redundant, degrees of freedom, and which have many sensor systems. We shall refer to such a robot as a *complex robot system*. These robots exist currently in research environments only — not because they are specially difficult to construct, but because they are difficult to control.

The essence of this control problem is the fact that robots are currently being programmed using general-purpose programming languages, or dialects of such languages, in which the robot is treated simply as a peripheral device. It is difficult to program complex robot systems to exhibit their full potential for versatile and adaptable behavior with the tools developed for general-purpose programming. In this dissertation we argue that the computation carried out by a robot controller is a *special class of computation*. In order to be able to program complex robots well, it is necessary to construct a *model*

of computation, a way to describe how computation occurs, which is specifically tuned to the robot domain. This dissertation will describe the specification, construction and analysis of just such a model of computation.

§1. Motivation for Developing a Model of Computation for Robots

Our argument is that the computational needs of the robot domain are distinct enough to be embodied into a formal description of the way in which computation is done. To be most effective, a model of computation should have a *formal semantics* by which expressive (how easy it is to express useful programs) and computational (what algorithms can be executed) power can be quantified. The goal in constructing such a system is to simplify the expression and examination of robot behavior which might be too complex to specify and examine in a general-purpose computing scenario. Before the advent of complex robot systems, it was difficult to justify taking this step, and (therefore) in the current literature it is programming languages, rather than models of computation, which are discussed; and, in general, robot programming languages are constructed *within* general-purpose programming languages.

The trend in robot languages can be observed clearly; both Grovel et al. [27] and Noyes [67] provide an overview of current robot languages. Initial robot languages, such as VAL [82], were essentially collections of motor control statements with little computational ability. They expressed low-level motor control well but frequently neglected sensory input. All actions in VAL are specified in terms of the manipulator's degrees of freedom (DOF). For example, to acquire an object, one can only specify that the gripper jaws close to a specific position and orientation. Depending on the current state of the world, this *may* or *may not* be equivalent to acquiring the object; the art in programming these languages is to ensure that the manipulator's environment is so strongly constrained that objects are where the program expects them to be. Such languages are called *explicit* or *robot-level* languages.

Rather than follow this with an examination the *computational needs* of the robot domain, subsequent language improvements focused on greater general-purpose programming power, and *grafted* appropriate data-types (e.g., *vector* and *frame*) and operations on to general-purpose programming languages, e.g., ALGOL-like languages such as *Pascal* in the case of Stanford's AL [62] and Karlsrue's SRL [14].

The next step in robot language development considered the concept of *objects* against which actions could be referenced. Their mistake was in attempting to do this via the control and data structures of a general-purpose programming language. Instead of stepping back and attempting to determine what computational characteristics were relevant to the robot domain, the trend in language development was to provide more and more *intelligent* or *task-level* [49] operations. This resulted in the development of complex languages such as IBM's AUTOPASS [48] and Edinburgh's RAPT [72].

These languages explored the interface between the human task-level specification and the robot-level specification; yet, they neglected to address how a robot task can be best represented or executed on a computer system. It is very important to realize that this is much more than simply an implementation issue. Few complex programs would be conceived of, let alone written, if all programming was done in machine language. Similarly, without matrices and vectors, robot kinematics might be impossibly complex. An appropriate working model can bring the seemingly impossible into the realm of the possible.

Our approach is as follows: We shall analyze what is known about the robot domain, and formulate computational characteristics which are uniquely relevant. These characteristics will be the foundations of our model of computation. Note that a model of computation can be used in several ways: A machine can be built, in VLSI say, which directly implements the model; a program can be built, in microcode or in a general-purpose programming language, which emulates the model on a standard computer; or programs can be written in a general-purpose programming language in such a way that they obey the structure of the model. These are details of the *implementation* of the model, and we shall not raise them again until Chapter 10, where we describe an experimental implementation.

§2. Strategy for the Development of the Model

Having motivated the development of the model, we now plan how best to develop the model structure. It is clear that the basic structure of the model must come from observations about the robot domain. We shall investigate one particular complex robot domain in detail in order to help generate these observations. This domain will also serve to provide examples throughout the rest of the dissertation, including a chapter-

long implementation example. As we have mentioned, there currently exist a number of complex robot systems. We choose as our example domain, that of grasping and manipulation with a *dextrous robot hand*. We shall argue in the next chapter that the control of such a hand involves many of the problems central to robotics. In our analysis we shall not restrict ourselves to this domain, but it will be the main example which is carried all the way through the dissertation. Our strategy for model development is as follows:

1. The computational characteristics of the robot domain are formalized in a list of observations.
2. The model is presented informally, and its structure is related back to the observations on computational characteristics.
3. An automata-theoretic semantics is developed, and then used to investigate the structures previously presented informally.
4. A temporal logic system is constructed with the automaton semantics as its model, for the purpose of specification and verification of program behavior.
5. A number of examples are implemented in the model in order to demonstrate its representative power and the validity of the assumption that developing a special model of computation for robots allows the specification of complex robot behavior.

§3. Overview of the Dissertation

This dissertation is organized into 10 Chapters, followed by an Appendix, a Glossary of Terms, and a Bibliography. In more detail, these are:

Chapter 2: This chapter presents an analysis of a particular example of a complex robot system, the *Dextrous Hand*. The goal of this chapter is to provide fuel for the specifications of the model of computation, and to provide an example domain in which to use the model.

Chapter 3:

From the analysis in Chapter 2, augmented with references to the literature, a set of four observations on the nature of robot computation are made. These observations form the specification for the model of computation.

Chapter 4: The model is presented informally. This chapter starts with a global overview of the model, relating its structure back to the specifications of Chapter 3. The second part of the chapter is a detailed description of the model notation, illustrated with examples.

Chapter 5: In this chapter, the automata-theoretic semantics for the model is formulated using the *Port Automaton Model* of Steenstrup et al. [1983] as its basis. A definition of a port automaton is constructed which has special ***input*** and ***output*** ports, as opposed to the general definition in which all ports are ***bidirectional***. The concept of the ***input-output trace*** of an automaton is developed in order to establish the connection between automata with unidirectional ports and those with bidirectional ports. This definition is also used to build the semantics of the model notation developed in the previous chapter.

Chapter 6: A temporal logic is constructed to aid in the development of programs in the model. A method for verifying that programs in the model obey some temporal logic specification is developed, as well as mechanisms for the top-down development of such programs from outline specifications. Where Chapter 5 provides us with a tool to analyze the model of computation, Chapter 6 provides us with a tool to synthesize programs in the model.

Chapter 7: Some common programming contructs and data structures are translated into the terms of our model in this chapter: recursive and iterative programs; data structures such as sets, arrays, stacks, etc; and typical AI representations, such as production systems, semantic networks and frames.

Chapter 8: This is a chapter-long example of the implementation, in our model, of the grasping and manipulation material in Chapter 2.

Chapter 9: In this chapter, our model is compared with models which have similar goals. In particular, the NBS system [2], CSP [32] and OCCAM [59], and Actors [30].

Chapter 10: This chapter presents our conclusions. An implementation of a version of the model is described, and the directions of future research discussed.

Appendix: More parts of the grasping and manipulation program are presented, for reference.

Glossary: This summarizes the notation and terminology used in the dissertation. It has three sections: kinematics of grasping, formal semantics, and temporal logic.

CHAPTER II

EXAMPLE DOMAIN: GRASPING AND MANIPULATION WITH A DEXTROUS HAND

In this chapter, we examine the control of a complex robot system, *a dextrous robot hand*. From this analysis, as well as from the current literature, we shall draw the computational characteristics of the robot control domain (Chapter 3). Once our model of computation has been constructed (Chapters 4, 5 and 6), we shall program the grasp analysis in this chapter as a detailed example (Chapter 8). For those mainly interested in the robot model of computation, this chapter can be skimmed over and used as reference material for subsequent chapters.

The control of a dextrous hand involves many of the problems central to robotics: coordinated, coherent control of multiple and redundant degrees of freedom, the use and fusion of complex sensory information, and the development of an appropriate task-level view. Potentially, the dextrous robot hand offers an improvement in the performance of robot assembly systems since it is more versatile than standard two-fingered, parallel-jawed robot grippers. Such a hand can grasp a wider range of objects, and manipulate held objects. However, deriving the principles behind the use of the mechanism involves understanding both the physical interface between hand and object, and also the task context constraints on the hand.

In Section 1, we shall overview the current status of the dextrous hand literature, setting the scene for the development of a framework for grasping and manipulation. Section 2 introduces the basic concepts of this framework, while Section 3 supplies the details of the *set of simple grasps* [54]; a simplified grasping and manipulation system which will be used as a specific source of examples. Section 4 describes the interface between the task description and the grasping mechanism, and formalizes grasp selection.

Much of the work in this chapter followed from discussions with Thea Iberall and Michael Arbib, and was subsequently reformatted in the light of later discussions with Rukmini Vijaykumar and Subramanian Venkataraman.

Finally, in Section 5, we describe how our grasping work fits in with comparable work elsewhere, and with our goal of developing a computational model for robot programming.

§1. Previous Work in this Area

The literature to date in the dextrous hand field can be divided into a number of discrete areas. The field, in general, has a strong ***bottom-up*** tendency, for historical reasons. The initial emphasis was on the construction of appropriate hand mechanisms, such as that of Crossley and Umholtz [1977], Asada [1979], Salisbury [1982], and Jacobson et al. [1983]. As well as construction, these experimenters addressed the problem of controlling their particular hand models at the level of joint movement.

The basic requirement of the object acquisition phase of the grasping movement for any hand, is that the object be constrained to some controllable degree by contact with the hand. Salisbury [1982] considered what hand-object contact constraints were necessary to completely constrain a grasped object. The problem of choosing a series of grasp points such that the net force and torque on the object is zero is frequently called the *static* stability problem. Given friction, there will frequently be many choices of grasp configuration for a specific hand and object to impose static stability. If a grasp configuration is such that the object experiences a correcting force from the contact points whenever it is disturbed a small amount from its nominal position, then the configuration is said to be ***dynamically*** stable.

Hanafusa and Asada [1977] were among the first to consider the stability of a multi-fingered grasp. For one-DOF elastic fingers, they defined stability as the ability of the grasp configuration to counter small object displacements with correcting forces. This work has been extended by Baker et al. [1985] using a modified hand model. They investigate stability for two classes of grip configuration: Their ***triangular grip***, where a maximal inscribed circle on a 2-d polyhedral object touches the boundary at three points, with less than 180 degrees between successive points, and the points then correspond to finger positions; and their ***parallel grip***, where the maximal inscribed circle touches the boundary at two points.

Salisbury [1982] analyzes object-finger contact conditions into a number of categories and uses these to define the ***mobility*** and ***connectivity*** of a grasped object and hand. His immediate goal was not to analyze grasp stability but to determine what contact

conditions can best guarantee complete object restraint when finger joints are locked and complete object control when the finger joints are active. A grasp configuration through which arbitrary forces and moments can be exerted on the object through the contact points is called a *force closure grasp*. In such a configuration, any motion of the object will be resisted by the contact points. Nguyen [1986] describes a number of approaches to the problem of synthesizing force closure grasps. Iberall et al. [1985] discuss the concept of configuring a human hand to produce specific contact forces at 3 pairs of opposition sites on the hand; they demand as an input parameter, the object 'tagged' with some number of opposition sites, selected to facilitate future manipulation.

Both definitions of stability relate to the 'steady-state'; the hand and object as a pair. A necessary condition for this is that the hand *can* acquire the object. This means, firstly, conveying the hand to the object vicinity, and then establishing the finger contact points in such a manner that the object can eventually be grasped in the correct configuration. Fearing [1984] divides this phase into two classes: the *object priority* acquisition, where the object is considered fixed through the course of his *initial touch phase*; and the *hand priority* acquisition, where both fingers and objects move during the course of the initial touch phase. The former can be implemented by moving the fingers to their desired object contact points in such a manner that they exert zero or negligible force on the object when they initially contact. Once all fingers have contacted, the appropriate contact forces can be set up. The hand priority grasp actually entails a form of simplified manipulation; the fingers move the object while closing in on it.

Once the acquisition phase has ended and the object is stably acquired, a subsequent possible usage of the hand is for object manipulation. If the object has been gripped in a force closure grasp then it is possible to control arbitrary small motions of the object via the contact points. Salisbury [1982] developed the *grasp transform* matrix for his hand design; relating motions and forces on the grasped object to motions and forces at the finger-tips. An object is stably grasped if this matrix has full rank. Kerr [1986] discussed the role of optimally choosing *internal grasp forces* (squeezing the gripped object). Fearing [1984] addresses a simplified form of manipulation, possible without knowledge of the geometry of an object model. Lyons [1985] suggests a range of grasps is necessary, with a spectrum of manipulation abilities and object model requirements. He determines a way to go from a simplified description of the requirements of the task to an appropriate manipulation model (grasp).

One of the few papers to address the requirements of grasping from a task description is Cutkosky [1985], who develops an explicit list of *stability testing criteria* to investigate a particular grasp configuration (characterized by the arrangement of fingers on the object, and the stiffness and kinematic design of the fingers). His fundamental motivation is that the robot may choose between a number of grip configurations to determine which are the most suited in view of expected stability requirements. More recently, Li and Shastry [1986] have introduced the concept of a task-ellipsoid as a way of representing the task requirements for a grasp. They construct a quality measure which determines how *well* a particular grasp will satisfy such task requirements.

This previous work is lacking in a number of areas. Firstly, it provides no clear or uniform structure in which to see the problems of grasping and object manipulation. To date, some aspects of the area have been worked on a great deal: hand construction and stability of grasp, for example. Other areas have been only touched, or not considered at all: stability at the acquisition phase, simplified manipulation models, integration into a task-level language, phalange (as opposed to finger-tip) style grips, and the formal development of reliable joint controllers. One of the first requirements for our example domain, therefore, is that we construct a *framework* for the entire grasping and manipulation problem. This framework allows us to classify existing work and to identify the 'bare' areas.

Secondly, few, if any, hints are given as to how to relate the requirements of a particular task to the way in which the hand is used. That is, to go from the knowledge of some task domain or operation to be performed on some object, to the specification details for the dextrous hand. Grasp stability is definitely a task requirement, but it is not the only one. In this chapter, we attempt to rectify both of these problems by formulating a general framework for grasping and manipulation, and by providing explicitly for *task requirement input* in the framework. By doing this, we allow for the integration of grasping with higher-level processing such as task planning and learning.

§2. A General Model of Grasping and Manipulation

We shall consider the use of a dextrous hand at the level of *tactical grasp planner*. By this, we mean a local planning system whose scope covers a single grasp (which we will take to include subsequent manipulation). The tactical planner must interface at the

topmost level to a *strategic planner*, whose purpose is to reason about assembly order and operations. The interface to the tactical planner is task-level; via an instruction to grasp, an object identifier, and an indication of the operation to be performed. At the lower level the tactical planner interfaces to an arm and hand control system.

We start our analysis by considering the completion of a simple assembly task with a dextrous hand. We shall format an instruction list for the assembly. Directly from this, we can derive directly a level of necessary task-level input to the grasping and manipulation process. From this example task, we can also place some structure on the grasping and manipulation process, the start of our framework.

§*2.1 Example Task*

Figure 1 shows the components of the stripped down servo amplifier assembly which we shall use as our example assembly task. The completed assembly is shown in Figure 2. The operations necessary to complete this task are as follows:

1. Acquire the Base-Plate and position it at some convenient location in the workspace.
2. Acquire the Large Transformer and place it on the Base-Plate, so that the four affixing legs are aligned with the four screw holes in the Base-Plate.
3. Screw the Transformer to the Base-Plate with four small screws.
4. Acquire the Capacitor and insert it into the large hole in the Base-Plate, such that the three affixing legs on the Capacitor are aligned with the three screw holes on the Base-Plate.
5. Secure the Capacitor to the Base-Plate with three small screws.
6. Acquire one of the Side-Plates and position it so that the heat-fins point outwards from the assembly, and the two large bolt holes are aligned with the two large bolt holes in the Base-Plate.
7. Acquire and place the other Side-Plate in a similar fashion.
8. Acquire the Top-Plate and position it on top of the two Side-Plates, in such a fashion that the four large bolt holes in it are aligned with the bolt holes in the Side-Plates.
9. For each of the four bolt holes: Acquire a Large Bolt, and insert it in one of the large bolt holes in the Top-Plate, through the Side-Plate, into the Base-Plate, and screw it tight.

It is almost a trivial observation that this task consists of a number of object grasps, followed in each case by some amount of subsequent manipulation, followed eventually by releasing the object from the hand. We formalize this observation into the basis of our framework by defining a *grasp* to consist of the following:

- Configuring the hand to acquire a particular object for a particular task.

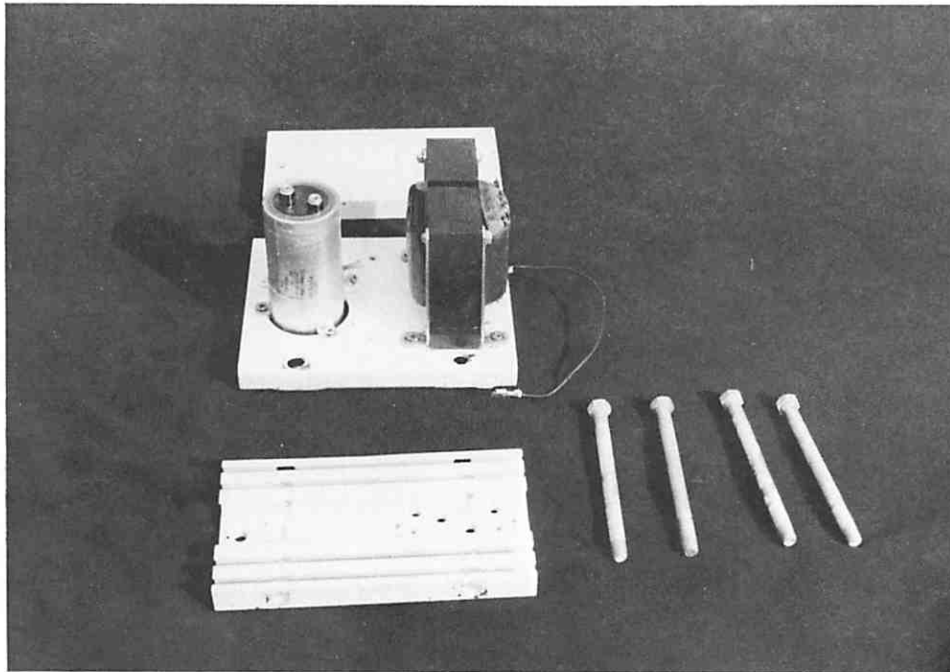


Figure 1: The Components of the Example Assembly Task.

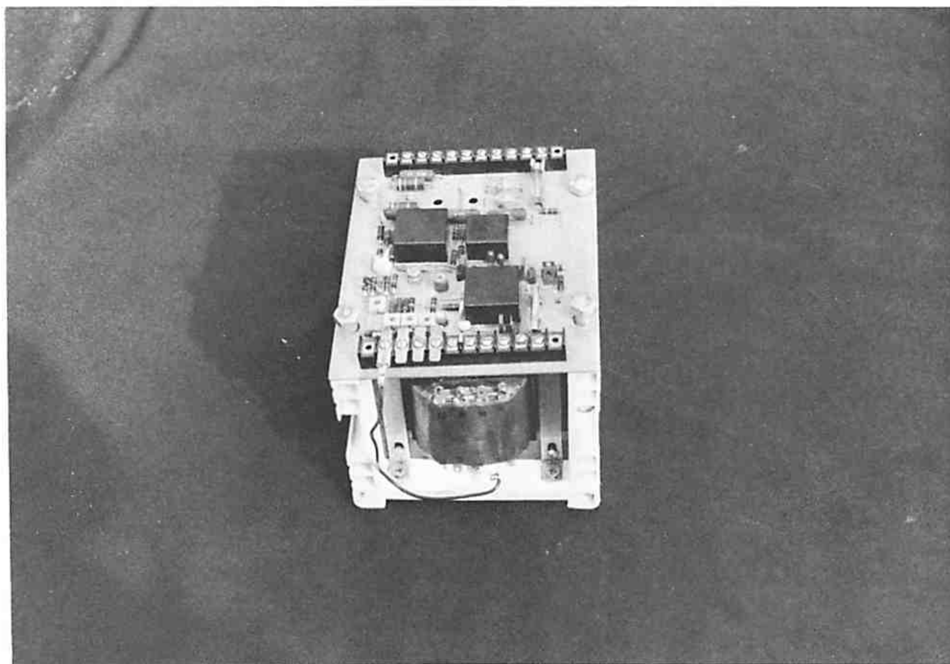


Figure 2: The Completed Example Assembly Task.

- The acquisition of the object in a stable manner.
- Manipulation of the object within the grasp; i.e., exercising some degrees of freedom associated with the grasp. Until, finally, the object is released.

So, for example, in the case of the Base-Plate: The manipulation objective for acquiring the Base-Plate is simply to place it at some central location in the workspace, which will then act as the site for the assembly. The configuration of the hand to suit the object and task can occur in parallel with the movement of the hand to a position adjacent to the target object. This arm movement, during which the grasp is maintained, illustrates some of the concurrency of arm and hand control. We shall call these the *preshape* and *reach* parts of the grasp respectively. The acquisition process can be difficult, depending on the relationship between the current pose of the object and the desired grasp pose; it may have to undergo some initial manipulation. Once grasped, the Base-Plate must be moved approximately to some central location in the workspace, and placed it there.

In the case of one of the bolts: There is a complex manipulation objective for acquiring a bolt. It must be inserted into a bolt-hole in the Top-Plate, and then screwed down. The hand configuration chosen must facilitate fine-motion manipulation characteristic of the insertion task. The bolt must be acquired in such a manner that the insertion can proceed (e.g., its insertion axis must not be blocked).

It is clear from these examples that the *grasp* must be based on the constraints of desired subsequent manipulation, as well as object characteristics. Within this assembly task we can identify a number of classes of assembly operation. Each class is typified by certain global choices such as position versus force control or by particular stereotypical details of the operation or task geometry. One class is the acquisition of an object and its *placement* at some (possibly precise) location (e.g., the Base-Place). Another class of operation is the acquisition of an object in order to *insert* it into some container (e.g., the motor or the bolts). A third class of motion is the screwing down of the bolts once they have been inserted. If we consider the details of trying to align the Top-Plate, or perhaps the motor with some screw holes, then another class of motion is sliding an object to *align* it on a surface. We can extend this still further by considering the use of tools; a hammer must be swung down to hit its target – this is a particular ballistic class of manipulation. It is clear that, in general, there may be many such operation classes. The task-level input to our grasping and manipulation framework will, therefore, consist of the following: an *object description* (we shall explore its contents later), and

an indication of the *operation* class to be performed (one of a small subset of operation classes we shall define).

§2.2 *A Framework for Grasping*

In this section, we shall formalize our observations into a framework for grasping and manipulation. This framework specifies the structure of the tactical grasp planner mentioned earlier. In general, there are cases when tactical considerations may be insufficient to complete some part of the grasp; we formalize a number of such *error* cases.

The ways in which a complex robot system, such as a dextrous hand, can interact with the environment are many. We partition the set of all such interactions into a number of *domains of interaction*, each with specific characteristics. These specific characteristics can then be chosen to suit the desired operation class and the target object description. Within each such domain, hand control is accomplished by a specific *approximation* to the full power of the hand. Correct choice of domain will mean that the aspects of the hand emphasized by the approximation will be those important for this object and operation class, whereas the aspects ignored will be those that are not relevant. We define an abstraction called a *Grasp* which will embody the approximate model for a domain of interaction.

We start by assuming a hand coordinate system in the wrist of the hand model (model i) $\mathbf{F}_{iw}$ (see Figure 3(a)), and an object-centered coordinate system $\mathbf{F}_o$, (see Figure 3(b)) with its origin at the center of mass of the object. In the next section, the grasp components are described in some detail, and this is followed by discussion of the grasp selection and parameterization mechanisms.

The Grasp Components

A grasp $\mathbf{G}$ is a tuple $(\mathbf{P}_g, \mathbf{A}_g, \mathbf{M}_g)$ consisting of:

Preshape Component,P_g: Once the operation class and characteristics of the target object are known, a grasp can be chosen suited specifically to these. The preshape component *configures* the hand in preparation for object acquisition. The preshape configuration is, of course, a function of a particular grasp; but it is also a function of the object shape and size (see the section below on grasp parameterization). We shall consider the *reach* as part of the preshape component; it conveys the hand to the vicinity of the object, and orients it to the object pose.

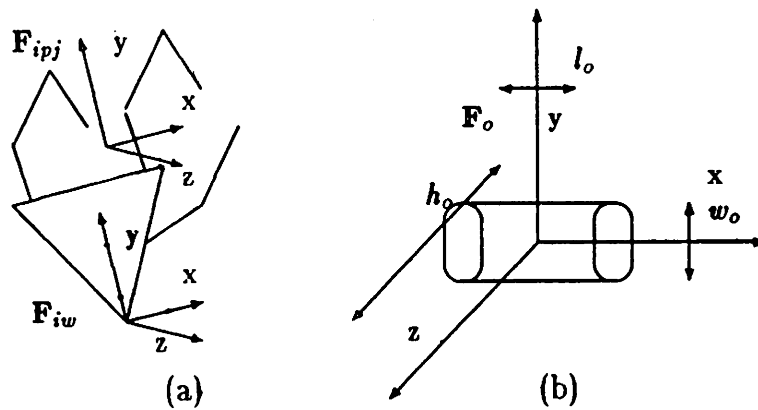


Figure 3: Wrist (a) and Object (b) Centered Frames.

A typical hand model is shown in (a); $\mathbf{F}_{iw}$ for the model is based in the wrist, and $\mathbf{F}_{ipj}$ is shown at some arbitrary point with respect to $\mathbf{F}_{iw}$. The object coordinate system is shown in (b); the length, l_o, width, w_o, and height, h_o, parameters are object measurements along the x, y and z axes respectively

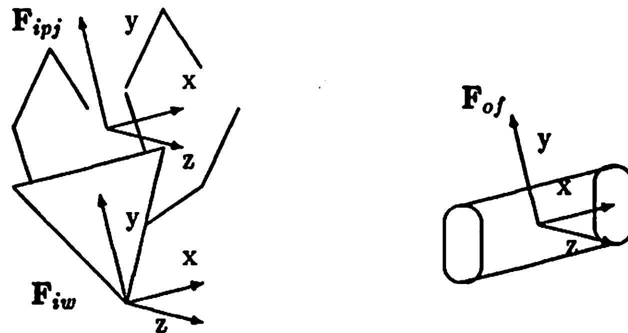


Figure 4: The Kinematics of the Reach.

We assume a coordinate system (frame) $\mathbf{F}_{iw}$ (i=(s)alisbury, (h)uman), based in the wrist of each hand model. Relative to this frame, we can locate, for any grasp, that point where the center of mass of the object should eventually lie. For each grasp, this will be some fixed position and rotation offset in the wrist frame. Additionally, it is necessary to define an *approach axis* for the grasp, along which the hand must approach the object in order for the object to reach its desired final position. To simplify this process, we define a grasp preshape coordinate frame, $\mathbf{F}_{ipj}$ (j =grasptype), whose origin is at that point in $\mathbf{F}_{iw}$ where we should like the object center of mass to lie, and whose $z - axis$ will define the approach axis for the grasp. For any object and grasp, $\mathbf{F}_{ipj}$ can be described as some *fixed* translation and rotation of $\mathbf{F}_{iw}$.

We assume an intrinsic object coordinate system $\mathbf{F}_o$. An *input parameter* to our grasping system will be a fixed homogeneous transformation matrix, $\mathbf{T}_f$, used to define in what *pose* the object is to be acquired. $\mathbf{T}_f$ defines, therefore, a fixed transformation of $\mathbf{F}_o$. The approach axis of any grasp will be along the $z - axes$ of both $\mathbf{F}_{ipj}$, and $\mathbf{F}_{of} \triangleq \mathbf{F}_o\mathbf{T}_f$. This situation is described in Figure 4.

In the reach, F_{gp} is oriented to F_{of} (by wrist movements), and displaced a distance δ along the z-axis of F_{of} (Figure 4). The value of δ is an 'elastic' constant, indicating how close the hand can safely approach the object without disturbing it. We shall assume a *constant value* of δ throughout, but it is feasible that δ could be related to factors such as the stability of the assembly, etc.

Failure at the preshape phase can only occur if the hand cannot complete the approach to the object; we shall refer to this as **approach failure.**

Acquisition Component, A_g: Once the hand has been preshaped, oriented to the object and conveyed to within an appropriate distance, the acquisition component begins. There are three cases with which the acquisition component may have to deal:

- Simple Acquisition: By translating the wrist a distance δ along the z-axis of F_{of}, making, thus, the origin of $\mathbf{F}_{gp}$ and $\mathbf{F}_{of}$ coincident, the fingers can immediately be placed in position so that they can reach their desired contact points on the object.

- Two-Phase Acquisition: The object cannot be acquired in the manner above. However, the object *can* be acquired in some temporary grasp, and manipulated to its final desired configuration.

- Constrained Acquisition: The choice of grasp, and in particular the preshape, has made it impossible for the hand to approach the object. Acquisition can only proceed by modifying the preshape in some fashion, or by choosing a different grasp.

Apart from the general principle of *stable acquisition* (where the object must not move out of the grasp during acquisition), we identify separate *acquisition strategies* with each grasp. These strategies fall into three classes, under the headings above. In the next section, when we develop a particular set of grasps, we shall restrict our attention, for simplicity, to the *first case only*, Simple Acquisition.

The acquisition component for each grasp in the SSG system will be a variation on the following description: Fingers are moved in to contact at low stiffnesses, so that the object experiences a negligible disturbance force when contact occurs. This is similar to what Fearing calls his *Object Priority Grasp* [24]. Once all contacts are in place, (the number of contacts and termination conditions will vary according to the grasp), contact forces to secure the object must be calculated. Once all links have made contact, knowledge of exact finger placement can be used to set the conditions for static equilibrium (this process much simplified by the presence of static friction), e.g., Salisbury [1982], Kerr and Roth [1986]. If r_x is the vector from the origin of $\mathbf{F}_{ipj}$ to an object contact point, and f_x is the force exerted at that contact point then static equilibrium demands (where

F_y is any force exerted on the body, e.g., gravity):

$$\sum_x r_x \times f_x = 0, \qquad \sum_y F_y = 0$$

We identify two error cases which can be diagnosed during acquisition. If, due to some unmodeled features of the object, or some unmodeled forces in the environment, the stable acquisition principle is violated and the object slips out of the grasp, this is called **capture failure**. If the fingers could not assume their final desired position on the object, this is called **contact failure**. In more practical terms, if the fingers achieve contact before they touch the object, this will be classified as a contact failure. If they achieve contact at some finger-tip separation which is smaller than the object width, then this is considered a capture failure.

When terminated, the acquisition component has set up a particular hand-object configuration which suits the model of the hand embodied in the grasp, and is used in the manipulation component.

Manipulation Component, M_g: The manipulation component is, in some sense, the key component of the grasp, since it describes how the grasp can be used to complete an assembly operation. Each grasp has a different manipulation component – a different model of how the hand can effect motions on the grasped object, both in terms of the degrees of freedom available, and in terms of the precision with which these can be controlled.

The simplest case of a manipulation component is the *null* component: The hand acquires the object, but any subsequent manipulation can only be done by moving the wrist around – no finger manipulation is possible. This is suited to a pick-and-place style task, and is exactly what is available from a standard two-fingered robot gripper. The most complex case of the manipulation component is when arbitrary forces and moments can be effected by the hand on the object. In between the two is a spectrum of manipulation models which represent an economical mixture of optimal hand usage for a given task with a simple task-related interface for the programmer. The programmer does not deal with the full power of the hand (and hence the full specification of all its parameters), but instead, deals with a task-related abstraction (the grasp) which provides degrees of freedom and precision specifically relevant to the task. From a computational point of view, the task-related model may be more economical than the full hand model.

Failure at the manipulation level can occur if the object slips out of the grasp (**stability failure**), and, also, if the manipulation cannot be completed for some reason (**manipulation failure**).

Grasp Selection and Parameterization

Grasp Selection: We identify specific characteristics with each domain of interaction, each grasp. The task requirements and target object description can then be used to choose an appropriate domain of interaction based on these characteristics. A similar concept is suggested by Cutkosky [1985] for selecting a grip configuration which is best suited for a task. However, Cutkosky considers only *stability* metrics, and he selects the gripping configuration *only*.

The first step in realizing this approach is to determine useful characteristics with which to partition hand usage into specific domains of interaction. In our informal example in the previous section, we made a start on this – identifying particular assembly operations such as *Insert* and *Place*, which delineate particular sub-problems a grasp can be specialized to solve. In the next section, we shall identify specific task requirements which can be abstracted from such an operation.

Object characteristics also play an important role in grasp selection. Size, shape and mass classifications can be used to determine if a target object will suit the domain of interaction represented by a particular grasp. Once the grasp has been chosen, more detailed object knowledge is necessary to parameterize the grasp components.

Grasp Parameterization and Object Models: Once a given grasp has been chosen, the specific description of the target object can be used to particularize the grasp components for the object. The detail with which the object model is represented will determine how well the grasp can be particularized. For example, if detailed object geometry is known, it may be possible to analyze the object for stable grasp points. If only coarse object geometry is known, then a less precise approach to grasp stability must be taken. An important tool in grasp parameterization, and in grasp description, is the *virtual finger*.

Virtual Fingers: It was noted by Arbib et al. [1983] that the same "grasp" may use different physical fingers, depending on object characteristics. They noted that the mug-grasp can be described using three components: a downward force from above the handle; an upward force from within the handle; and a third force to stabilize the handle from below. However, different size mugs would require these three components to be

supplied by *different* groups of fingers; thus the grasp must be defined independently of physical fingers, and only mapped onto physical fingers when the details of the object are known. They called these logical units *virtual fingers*. The concept of the virtual finger provides independence from both physical hand structure and from particular object models, since consideration of what physical resources the hand offers, as well as the object characteristics can be taken into account when the virtual finger mapping is done.

All grasp components are described in terms of virtual fingers (VF). Each grasp has its own integral VF selection criteria to determine how many VF are required to describe the grasp and, also, its own integral VF mapping functions, to determine how virtual fingers relate to physical fingers on a specific hand model.

§3. A Simple Set of Grasps

In this section, a particular example of the grasping and manipulation framework discussed in the last section is developed. A small but useful set of grasps, called the *simple set of grasps* or SSG for short, is constructed. The SSG system is a simplified model; it considers little of the hand and task dynamics. Even so, it contributes to the literature in that it is a multiple-grasp model with task-requirement input, based on hand-independent descriptions, and complete enough to integrate into an assembly program. Its main fault is that it could be more detailed; however, if it were, it would be too big to consider as an example for our computational model.

We have already mentioned that an acceptable set of grasps is one which provides a range of different manipulation models and abilities. In this way, a grasp can be chosen for some assembly operation which makes best use of the hand and also provides a simplified task-related interface to the hand for the programmer.

Grasp Functionality in SSG: We shall first identify the task-related functional characteristics on which the SSG system will be based. Earlier we pointed out how certain typical subtasks occur in any assembly operation; these subtasks, such as *Insert* and *Place*, have characteristic *geometries* and *control requirements*. We shall build the functional characteristics of the SSG system one step below and simpler than these subtasks. We identify two functional characteristics of a grasp: its ability to finely manipulate a grasped object and its ability to grip the object securely.

In an assembly task, objects will need to be grasped differently to provide differing

amounts of precision manipulation. In addition, precision of manipulation is not *at all* a goal for some types of grasping; instead, the object must be gripped as rigidly as possible. Some tasks, such as wielding a hammer, may require firm affixment. Others, such as placing a small object, may not require such firm affixment. We shall take a binary indication of the amount of precision manipulation ($P, \neg P$; *precision, not precision*) and the amount of firm affixment ($F, \neg F$; *firm, not firm*) as the *functional input* to our grasp selection process. In order to keep the SSG system simple, we shall assume that the mapping from a subtask such as *Insert* or *Place* to our functional requirements can be, and has been, done.

An Object Model in the SSG system: We shall not assume detailed object knowledge. For grasp selection the following classifications are provided: Two important object characteristics are *size* and *shape*. Objects which are large with respect to the hand cannot be lifted with just a finger-tip grip. We roughly classify size into *large* and *small* with respect to the hand. Objects which have flat faces can often be grasped better in certain hand configurations than those with curved surfaces, and vice versa. We divide objects into *round* or *flat* depending on whether their flat edges or curved edges are the dominant characteristics. We also define objects as long or short with respect to their aspect ratio[1].

For grasp parameterization, we shall assume basic measurements of the object; length, width and height parameters, l_o, w_o, h_o (see Figure 3). We place a constraint on the analysis at this point — the object description is assumed to refer to the complete object *or* to the grasp-site on the complete object, whichever is appropriate. Grasp-site is an *input parameter* to our model, and one which allows for the direction of grasping activity by a high-level process [89]. In this way, a large object may be grasped at sites distant from its center of mass, when this is necessitated by higher-level concerns (e.g., tool-usage, stability requirements, etc.).

§3.1 *Outline of the SSG Grasps*

Our basic approach is to design a small set of grasps which have a spectrum of manipulation models and grasp securities (our two functional characteristics). We describe a set of three grasps: a grasp which has *no* manipulation ability but very secure affixment; a grasp which has the ability of arbitrary object motion, but minimal security; and a

[1]In a later section, we discuss how to embody *continuous*, rather than discrete, characteristics.

grasp which offers some manipulation ability and some security. The first two grasps represent opposite ends of the spectrum, and will probably be present in every useful set of grasps. In the third grasp, we show that it is possible to construct a grasp which has a simplified manipulation model. These three grasps compose the *set of simple grasps.* In more detail they are the following:

An Encompass Grasp: has the characteristics that the fingers are used to envelop the object in a compliant fashion. The goal of this grasp is to attach the object rigidly to the hand. Manipulation can only occur through movements of the wrist – the fingers operate just to grip the object securely.

A Precision Grasp: has the characteristics that the finger-tips are positioned on the object in such a way that the object is restrained, yet arbitrary fine motions can be imparted to it by the hand configuration. Manipulation is primarily a finger-related function, the wrist being used only for gross motions.

A Lateral Grasp: has the characteristics that the object is gripped not with the finger-tips, but with the inner planar surface of the digits. Unlike the encompass grasp, the fingers do not envelop the object, they merely hold it in a vise-like grip, similar to that of a two-fingered gripper. Using the phalanges, rather than the finger-tips, overconstrains the object at the contact points. Yet, since the fingers are not wrapped around the object, *some* joints are free to effect a measure of precision manipulation.

§3.2 *Kinematic Details of the SSG grasps*

In this section, we describe the *preshape, gripping and manipulation components* for the *Encompass, Lateral* and *Precision* grasps. The analysis throughout is primarily kinematic. The main thrust of this work is to form a framework for grasping and manipulation, with the ability to input some form of task requirements to the grasping process. We have constructed the SSG system as an example of the use of this framework. This approach is too simplistic when dealing with issues of stable grasping and manipulation; however, these areas are *not* the main thrust of this work, and while a kinematic version of finger-tip manipulation is outlined, it is simply included for completeness. We start by defining some terminology.

Terminology and Notation: A subscripted, uppercase **F** denotes a *coordinate frame.* A subscripted uppercase **T** denotes a *homogeneous transformation matrix.* We use the terms Roll, Pitch, and Yaw [70] for the rotations ψ, θ, ϕ around the x, y and z axes

respectively. The order of rotation is defined by:

$$RPY(\phi,\theta,\psi) = ROT(z,\phi)ROT(y,\theta)ROT(x,\psi) \tag{II.1}$$

where $ROT(a,b)$ is a homogeneous rotation matrix around axis a of b radians.

A base coordinate frame is defined in the *wrist of each hand model*; $\mathbf{F}_{hw}$ for the Human hand model, and $\mathbf{F}_{sw}$, for the Salisbury hand model (Figure 5). Rather than consider any particular arm, the wrist-based coordinate frame is related to the world coordinate frame $\mathbf{F}_b$ by an arbitrary arm transformation matrix $\mathbf{T}_{bw}$:

$$\mathbf{F}_{iw} \triangleq \mathbf{F}_b\mathbf{T}_{bw}, \quad i = s, h \tag{II.2}$$

The link and degrees of freedom (DOF) numbering used in this paper is presented in Figure 5. Both hand models are considered equipped with wrists with three rotational DOF.

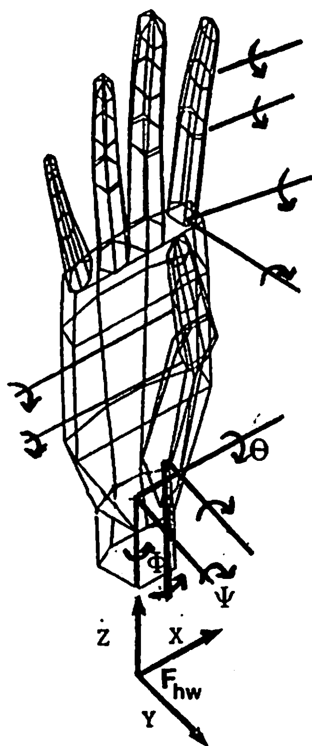
When the grasp components are defined later, a set of grasp-specific coordinate frames will be associated with each one. Each object is placed within its own local coordinate frame, $\mathbf{F}_o$, for some object, O (Figure 3). Associated with a particular grasp on a particular object is a *grasp frame*, $\mathbf{F}_{of}$, defined in terms of the object frame as:

$$\mathbf{F}_{of} \triangleq \mathbf{F}_o\mathbf{T}_f \tag{II.3}$$

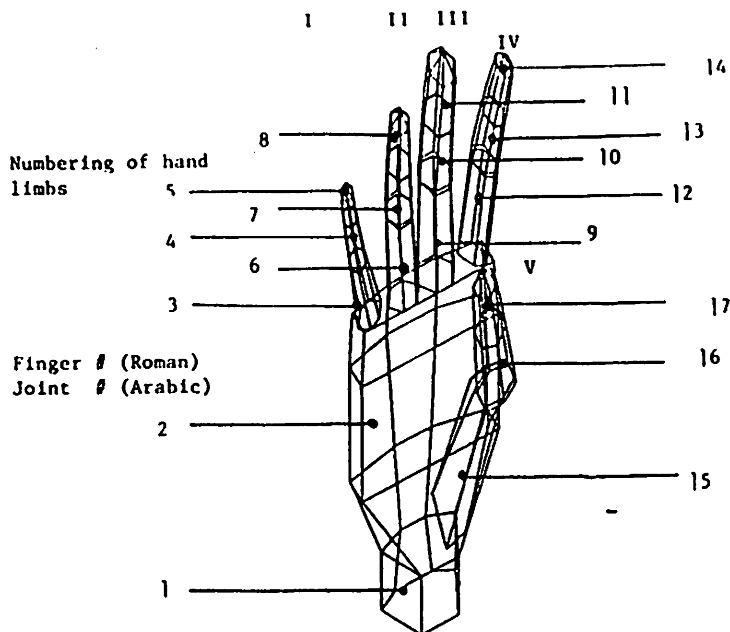
In this case, $\mathbf{T}_f$ defines how the grasp is to be applied to the object (i.e., grasp site selection). We shall let $\mathbf{T}_f$ be an *input parameter* to the SSG system.

The following $\mathbf{F}_o$ conventions are assumed for all objects. The origin of $\mathbf{F}_o$ is placed at the center of mass (COM) for the object, with the x-axis along the long axis of the object (if one exists). The object will be approached along the z-axis of $\mathbf{F}_{of}$, the final orientation of the object within the hand, when gripped, is grasp-specific.

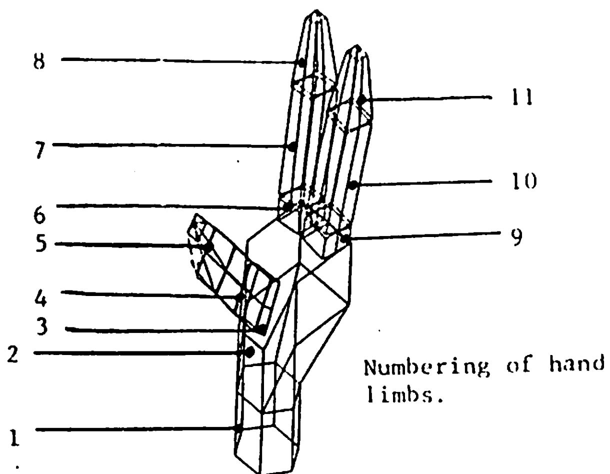
Virtual Fingers: We shall define a virtual finger as a particular *logical* mechanism, having an associated *mapping set*. The ith virtual finger is denoted by VF_i and its mapping set by $VFset_i$, containing the indices of fingers considered part of VF_i. The translation from VF_i actions to actions of some set of physical fingers is a two-step process: Firstly, the VF_i is translated to some set of physical fingers by $VFset_i$; secondly, VF_i actions are translated into actions of *each* physical finger. This second step is the interface of the SSG system with a particular physical finger model. We will store physical hand



DOF of the human hand model.



DOF of Salisbury hand model.



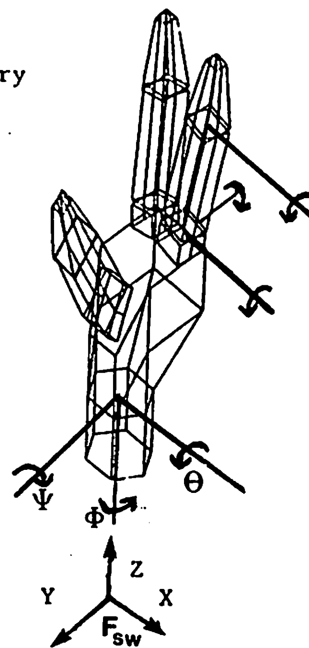


Figure 5: Hand Models: DOF and Link Numbering.

information in two specific mapping mechanisms, which we will use at the second step of the VF mapping.

Each grasp can have different VF mapping functions. In translating VF actions to a particular physical finger on a particular hand model, the grasp component can control what physical finger properties to emphasize; i.e., what finger or hand property *must* be the same across all hand models. Iberall et al. [1985] considers just a single VF mapping mechanism throughout grasping; however, she is working with only a single hand model, the human hand. In order to stress the hand independence of the SSG system, we design appropriate VF mapping mechanisms for both human and Salisbury hand models.

We construct two kinds of mapping mechanisms for the second step of the VF mapping process: one is an angular specification and the other is the end-point specification. The angular method involves storing an empirical relationship between finger-tip and thumb-tip separations versus finger and thumb angles. End-point specification involves the *forward* and *inverse kinematic* solution for the finger [70]. The angular specification is used to preshape the hand so the object can fit into it, and as a feedforward value for surface contact in the acquisition component. End-point position is used to control the object-finger contact points for object manipulation.

Angular Separation Equations:

A basic grasp characteristic for any hand model is the separation between thumb-tip and finger-tip. In this section, we derive an empirical relationship between the finger angles and the finger-tip/thumb-tip separation, for each finger and each hand model. This relationship is important because it allows us to go quickly from an object-oriented measure, the object width value, to the finger angles in order to permit the object to fit into the hand. We call this empirical relationship the *angular separation equations*, and we shall use it frequently in mapping from virtual finger actions to physical finger actions.

We derive the angular separation equation for a given finger on a given hand model as follows: Starting from some finger and thumb configuration in which the finger-tip and thumb-tip just touch, and in which no joint is close to one of the limits of its operation[2], we measure the separation between thumb-tip and finger-tip, while iteratively increasing all the θ joint angles of the finger and thumb by some fixed amount $\delta (= 1)$. The graph of separation versus total joint increment Δ for the Salisbury hand is shown in Figure

[2]Because this increases the non-linearity in the relationship.

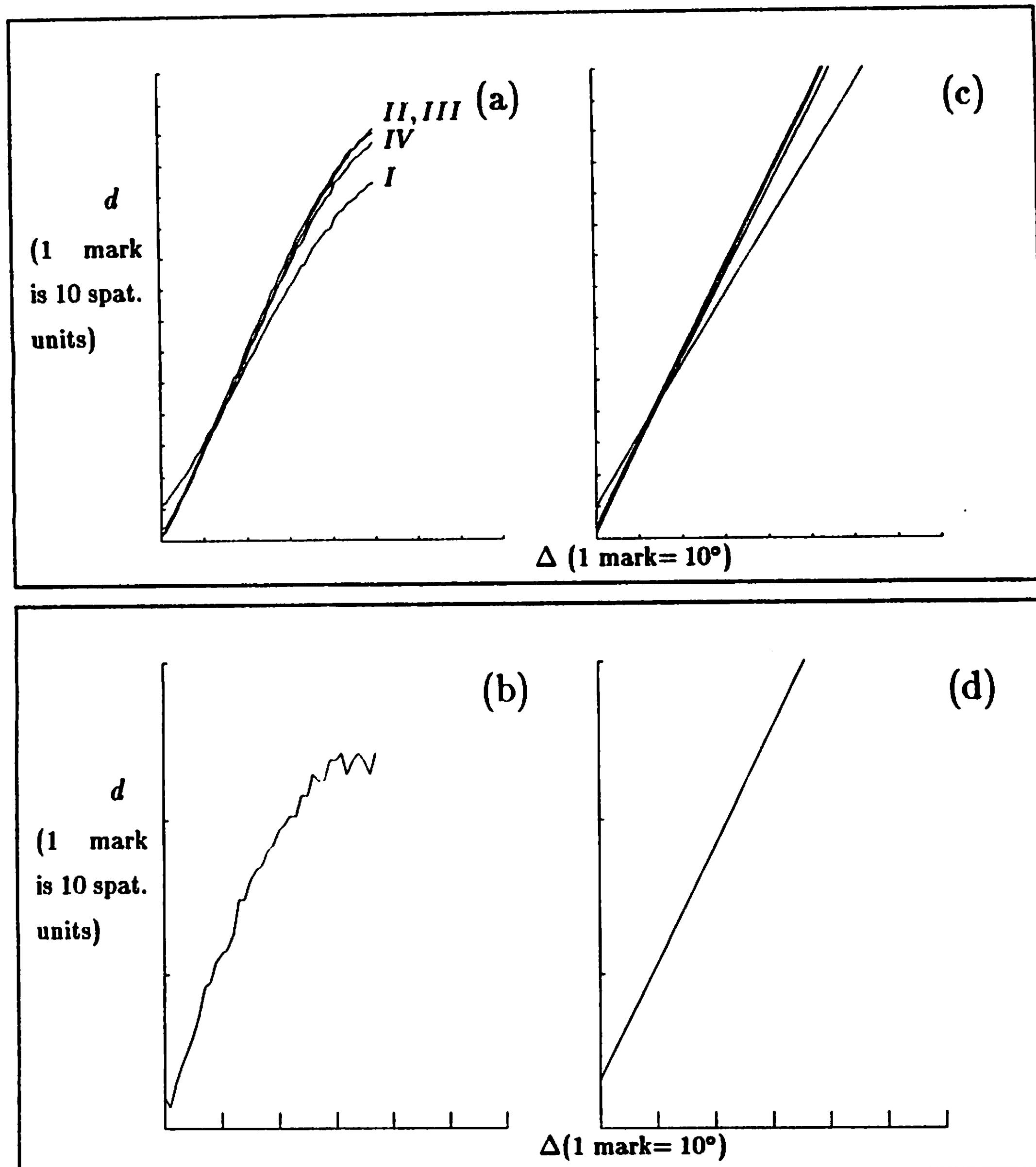


Figure 6: Graphs for Finger-tip/Thumb-tip Separation Equations.

Graphs of finger-tip/thumb-tip separation (d) against equal increments (Δ) to all angles (from the initial positions in Table 1), for both human (a) and Salisbury (b) hand models. Least-square linear fits to these graphs are presented in (c) and (d).

Salisbury Hand Model:

Finger Number	Finger Base	ψ_2	ψ_3	offset c	slope μ
1.	6,30,19	-30.0	-50.0	3.1	0.73
2.	-6,30,19	-30.0	-50.0	3.1	0.73
3.	0,22,-4.5	35.0	65.0		—

Human Hand Model:

Finger Number	Finger Base	ψ_1	ψ_2	ψ_3	offset c	Slope μ
1.	-17,69,0	60.0	35.0	35.0	1.94	2.76
2.	-6,69,0	60.0	30.0	40.0	3.36	2.66
3.	5,69,0	60.0	35.0	45.0	3.71	2.74
4.	17,69,0	60.0	35.0	35.0	10.2	2.23
5.	6,0,-5	20.0	50.0	-40.0	—	—

Table 1: Linear Angular Separation Equations for Both Hand Models.

The angular starting positions of each finger and thumb, and the slope and offset values of the least-square fit for the graph of finger-tip/thumb-tip separation versus equal increments to all joint angles.

6(b). Note that it is non-linear, initially, and towards the end. At the same time, it is reasonably linear over the important working range for finger-tip/thumb-tip separations, however (for increments of about 1 degree to about 22 degrees in all joints). Note that there is only one graph for the Salisbury hand, since both fingers are identical. Figure 6(a) shows this relationship for the four fingers of the human hand model. Note that it, too, is linear over a large range (5 degrees to about 30 degrees). The initial joint angles for which these graphs were measured are given in Table 1. Parts (c) and (d) of Figure 6 show a least-square linear fit for the graphs in parts (a) and (b). The slope and offset values for these fits are given in Table 1. We shall use this linear approximation in our angular separation equations.

End-point: We make use of both the forward and inverse kinematic solutions of the physical finger. The forward kinematics are used to select finger separations which will result in the expected contact points being distributed around the object circumference: Given a set of θ angles for the finger from the angular separation equations, a base joint ψ value must be found which effects some given finger separation. With θ angles fixed, the finger can be considered a single link. To determine some separation, ℓ, between a finger i and its adjacent neighbor, $i+1$, we start by subtracting the current end-point separation ℓ' from ℓ. The forward kinematic solution allows us to generate these end-point positions. After subtraction, we calculate the additional angle which the end-point must traverse to achieve the remaining separation $\ell - \ell'$, using the arc equation $\psi = \frac{\ell - \ell'}{fl}$, where fl is the finger length (base position to end-point, again from forward kinematics).

Finger separation may be necessary between adjacent fingers within a VF, to give a VF a 'finger' width ℓ_{vf}; and between adjacent VF (between adjacent fingers which are in adjacent VFs) s_{vf}. We set the *convention* that each finger examines the finger with the next highest index number to determine its separation angle. In order to rule out the effects of fingers moving while ψ is being calculated, we demand that ψ is calculated continuously throughout the preshape. A finger, i, determines whether to apply a separation ℓ_{vf} or s_{vf} by evaluating: $i \in VFset_k \wedge (i+1) \in VFset_k$, then use ℓ_{vf}. The thumb is also covered by this strategy; however, it works out its angle relative to the first finger of the hand (since the thumb has the highest finger index in both models). Notice that this convention will limit $VFset$ to sets of adjacent fingers.

The inverse kinematic solution is used to determine the joint angles which will produce

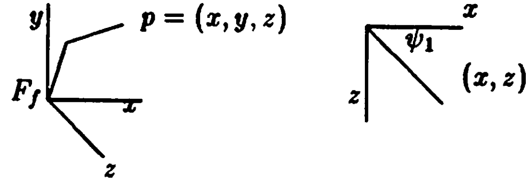


Figure 7: Geometrically-Reasoned Inverse Kinematic Solution.

a given finger end-point position. We note that if one of the θ angles for the human hand model is frozen, then the finger construction is identical (except for lengths) to the Salisbury hand model. We shall take advantage of this by using the same inverse solution form for both models. We shall not generate the inverse kinematics by symbolically inverting the forward kinematics [70], but will reason it geometrically.

Consider a coordinate system in the base of each 3-link, 3-DOF finger, as in Figure 7, F_f. To reach $p = (x, y, z)$ in this frame, the angles θ_2 and θ_3 produce a triangle, whose third side is a line drawn from the base of the finger to the point p; the length is, therefore, $\mid p \mid$. All three lengths are known in this triangle. However, note that there are two possible angular solutions. In deference to the human hand model, we shall take the convention that θ_3 must always be positive; this is simply a convention taken to select a unique angular solution for a point p. Let ρ be the angle between the line from the origin to p and the $x - z$ plane. The cosine rule can be used to generate values for the θ angles in terms of the lengths $\mid p \mid, \ell_2, \ell_3$ and the angle ρ, in terms of ARCCOS. The solutions are:

$$\begin{aligned} COS(\pi - \theta_3) &= \frac{|p|^2 - \ell_2^2 - \ell_3^2}{-2\ell_2\ell_3} \\ COS(\tfrac{\pi}{2} - \theta_2 - \rho) &= \frac{\ell_3^2 - |p|^2 - \ell_2^2}{-2\ell_2|p|} \end{aligned} \tag{II.4}$$

We can use a tangent half-angle substitution to get these angles in terms of ARCTAN: If $COS(a) = x$, then $2a = ARCTAN(\pm\sqrt{\frac{1-x}{1+x}})$. The angle ψ_1 must line the finger up with the point p in the $x - z$ plane of F_f; hence, $\psi_i = ARCTAN(x/z)$.

<u>Preshape Coordinate-Frame</u>: Each grasp preshape has a preshape coordinate frame; $\mathbf{F}_{ipj}$, where $i = h, s$ and $j = e, l, p$, specific for the Encompass (e), Lateral (l), and Precision (p) grasps. $\mathbf{F}_{ipj}$ is set up to *facilitate* the subsequent gripping operation and is defined relative to $\mathbf{F}_{iw}$ by the preshape matrix $\mathbf{T}_{ipj}$.

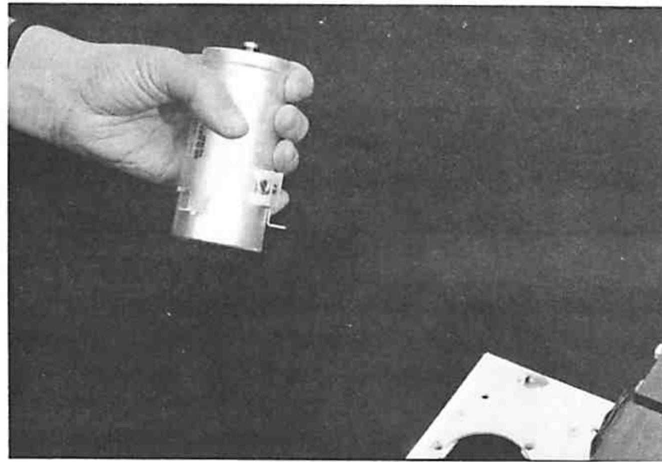
The Reach; The ***preshape phase*** consists of the preshaping of the hand, and the extension and orientation of the hand to the object vicinity. This latter stage is called the ***reach***, where, as we have described: $\mathbf{F}_{ipj}$ is a coordinate frame fixed with respect to the wrist for a given grasp and object, and $\mathbf{F}_{of}$ is a frame fixed with respect to the object, then the reach has four (standard) steps:

1. Align the axes of $\mathbf{F}_{ipj}$ to $\mathbf{F}_{of}$ using the wrist DOF.
2. Position $\mathbf{F}_{ipj}$ at δ offset on the $z-axis$ of $\mathbf{F}_{of}$ (along the approach axis) using the arm DOF.
3. From the offset δ on the $z-axis$, move the hand so that the origin of $\mathbf{F}_{ipj}$ coincides with the origin of $\mathbf{F}_{of}$.
4. Trigger the acquisition phase.

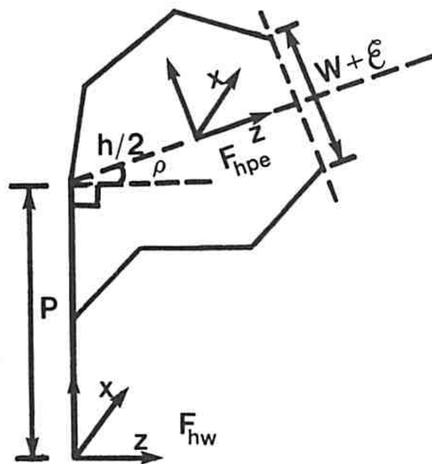
In the next three sections, we shall describe the Encompass, Lateral and Precision grasps. As a descriptive tool, we employ photographs of a human hand assuming a grasp configuration typical for each of our grasps. It is important to remember that a grasp is defined by its preshape, acquisition and manipulation components, and not ***just*** by the kinematic configuration of the preshape, or a grip, on an object (which is all a photograph can illustrate).

§3.3 *Encompass Grasp: (P_e, A_e, M_e)*

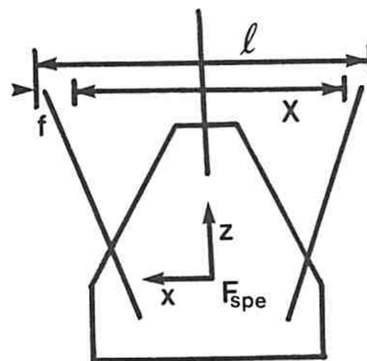
The Encompass grasp provides grip security by encompassing and enveloping the object with the fingers (see Figure 8(a)), thus losing any ability to manipulate the object with the finger-tips. The preshape sets up the hand to facilitate the secure gripping of the objects. Termination of the reach will have placed the COM of the object (origin of $\mathbf{F}_{of}$) at the origin of the $\mathbf{F}_{ipe}$ frame. Given the fact that specific details of the object are not known, a good ball park estimate of finger placement to facilitate secure gripping is to distribute the fingers evenly around the object length, about the origin of $\mathbf{F}_{ipe}$ – fingers on one side of the object, and the thumb on the other. For ***static equilibrium***, there must be ***no net force or moment*** on the object. For ***stable grasping***, the grip must be able to counter small displacements of the object with correcting forces [28, 21]. The standard approach to this is to choose finger placements on the object surface such that



(a)



(b) Placement of $\mathbf{F}_{ipe}$: This process is the same for both hand models, but the diagram shows the human hand model. Finger-thumb separation $w+\varepsilon$ is set by the angular separation equations. The $z-axis$ of $\mathbf{F}_{ipe}$ is along the line from the top of the palm to the midpoint of the line joining the finger and thumb-tips. The origin is a distance $\frac{h}{2}$ along this line. The $x-axis$ is parallel to that of $\mathbf{F}_{iw}$, and the x offset is 0.



(c) Interfinger Separation: Shown here for the Salisbury hand model (looking from overhead) – note placement of $\mathbf{F}_{spe}$. Fingers are separated to a distance $\frac{l-f}{n-1} - f$, where l is the object length, n is the number of fingers, and f is the finger width.

Figure 8: The Encompass Grasp.

that stability is assured. The approach taken by the Encompass grasp is less dependent on object knowledge, and more dependent on the multiplicity of finger-object contacts, in order to yield a stable configuration. However, the approach supplies no ***guarantee*** of stability; objects requiring special handling can be dealt with by careful placement of the $\mathbf{F}_{of}$ frame (deriving $\mathbf{T}_f$ to suit the situation), with techniques similar to [28].

Preshape Component

The Encompass preshape distributes the fingers on a cylindrical volume centered at $\mathbf{F}_{ipe}$, around a 'virtual' object, foreshadowing the actual object in configuration and size. Two important conditions must be met in this preshape: There must be enough finger-tip/thumb-tip separation for the object to enter the hand, and the fingers must be distributed along the length of the object. The thumb will oppose the fingers and should be centered with respect to object length. The Encompass grasp is a ***two virtual finger task***; VF_1 describes the actions of the thumb, and VF_2 describes the actions of the other fingers[3]. We can relate the number of fingers assigned to a VF (in particular, VF_2 in this grasp) to the object length: The larger the object, the more fingers should be assigned to VF_2. For a particular hand, we summarize this relationship in the function ν; For a given hand model, for virtual finger k, $\nu_k(l) = (a, b)$ (where l is object length), i.e., $VFset_k = \{a, \ldots, b\}$. For example, in the Encompass preshape $\nu_1(l) = (V, \ldots, V)$ for the human hand model.

For each physical finger in $VFset_2$, we use the angular separation equations to go from object width w (see Figure 3) to a set of finger angles. We set the finger-tip/thumb-tip separation to be the object width plus some ***margin of error*** ε. The angular separation equations will specify all the finger θ angles. With any virtual finger, we use the interfinger separation algorithm on page 27. If the object length is l, the fingers of VF_2 must have an interfinger spacing of X, where $l = (n-1)*(X+f)+f$, where n is the number of fingers in VF_2 and f is each finger width.

We place the origin of $\mathbf{F}_{ipe}$ at the COM of the 'virtual' object, and orient it with respect to $\mathbf{F}_{iw}$ as follows: With reference to Figure 8(b), draw an imaginary line from the top of the palm to the midpoint of a line joining the finger and thumb-tips (dashed

[3]If the hand has no distinct thumb element, then it is feasible that this grasp could be described as a one VF task.

lines); this will be the approach axis for the grasp, and the $z-axis$ of $\mathbf{F}_{ipe}$. The origin of $\mathbf{F}_{ipe}$ will lie on this line, a distance $\frac{h}{2}$ (where h is the object height, Figure 3) from the top of the palm. The $x-axis$ of $\mathbf{F}_{ipe}$ is parallel and in the same sense as that of $\mathbf{F}_{iw}$, and the x offset is zero. If P is the palm height, and ρ is the angle between the approach axis and the $z-axis$ of $\mathbf{F}_{iw}$, then the relationship of $\mathbf{F}_{ipe}$ with respect to $\mathbf{F}_{iw}$ is:

$$TRANS(0, P + \frac{h}{2}SIN\rho, \frac{h}{2}COS\rho)ROT(\rho, 0, 0) \tag{II.5}$$

Acquisition Component

The reach termination occurs when the origins of $\mathbf{F}_{of}$ and $\mathbf{F}_{ipe}$ coincide. When the preshape terminates, we have:

$$\mathbf{F}_{ige} \triangleq \mathbf{F}_b \mathbf{T'}_{bw} \mathbf{T}_{ipe} = \mathbf{F}_{of} \tag{II.6}$$

where $\mathbf{T'}_{bw}$ is the arm DOF when the reach terminates. The grip progresses by the fingers closing around the object. In this component of the grasp, all virtual fingers have the same movement strategy. They all move in at low stiffness (so as not to disturb the object when they contact). Again, the angular separation equations are used to generate a feedforward (angular) contact target for the fingers. The goal is for all phalanges (links) on all fingers to contact the object.

For each finger, we identify the following acquisition contact strategy:

```
If link i of finger j contacts the object
then if link i-1 of finger j also has contact
    then link i stops moving;
otherwise if link i-1 has no contact
    then link i moves backwards.
```

where, by moving backwards, we mean moving away from the object, up to a maximum limit, called LIM_i (see Figure 9, frame 4). The finger is considered to have the best obtainable contact profile, if for each joint i, either it is in object contact, or it is at LIM_i.

Note that, for the Encompass grasp, any force which pushes the object back towards the palm (negative z-axis of $\mathbf{F}_{ige}$), pushes the object 'into' the hand. An alternate acquisition contact strategy which makes use of this fact is to close the fingers so that the

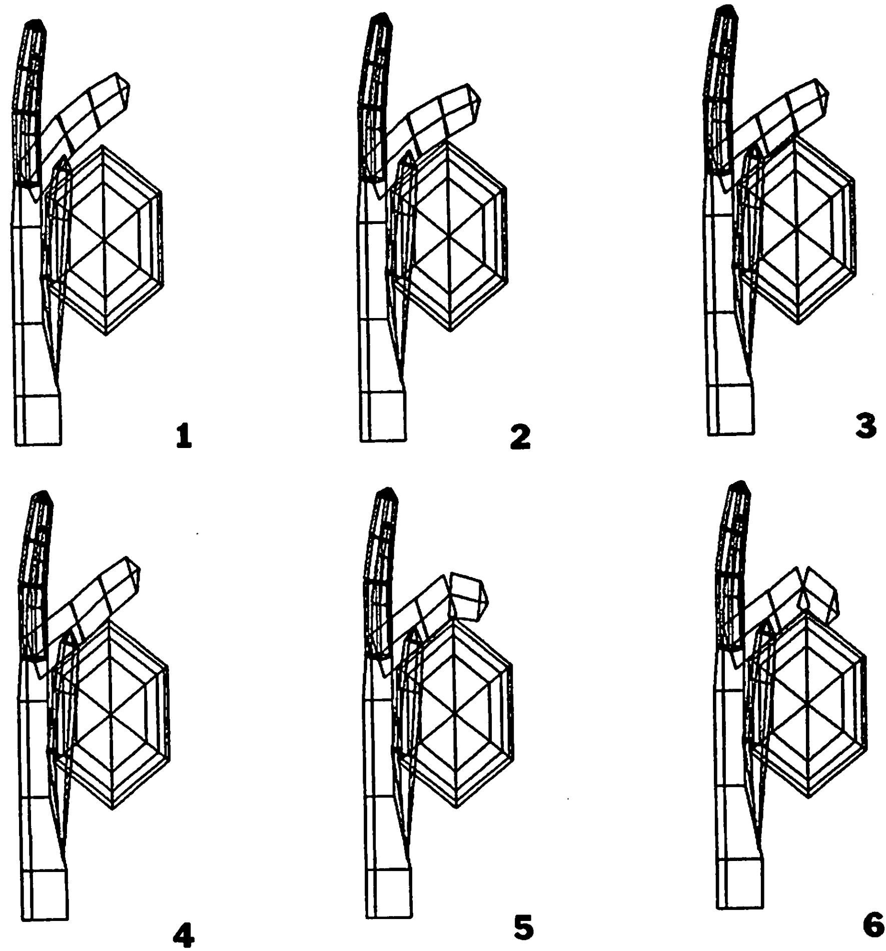


Figure 9: Encompass Acquisition Strategy.

The frames are ordered top to bottom, left to right, and illustrate the Encompass grasp acquisition strategy for one finger.

most distal (distant from the palm) links make contact first (unless the object is so large that the fingers cannot wrap around it at all).

Manipulation Component

No finger manipulation can be effected by the Encompass grasp. The manipulation component simply consists of a matrix relating finger and arm configurations to the grasped object configuration. Recall that the wrist frame is related to the world frame with the transformation $\mathbf{T}_{bw}$, as $\mathbf{F}_{iw} = \mathbf{F}_b\mathbf{T}_{bw}$. Let $\mathbf{T}_{ige}$ be a constant matrix relating $\mathbf{F}_{ige}$ to $\mathbf{F}_{iw}$, then we can derive a manipulation matrix $\mathbf{M}_{iwe}$ which relates movement in object coordinates to movements of the arm and wrist:

$$\begin{aligned} \mathbf{F}_{of} &= \mathbf{F}_b\mathbf{T}_o\mathbf{T}_f \\ &= \mathbf{F}_b\mathbf{T}_{bw}\mathbf{T}_{ige} \\ \mathbf{M}_{iwe} &= \mathbf{T^{-1}}_{ige}\mathbf{T^{-1}}_{bw} \end{aligned}$$

Every grasp will have a $\mathbf{M}_{iwj}$, $j = e, l, p$, but other grasps may have finger manipulation transforms in addition.

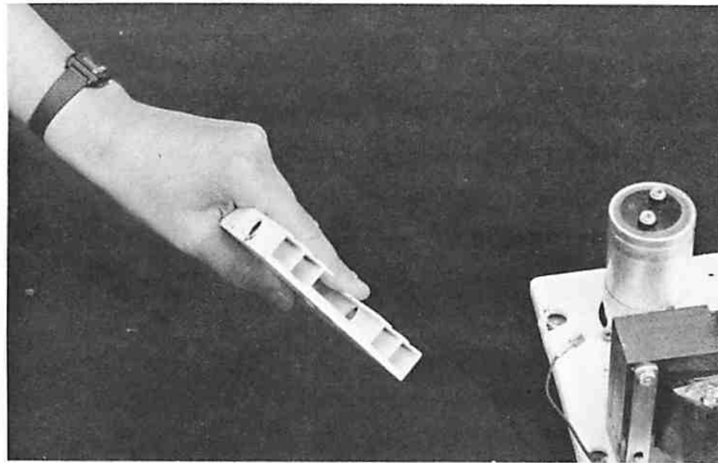
§3.4 *Lateral Grasp: (P_l, A_l, M_l)*

The Lateral grasp fashions the hand to work like a vise, or a parallel-jawed, two-fingered gripper[4]. The preshape sets up the thumb to oppose a planar array of fingers (Figure 10(a)). Again, termination of the reach will place the object COM at the origin of the $\mathbf{F}_{ipl}$ frame. The fingers close in such a way that the thumb forces act through the origin of $\mathbf{F}_{ipl}$ and balance the resultant of the finger forces; in this way, static equilibrium can be established. There is a body of literature concerning stable grasping with two-fingered grippers [15,45,88].

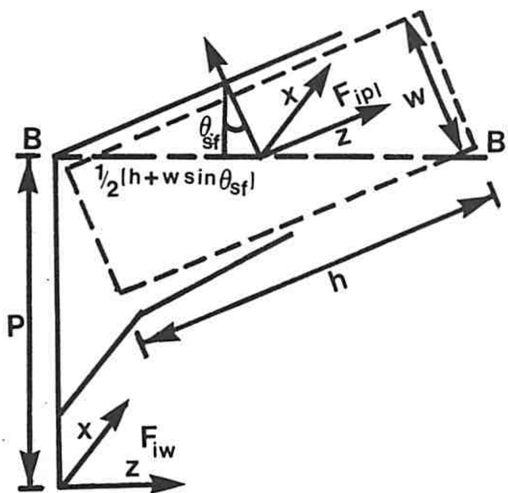
Preshape Component

The Lateral grasp preshape simply opens the thumb and fingers as the two 'jaws' of a 'virtual' two-fingered, parallel-jawed gripper. The Lateral grasp is a *two Virtual Finger* grasp. VF_1 being the thumb, and VF_2, any of the rest of the fingers. Again, the

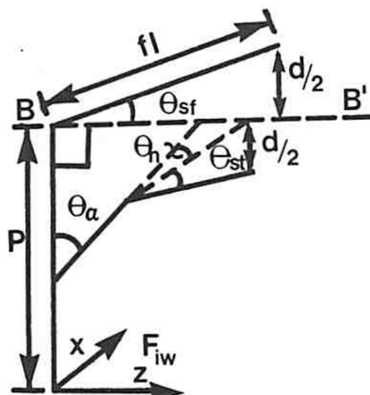
[4]Of course, the VF, and other parameterization techniques, allow the hand to perform the functions of a whole array of different-sized grippers.



(a)



(b) Placement of $\mathbf{F}_{ipl}$: The $x - axis$ is parallel and in the same sense as $\mathbf{F}_{iw}$. The origin is on the base line, a distance $\frac{1}{2}(h + w * SIN\theta_{sf})$ from the palm. The $z - axis$ is rotated by θ_{sf}.



(c) Selection of Finger-Thumb Separation: This diagram will suit either hand model. The imaginary base line BB' is constructed at 90 degrees to the palm; θ_{sf}, θ_{st} are measured against this. The total separation is $w + \varepsilon$, where w is the object width, ε is some error margin, and it is assumed that d is small. Thumb and finger separate to equal distances to achieve d.

Figure 10: The Lateral Grasp.

assignment of physical finger to virtual fingers is based on the object length, and is a function of physical hand parameters. We summarize the relationship with the function $\nu_k(l)$ for VF_k.

VF_1 and VF_2 must be separated enough to allow the object to enter the grasp. In the Lateral grasp, *only the base angles* of each finger bend to achieve the desired separation. Because of this, the finger can be treated as a single link, with a single degree of freedom θ_1 at its base. We implement this separation as follows (see Figure 10(c)): Construct an imaginary line perpendicular to the palm, and parallel to the $z-axis$ of $\mathbf{F}_{iw}$. We define a separation angle for the thumb and finger with respect to this line. The angle for the finger θ_{sf} is simply $(90-\theta_1)$. Let θ_n be the thumb angle at which the thumb end-point lies on the imaginary base-line. In all our hand models, the thumb has a fixed angular offset from the palm; call this θ_α. The separation angle θ_{st} for the thumb is $(\theta_1-\theta_n)$. If the finger length is fl and thumb length is tl, then to cause a finger-thumb separation of d, we assign a separation of $\frac{d}{2}$ to each of the finger and the thumb. In which case, the separation angles (in radians) respectively are: $\theta_{sf}=\frac{d}{2fl}$ and $\theta_{st}=\frac{d}{2tl}$. The arc approximation is valid here if we restrict the Lateral grasp to objects of small width (e.g., Figure 10(a)).

We use the same algorithm used in the Encompass grasp to spread the fingers of VF_2 to cover the object length. The $\mathbf{F}_{ipl}$ frame is placed on the imaginary base line from which the separation angles are measured as follows (and see Figure 10(b)): The $x-axis$ of $\mathbf{F}_{ipl}$ is parallel and in the same sense as that of $\mathbf{F}_{iw}$, and the x offset is zero; the z-axis of $\mathbf{F}_{ipl}$ is rotated by θ_{sf} relative to $\mathbf{F}_{iw}$, so that upon contact VF_2 is parallel to the surface of the object. The transformation of $\mathbf{F}_{ipl}$ relative to $\mathbf{F}_{iw}$ is:

$$TRANS(0,P,\frac{1}{2}(h-w*SIN\theta_{st}))ROT(\theta_{sf},0,0) \qquad \text{(II.7)}$$

Acquisition Component

Acquisition proceeds by closing the fingers in towards the object's COM. Both VFs have the same motion strategy; they move towards the object by incrementing the base angle θ_1, until object contact is made. For best grasp security, the same acquisition contact strategy as in the Encompass grasp can be used. Again, an object priority grasp approach is followed – where the fingers approach the object with low stiffness until both

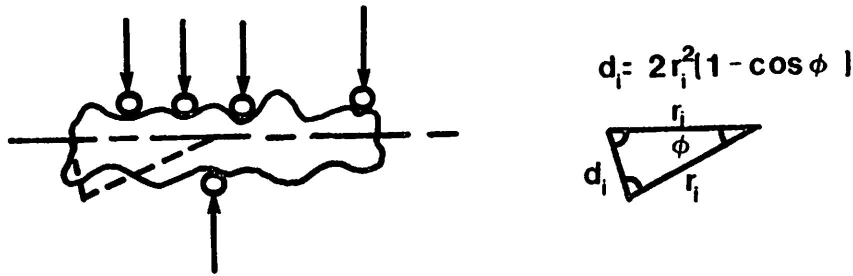


Figure 11: Compliance of Lateral Grasp to Shape (a) and Limited Manipulation of the Grasped Object (b).

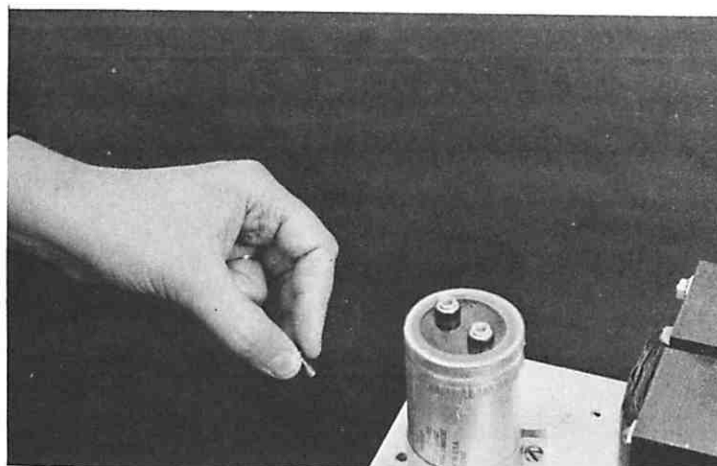
fingers are in place. The compliance of the VF to object shape ensures a more secure grasp than would otherwise be possible with a two-fingered gripper (Figure 11(a)). Once the fingers are in place, the forces necessary for static equilibrium can be deduced from finger placement. Note, however, that the ***number*** of object contact points have been much reduced between the Encompass grip and the Lateral grip; this exacerbates the problems of stable grasping. In general, to ensure a stable grasp, analysis of object boundary will be necessary.

Manipulation Component

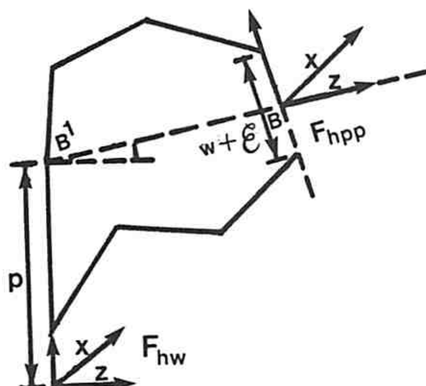
The Lateral grasp has an $\mathbf{M}_{ipl}$ transform relating object configuration to arm and wrist configuration. Yet, in addition, only a certain amount of finger manipulation is defined for rotation about the object z-axis (Figure 11(b)). Given the finger contact positions on the object, we can deduce the amount of movement necessary to rotate the object about its z axis. For manipulation we assume VF_1 and VF_2 to be single point, or collections of point contacts, so that the object can rotate around each contact. Also we associate positive and negative limits to the possible finger rotation Φ_{iml+}, Φ_{iml-}. Any call for a rotation beyond these will cause a wrist/arm movement to the correct ball park, followed by finger rotation for final precise positioning. It is important to note that ***static equilibrium*** must be maintained during the rotation.

§3.5 *Precision Grasp: (P_p, A_p, M_p)*

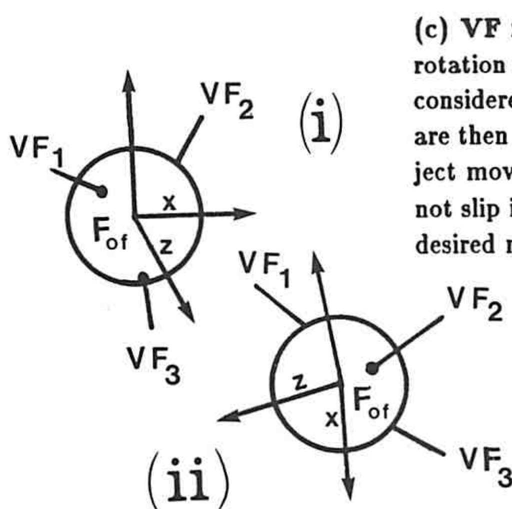
The Precision grasp is unlike the preceeding grasps, in which the inner surfaces of the fingers and thumb were used to contact the object. The Precision grasp employs the the



(a)



(b) **Placement of $\mathbf{F}_{ipl}$**: The $x-axis$ is parallel and in the same sense as that of $\mathbf{F}_{iw}$. The origin lies on the midpoint (B) of an imaginary line joining the finger and thumb-tips. The $z-axis$ is parallel to a line drawn from the top of the palm through the point B. The x offset is zero.



(c) **VF finger-tip Manipulation**: To cause the rotation of $\mathbf{F}_{of}$ shown from (i) to (ii): The VF are considered as point contacts. These contact points are then rotated by the amount of the desired object movement. Assuming that the finger-tips do not slip implies that the object also undergoes the desired motion.

Figure 12: The Precision Grasp.

finger-tips of the hand to secure the object, (Figure 12(a)). Depending on the hand model being used, this allows the fingers to be used to impart small forces and displacements to the object [76]. Because the number of hand-object contact points have been reduced dramatically, grasp stability is an important issue in this grasp.

As an extension to our basic system, the work of [10] can be used to parameterize the preshape by object shape (*round* or *flat*) to yield their *triangular* and *parallel* grips respectively. This would allow us to estimate, heuristically, a stable target grip configuration; at the same time, it would still be necessary to ensure conditions of static equilibrium while the grip is progressing.

Preshape Component

The Precision grasp provides for arbitrary object motion by finger-tip manipulation. To do this, we need to be able to specify three, non-collinear contact forces on the object [76]. The Precision grasp will, therefore, be a 3 VF grasp, with each VF responsible for exerting one of the controlling forces. As before, each VF may consist of more than one real finger, and, again, we make use of the mapping function $\nu_k(l) = (a, b)$, meaning that, $VFset_k = \{a, \ldots, b\}$, and we constrain VF_1 to be the thumb. The main components of the preshape are that the fingers be opened wide enough to permit the object to enter, and that the VF be spaced around the object. We use the angular separation equations to compute the joint angles for each finger to allow the object to enter the hand, and the interfinger separation algorithm (as specified for the Encompass grasp) to spread the fingers along the object length.

The $\mathbf{F}_{ipp}$ frame is located with respect to $\mathbf{F}_{iw}$ as follows: The $x-axis$ is parallel and in the same sense as that of $\mathbf{F}_{iw}$. The origin is on the midpoint of the line joining the finger-tips to the thumb-tip. The $z-axis$ is on the line from the origin to the top of the palm (this, remember, is the approach axis for the grasp). The x offset is zero. If $90+\rho$ is the angle that the approach axis makes with the palm, and B_z is the z ordinate of the point B, then the transformation to get $\mathbf{F}_{ipp}$ is:

$$TRANS(0, P + B_z * TAN\rho, B_z)ROT(\rho, 0, 0) \tag{II.8}$$

Acquisition Component

The reach phase terminates with the object COM between the finger-tips, which are just ready to close in. Acquisition proceeds by bringing in the finger-tips in each VF

along a trajectory through the origin of the preshape frame (ensuring therefore, that all the forces act through this). Finger contact is made at low stiffness and in such a way that the exerted forces pass through the COM of the object.

Manipulation Component

The Precision grasp also has a $\mathbf{M}_{ipp}$ transform relating object configuration to the arm/wrist configuration. As with the Lateral grasp, we take an essentially kinematic view of finger manipulation. Consider each of the VFs as a single point (see Figure 12(c)), and consider the object as a sphere centered at the origin of $\mathbf{F}_{of}$. Let us assume that a positive contact force is exerted at each contact point in such a way that the object does not slip at that contact. To achieve some motion of the object, we now simply apply that same motion to the contact points (hence, we need the inverse kinematics developed in Section 3.2 to go from the new end-points to finger angles). Note that static equilibrium must be maintained while the manipulation is in progress.

§4. Selection of Grasps

Grasp selection is the process of choosing one of the SSG grasps on the basis of the operation to be performed and the target object characteristics. To reiterate: The functional characteristics of a grasp are its ability to carry out precision movement $(P, \neg P)$ and the amount of secure affixment to the object $(F, \neg F)$. The object characteristics for grasp selection are: size $(large, small)$, shape $(round, flat)$, and length $(long, short)$.[5]

We form the mapping from functional and object characteristics to grasps on an empirical basis as follows: We discuss a number of operation sub-sequences from the simple assembly task (Figure 2) and consider what grasp, or grasp sequence, would be appropriate. These sequences are then generalized to produce grasp *index equations* linking object and task context to appropriate grasps. Finally, a more flexible method of grasp selection is considered, including a possible role for learning and adaptation in grasp selection.

Section 2.1 gives a list of the instructions to complete the example assembly task. Table 2 lists the object characteristics of each component in the assembly. First it is

[5] Of course, actual numeric values are necessary in the grasp parameterization.

Component	Size	Shape	Length
BASE-PLATE	large	flat	long
CAPACITOR	large	round	long
TRANSFORMER	large	flat	short
SIDE-PLATE	small	flat	long
TOP-PLATE	large	flat	short
BOLT	small	round	long

Table 2: Object Characteristics for the Example Assembly Task.

necessary to acquire and place the Base-Plate. From the table this is a long, flat object. The assembly instructions indicate no precision is necessary in placement – this means an Encompass (ENC) or Lateral (LAT) grasp can be chosen, avoiding the unnecessary manipulation overhead of the Precision (PRE) grasp. Of the two grasps, the LAT is suited to the acquisition of fat objects, and is chosen.

Next, the Transformer is acquired in order to place it on the Base-Plate. This operation will need strong affixment to carry the heavy Transformer over to the Base-Plate; and a precision manipulation will be necessary to align it with the four screw holes. If the Transformer was *small*, a PRE could be applied. However, since it is too large for finger-tip manipulation, a LAT or ENC will need to be applied; *neither* of these provides the necessary precision degrees of freedom to do the alignment. This is solved by using either ENC or LAT (whichever is more appropriate to the object shape), to place the component coarsely, and followed by the application of a PRE to a sub-site on the object to maneuver it precisely into position – thus dispensing with the need for the grasp to counter gravity while manipulating the object. This two-phase grasping will be part of the grasp selection rules.

The Capacitor is a round, long, large object which needs only coarse control for its insertion. Either ENC or LAT can thus be chosen; ENC best suits the object shape. Once the coarse insertion has been completed, a PRE can be applied to the top of the Capacitor to align the screw holes.

The Side-Plates are long, flat objects, and they need to be placed with some precision. The LAT grasp is chosen on this basis, since it provides for some manipulation. The Top-Plate is almost identical to the Side-Plates; a LAT is chosen, and followed by the application of a PRE to the side of the plate to maneuver it precisely.

The bolts are large round rods which need to be inserted through the Top-Plate into the Base-Plate. The insertion task, in this case, needs precision in three freedoms: ψ and θ around the x and y-axis, to control the pose of the bolt to get it into the hole, and to prevent wedging and jamming within the hole; and z, to control the force exerted to push the bolt in.[6] A PRE is chosen; since the object is *long*, all fingers may well be called into play and a *parallel* grip configuration implemented. It is much easier to hold and insert the rod this way then to grasp it perpendicular to its long axis. Note, however, that the fingers must be removed as the bolt is inserted. Once the insertion operation has been completed, it is necessary to re-grasp the head of the bolt and screw it in – a two degree of freedom task, z and ϕ around z. Again, PRE is appropriate for this task.

In summary, the grasp selection criteria are: When precision is required in dealing with an object, PRE can be chosen immediately if the object is small enough. Otherwise, ENC or LAT have to be chosen to place the object coarsely, and the followed by a PRE to some appropriate sub-site on the object, to complete the precision maneuvering. The choices of LAT or ENC would depend on the object shape; ENC deals with round objects and LAT with flat. If firm affixment is required, then PRE can be ruled out, and again, LAT or ENC can be used depending on object shape; ENC is favored over LAT, since it generates better grasp security. If both precision and firm affixment are necessary, then it is necessary to weigh the capabilities of each grasp against each other. For small objects, the PRE is appropriate. For larger objects, LAT and ENC are chosen on the basis of shape. It may be necessary to follow these up by application of PRE to an object sub-site, for the final maneuver. If neither precision nor affixment is required, the simplest grasp which suits the objects shape is chosen. For the majority of cases this is the ENC grasp.

The following logic equations formalize this process, where $small = \neg large, long = \neg short, flat = \neg round$, P and F are as usual, and $\oplus$ denotes exclusive-or:

$$\begin{aligned} PRE &= P \wedge (small \vee (\neg F \wedge long)) \\ LAT &= (large \wedge flat) \vee \neg((\neg P \oplus \neg F) \wedge long) \vee (\neg P \wedge F \wedge long \wedge flat) \\ ENC &= (large \wedge flat) \vee (\neg P \wedge (small \vee (F \wedge round))) \end{aligned} \tag{II.9}$$

The two-phase grasp can be incorporated by the equation:

$$ENC \wedge P \supset Two\ Grasps\ Necessary \tag{II.10}$$

[6]Note that if this was a restricted insertion, needing 2 rather than 3 freedoms, a LAT could be used.

§5. Summary and Discussion

In summary, in this chapter we have constructed the global framework for grasping and manipulation, lacking in the literature, but necessary if we are to use the dextrous hand as the example domain in the development of our computational model. In particular, we have developed an example tactical grasp-planner system in this framework, the SSG system. We shall use the analysis in this chapter to provide some of the characteristic computational features of the robot domain discussed in the next chapter; in addition, we shall appeal to the robotics literature for these features. In Chapter 8, as a test of the representative power of our model, the implementation of the SSG system is presented.

Although grasp selection by index equation presents a simple approach, it is also somewhat limited; many objects will not neatly fit the categories of *small, long, round*, etc. And, while *precision* and *affixment* are useful task criteria, they certainly do not exhaust all the possibilities. A more flexible and upwardly-extendible grasp selection scheme would be preferable. If three *grasp-experts* are constructed, each embodying one of these equations, some inherent grasp-selection parallelism can be exposed. Using these equations, the grasp-experts would be mutually exclusive, and would never need to 'confer' to reach a decision. If continuous quantities are substituted for the discrete function, shape, and size characteristics (for example, each defined as a suitability rating between 0 and 1), then the logic equations can be replaced by a set of probability equations to yield the appropriateness of each grasp to the operation. In this case, the grasp-experts would need to confer to resolve their decisions.

CHAPTER III

CHARACTERISTICS OF THE ROBOT DOMAIN

In this chapter, we will look at the *computational characteristics* of the robot domain. In particular, we shall be examining the computational characteristics imposed by what we have defined as *complex robot systems*: robot systems with *many degrees of freedom* and *multiple sensory capabilities* [7]. We consider the dextrous hand control framework of Chapter 2 typical of complex robot systems.

These computational characteristics will be used as fundamental specifications when we build the robot computational model. We have argued, in Chapter 1, that previous attempts to build an environment or language for robot systems have always neglected to fully complete this step and define the computational needs of the domain. To rectify this, we shall analyze what is known about the robot domain, especially in terms of the grasping and manipulation of the previous chapter, and formulate computational characteristics which are uniquely relevant to the robot domain. This chapter is divided into five sections. The first four sections present major observations on the characteristics of the robot domain. The final section summarizes these and integrates them into a tentative model specification.

§1. Observation One: Inherent Parallelism

Our first observation concerns the *parallel* nature of the robot domain. We shall argue that the control of the DOFs of a robot is a task which can be distributed among a set of concurrent processes, each of which solves a local section of the problem:

1. At the most fundamental level, robot hardware is implicitly parallel – a set of actuators, each controlling a degree of freedom (DOF) on the robot. These can be simultaneously activated at varying degrees; although this simultaneous action may

or may not produce useful behavior. This parallelism is inhibited by the *coupling* effects between links (the effect on a link of the movements of other links).

2. Still concentrating on the hardware, the robot mechanism may consist of groups of actuators with common control needs. A simple example of this grouping is the wrist versus the arm actuators on a robot in which the arm and wrist (i.e., position and orientation degrees of freedom) can be separated kinematically [33]. A more complex example occurs in the fingers of a dextrous hand, or the legs of a legged robot. This type of grouping is usually quite static, in the sense that the members of the group will not change for a given manipulator.
3. In addition to physical groupings of DOFs, we can also identify *logical groupings* of DOFs which can be controlled in parallel. The prime example of this is the *Virtual Finger* discussed in Chapter 2. In this, we identify logical groups of actuators on the hand which, for the purpose of the ongoing task, can be grouped together. Raibert uses a similar concept in his *virtual legs* [73]. In contrast with the previous grouping, this grouping will need to be dynamic: For example, reconfiguring the physical fingers which are considered to be in a virtual finger.

Focusing now on the programs which control the robot, the following inherent parallelism is evident:

1. Much of current robot programming involves (and, arguably, will always contain some element of) matrix and vector operations. The benefits of parallelism within programs containing such operations are well known in Computer Science [90].

2. The structuring of a program as a set of cooperating/competing concurrent processes has received much favorable attention in both Computer Science [32] and Artificial Intelligence [30, 6]. Many higher-level robot issues contain elements of a parallel nature. For example, it is evident from Chapter 2 that *grasping* and *reaching* can be initiated in parallel; the preshape initializing and preparing the hand, as the arm conveys and orients it to the target object. *Grasp selection*, a higher-level issue, can be viewed as three competing "grasp-experts", one for each stereotype grasp, each voting for how well its grasp suits the object and task in progress.

3. Motivated by the work of Arbib [6] and Geschke [26], among others, a major incidence of parallelism in robot programming is the parallelism between the *perceptual*

and motor aspects of a task. We shall return to this topic in Observation Three.

§2. Observation Two: Formal Verification

Our second observation concerns safety and verification issues of the robot domain. Robot control involves a computer moving a physical mechanism in real-time. It is important, therefore, to be *as sure as possible* that a control program will behave as desired. Some of the main reasons for being interested in verification of robot programs are:

1. To ensure that the robot *will* carry out its programmed task.
2. The protection of the hardware against software errors, e.g., preventing the gripper control cables from being 'over-wound' in a wrist rotation.
3. The protection of humans working with, or near, the robot in the event of a software error.

In order to allow verification of programs, the computational model needs to have a well-developed *formal syntax and semantics*. There has been little previous work in verification of robot programs. For this, most robot languages rely on their connection with general-purpose programming languages. Verification for robot programming can be divided into two areas: concerns that the robot will, in general, perform some task (liveness) or not perform certain errors (safety), and concerns that the robot will perform tasks to a specific deadline (real-time demands). The development of a formal model also includes the advantages of being able to understand and explore the power of the model itself, in terms of how easy it is for useful programs to be represented, and what limits the model puts on computation.

Specifying the behavior of complex robot systems begins to entail issues which, up until now, have been proprietary to AI, such as searching and sorting among perceptual data and reasoning about the best next action. Indeed, one of the claims we shall make about our model is that it starts to bridge the gap between AI and robotics. Our approach to these AI issues is somewhat *Piagetian* [71], in that our primary model constructs will be sensori-motor; but as we attempt to express the complexities of dextrous hand control, we shall find ourselves expressing more and more abstract concepts. A formal structure

to describe such concepts is something the AI community has sought, and it will be interesting to see how our bottom-up approach compares with more top-down models, such as Hewitt's Actors [30].

§3. Observation Three: Perception and Action

Our third observation returns to the connection between perception and action in a robot model. As robotics has progressed it has become clear that sensory input and motor output are *not* simply analogues of the peripheral read and write commands in computer programming languages. We argue here that they are the *primal* structuring concepts for describing robot behavior in computational terms. Just as Hoare revolutionized interprocess communication by stating that data input and output operations are not extensions to a programming model but *the* fundamental structures in it, we shall also argue that sensory input and motor output are *not* examples of peripheral read and write commands but the fundamental operations of the robot domain. There has been much to support this view in the literature, as we shall see.

Arbib has long analyzed the connection between sensation and action, using as a fundamental guiding structure the *action-perception* cycle of Neisser [63]. Arbib's *Schema Theory* has been applied to Neuromodeling, primarily, but also to visual analysis, natural language processing, and robotics. Sensation makes sense, he claims, only *within the context* of some action. In [6], Arbib ascribes two functions to sensation: that of triggering some action (or, more specifically, *gaining access* to potential actions suggested by perception) in context, and that of parameterizing (by an *identification algorithm*) some ongoing action in context.

Motor schemas are control systems, the 'motor programs' which describe a task. A *Perceptual schema* represents (the existence of) a particular domain of interaction in the environment. A particular world model is an *assemblage* of perceptual schemas; each suggesting motor schemas which describe appropriate actions. Arbib suggests cooperation and competition mechanisms as the way in which suggested actions are constrained to generate the correct action for this particular world model.

In his dissertation on tactile sensing, Overton [69] describes a robot language based on Arbib's schemas. In this language, a robot task is described by a number of schemas. A schema is a unit of action for the robot, an abstract data-type which becomes active

when the environment matches some expected state. It has the following components: an *activation section*, in which the sensory and goal state of the robot is analyzed to determine whether the action represented by this schema is appropriate in the current state of the world (the triggering effect of sensing as described by Arbib); an *event section,* in which the action described by this schema is programmed, including the necessary sensing to correctly parameterize this action (the parameterization effect of sensing as described by Arbib); and a *memory section,* in which prior executions of this schema can be used to modulate the next execution.

The *guarded move*, a motor command with a sensory termination, has turned out to be a powerful robot-level programming concept, and has come to be included in some form in almost all robot languages. It can be seen that this is a coarse combination of the two main uses of sensory data described by Arbib, since it can be viewed as parameterizing the motor command, or as delaying the execution of subsequent motor commands until some sensory condition is met. However, until recently, this was the only *explicit structure* relating sensing and action in robot languages.

Taylor [81] discusses an approach to robot programming with sensors based on *procedure skeletons*, which represent prototypical motion strategies. Parameterizing a skeleton produces a specific instance of that motion strategy. Recently this approach has been extended by Lozano-Perez and Brooks [51] with the notion of geometric constraint propagation, to a general system for task-level planning. Their system addresses a number of planning issues which we do not consider here. Viewed in our terms however, we see that Taylor's skeletons provide the parameterization use of sensory data described by Arbib.

Geschke's RSS [26] is a concurrent programming model developed specifically for sensory based robot control. RSS was constructed in order to allow the lowest-level servo systems to become available to the robot programmer. We consider that the structure of RSS embodies concepts which are valid all the way through the robot programming domain, and not just at the servo control level.

In RSS, in order to command the robot to do some action, the programmer initiates a computing agent, called a Servo Process. Each such servo process is a *combination of a sensory event and a motor action*. For example, 'position point R$GRIP = A", initiates a servo process which moves the robot wrist to point A and keeps it there. The servo process must be *explicitly* terminated before it will cease. Geschke's is a concurrent model rather than a distributed model, i.e., his servo-processes have no necessary temporal

constraints linking them, *but* they do not work *together* to solve problems. RSS *facilitates the integration of sensory information with motor control*, which is what we would like to do but at a more general level.

Geschke provides the evidence for the *parallelism* between sensing and action, while Arbib and Overton provide the details of the *connection* between sensing and action. This perception-action structure must be part of the fundamental structure of our model.

§4. Observation Four: Parameterizable Action Plans and Object Models

Our fourth and final observation on the computational constraints afforded by the robot domain concerns the prototypical action mentioned before in the context of Taylor's skeletons. This structure is immensely useful in specifying the parameterization of action by perception; how behavior is modified by sensory input *in the context of this action*, as Arbib has argued for some time now [5]. A similar vehicle for context has emerged in the AI field, with Shank's *scripts* [77], Minsky's *frames* [60], and the use of Arbib's *schemas* in visual analysis [86]. As we have noted, however, Arbib's schemas are as much a statement about sensory and motor interaction, as they are about the use of contextual mechanisms in reasoning.

In our analysis of the grasping and manipulation domain, the use of such prototype mechanisms is evident. We described a mechanism which, on input of the characteristics (through sensory data) of a given target object, will choose one of the three grasps as being appropriate. Additionally, given a stereotype grasp, we described it as being parameterized by the characteristics of the target object to generate a specific instance of that grasp.

Objects are well represented by a prototype or template mechanism. This is particularly appropriate in an *action-oriented* view of perception; an object can be represented as an instance of a 'template' whose parameters are simply that object-model data, or sensory input about the object, which is relevant for the task in progress.

It is interesting to note that we have constructed here part of the basic prerequisites for a task-level programming system, even though we did not start out to address these issues. As a matter of fact, we faulted other research for aiming in that direction *without* first exploring the computational structure of the robot domain. It is a nice justification of the worth of our approach that it will facilitate a task-level view of robotics.

It is common, in robot languages, to find *discrete-sample* sensory primitives; operations for polling or sampling a sensor, e.g., SRL's *input* statement. This leads to a style of programming based on the loop outline: take-a-picture: analyse-picture: move-around. This clashes strongly with the *object-oriented* (in its programming sense [36]) style of object and action representation discussed above. Discrete-sample sensory primitives promote efficiency in a uniprocessor robot controller; where CPU time must be shared between collection of sensory data and the execution of the robot actions. We argue that, despite this, a *continuous-sample* sensory primitive fits in better with the object-oriented representation style developed so far, and, hence, facilitates a task-level representation. For example, if a particular 'real-world' object is being represented by an instance of some 'template' as discussed before, then the sensory input to this template is continuous and transparent in its collection. Whenever the parameters of this template are read, they represent the current state of the object as sensed. This can also be compared to the notion of a task-based *logical sensor* [29].

§5. Summary

In summary, we have described how previous attempts at formulating a language or environment for robot programming have neglected to perform the important first step of analyzing the *computational* needs of the robot domain. Instead, they have concentrated on segmenting the physical assembly task into useful functional robot operations. We correct this omission and define four computational characteristics of the robot domain. They are: inherent parallelism at all levels; the need for behavior verification; the importance of perception and action as fundamental structuring principles; and the use of prototypical specification mechanisms. We find it gratifying that our approach allows us to facilitate task-level programming. In the next paragraph we carry out a first pass at integrating our observations into a specification for a robot computational model. This is a prelude to the informal model construction in the next chapter.

Observations One and Four strongly suggest that a model of distributed computation is appropriate for the robot domain. Distributed computation is defined to occur when a number of computing agents work together to achieve coherent behavior [55]. The more general benefits of distributed computation are well known: to increase program understandability and ease of programming; to provide the potential for *combinatorial*

implosion (i.e., the increase in efficiency which occurs when a number of problem solvers work on the same problem from different directions); to avoid the Von Neumann bottleneck (the dataflow machine is one such fruitful architecture); to increase overall program speed with parallel hardware and to provide better recovery potential in the event of hardware failure. In the field of Artificial Intelligence several distributed models have been developed. Most are *human societal* models - for example, the Actor formalism [30], the Scientific Community Metaphor [44], and the Contract Net [79]. In addition, distributed AI systems have started to appear in the planning literature [22]; a typical example of which is the work of Lesser and Corkill in our own Department [19].

CHAPTER IV

STRUCTURE OF THE COMPUTATIONAL MODEL

In this chapter and the next, we construct our model of computation for sensory-based robot control. This chapter begins the construction by presenting the model informally, relating the constructs back to the requirements of the robot domain as explored in Chapter 3, and using examples for illustration. We stress the difference between a *model of computation* and a robot programming language. It is necessary, however, to introduce a *notation* with which to describe programs in the model; but the worth of the model does not hinge on this notation, but rather on its semantics – the concepts which compose the model of computation. $\mathcal{RS}$ notation, or *syntax*, is constructed with two goals: Firstly, to clearly and completely represent the fundamental structures in the model, and, secondly, to facilitate the development of formal semantics. One of the steps which is discussed in the final chapter of this document is the construction of a *robot programming language* with $\mathcal{RS}$ as its semantics.

In order to emphasize the continuity of this work with previous work based on Arbib's schema concept, we call our robot computational model **Robot Schemas** or, for short, $\mathcal{RS}$. We shall use $\mathcal{RS}$ to denote both our model and the notation developed to describe programs in the model. The rest of this chapter is divided into two main parts: In the first part the $\mathcal{RS}$ model is outlined informally, with the emphasis on the global picture; in the second part, we construct the model in a more detailed way, as a preparation for the formal semantics of the Chapter 5.

§1. Part I: An Introduction to $\mathcal{RS}$

To permit specification of both the parallelism and the interdependence between parallel activities in the robot domain, we use the concept of *distributed computation* as our most basic structure. Computation is performed in a distributed model by the *interaction*

of a number of *concurrent* computing agents. A computing agent subsumes the concept of a process, but can also refer to hardware; it is simply a locus of computation. The concurrency could be virtual, as in a time-sharing computer system, or real, as occurs in parallel and distributed systems.

A *schema* is a generic specification of a computing agent; each schema describes a *class* of computing agents. A computing agent is created from a schema by the *instantiation operation*. We will use `teletype` font for schema names; e.g., `Pos`. We denote a computing agent which is an instantiation of a schema, by subscripting the schema name with an *instantiation number*, e.g., $\texttt{Pos}_i$. The term *schema instantiation*, or SI, and the term computing agent are synonymous in $\mathcal{RS}$.

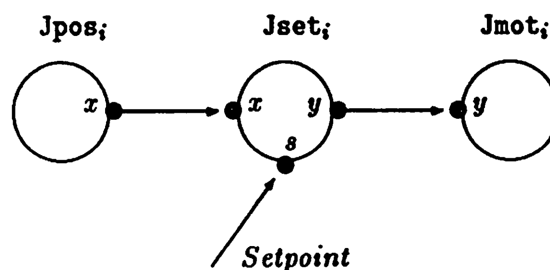
To allow communication between concurrent SIs, we define each SI to have a set of communication objects called *ports*. Each SI has a set of *input ports* and a set of *output ports*. Data is sent from an SI by writing to an output port; data is received into an SI by reading an input port. The instantiation operator takes, as parameters, the *connections* for the ports of the new SI. With each port we associate a *type*. This allows us to form rules about what may or may not be connected to a particular class of computing agent. This communication mechanism is a form of *message passing* [4].

For example, let `Jset` describe a simple position servo process; taking current position through some port x, a position setpoint through some port s, and producing appropriate motor output on some port y. The instantiation operation allows us to build arbitrary computing agents fashioned after this schema, and depending on exactly what is connected to the input and output ports, the servo process can be applied to any combination of sensors and actuators.

A key parameter in such a servo process might be some gain k. Notice that this parameter is *static* in comparison with the input and output port connections. The gain can be initialized when the SI is created, and need not be changed again. The instantiation operation can specify *initial parameter values*, such as an initial value for k in `Jset` for example, in addition to dictating what connections are applied.

We predefine a set of *primitive schemas* to interface the $\mathcal{RS}$ model with the robot and with the world. Instantiations of these primitive schemas are computing agents which represet the sensory input and motor output 'channels'. Computing agents representing sensory data are updated *dynamically*, and computing agents representing motor or actuator output control their hardware *dynamically*. For example, `Jpos` is a primitive sensory schema, which reports a joint position through its output port x. The *ith* instantiation

of **Jpos** is a computing agent which reports on the current joint position of the *ith* joint of the robot system (under some prearranged numbering of the robot mechanism). In similar fashion **Jmot** is a schema which accepts a motor input through its port *y*. The *ith* instantiation of **Jmot** is a computing agent which controls the *ith* actuator (the actuator of the *ith* joint) of the robot system. If we have a position servo process such as an instantiation of **Jset**, then we *establish the following convention*; to use the *ith* instantiation of **Jpos** and **Jmot** create the *ith* instantiation of **Jset**, such that:



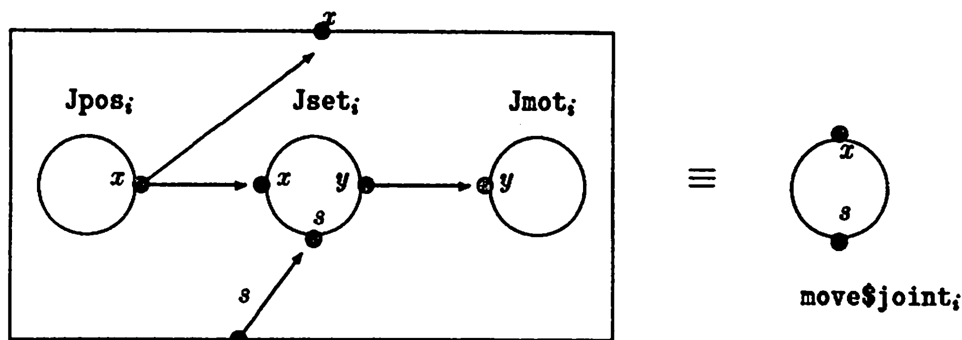
This makes specification of the instantiation numbering, for both the SI creation and for the port to port connections, more robust; as a matter of fact, given that we know that we want to create and connect the *ith* instantiation of all SIs in the network, we can dispense with mentioning the instantiation number at all! To facilitate this we will assume that an SI can reference its *own* instantiation number when necessary. There are still some questions which have to be answered — what happens if there *already exists* an *ith* instantiation of one of these schemas, and how can we set it up so that *i* just needs to be specified once? In the next paragraph we start to discuss some higher-level robot programming issues; in their solution, we shall find an answer to these questions.

Thus far, we have not done much more than describe a good computational model in which Geschke's RSS [26] or Ish-Shalom's CS [38] could be implemented and analyzed (given the formal material in the next chapter). Both of these languages were developed to aid the programming of sensor-based robots, by allowing the programmer to directly control the robot, to specify *control strategies* [58] in terms of sensory information and available actuators. Their emphasis is on the control theoretic, whereas ours is on the computational, hence the approaches are complementary. However useful an exercise it would be to have a formal computational model to complement RSS/CS, our observations in the previous chapter indicate that the structure we have introduced, so far, represents

the computational characteristics of the robot programming domain at *all levels*, and not just the control systems level addressed by RSS and CS. The fundamental construct in *RS* which deals with higher-level robot programming, even *task-level* programming, is the *Assemblage*, a mechanism to aggregate a network of SIs into a single SI.

From the programming point of view, a grouping mechanism is a strong structuring tool. Similar concepts have appeared in Cheriton's THOTH [17], Ousterhout's MEDUSA [68], Andrew's *Resources* [3], and LeBlanc's HPC [46]. An aggregation mechanism is important in approaching the *linear scaling* factor for distributed computation. The linear scaling factor dictates that as you increase the number of processors in a system, the computing power should increase in linear proportion. The problem is that as the number of processors is increased, so also are the communication problems. By modularizing computing elements into tightly-coupled local groups in communication space, the communication at any given level of abstraction is reasonably constant and manageable.

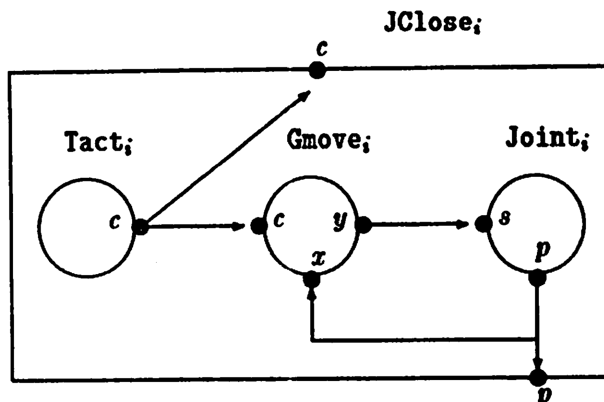
An *Assemblage SI* is a computing agent whose behavior is defined by the interaction of a network of communicating SIs. This aggregate SI can be considered the instantiation of a single *schema*, an *assemblage schema*, which must contain information on how the individual SIs are created and connected, and how the ports of the component SIs appear as the ports of the assemblage SI. An assemblage SI terminates when all its component SIs terminate. For example, the network of `Jpos`, `Jmot` and `Jset` previously discussed can be described in generic fashion using an arbitrary instantiation number i. We describe this network as an assemblage schema `move$joint`; the *ith* instantiation of which *internally* will be the network of *ith* instantiation of `Jpos`, `Jmot` and `Jset` connected as described, but *externally* will simply be a position servo process for the *ith* joint. It will have one input port s through which a desired position set point comes and one output port x through which the current position is read, i.e.:



The assemblage also has the function of *scoping* SI names; a globally unique name is (theoretically) available by considering all the assemblages within (the scope of) which a particular SI falls. However, we have no explicit need for a globally unique name and shall deal only with the local name of an SI – the schema name and instantiation number. But, chiefly, we shall try to render the instantiation number understood, as for the `move$joint` assemblage schema above.

All robot motions can be divided into two classes: *Gross* motions by the robot through free space, movement in which the robot is not in contact with any part of the environment; and *constrained* motion in which the robot moves while in contact with some part of the environment, and hence must *comply* with the environment. The interface between these two classes of motion is the classic robot-level programming construct, the guarded move [87]: A motor command with a sensory termination; for example, to approach and touch an object without producing excessive force on contact.

Let `Tact` be a primitive sensory schema for tactile sensation; the *ith* instantiation of which reports on the status of the *ith* tactile sensor. For simplicity, let us *initially* assume just *one* tactile sensor per joint (numbered in the same order as the joints), and having as output a binary signal; 0, indicating no contact, or 1, indicating contact. The task is to move some joint i until contact is obtained with tactile sensor i. Let `Jclose` be an assemblage schema describing this task as:



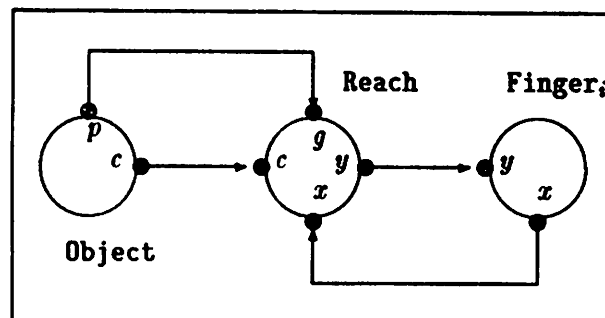
where Gmove implements the guarded move by incrementing the position of joint i as follows: Port y of Gmove is set to the contents of port x plus some increment if port c is zero, otherwise y is set to the contents of port x. Again, the instantiation number convention simplifies SI creation and connection within Jclose.

To some extent Tact$_i$ represents the *object* to be touched in this task. It is, however simple, an object model. The basis of task-level programming is that *actions* are referenced against *objects*, rather than against manipulator parts; task-level robotics is also called *model-based* robotics. In computational terms, the SI which implements the action needs to use an *object model* as reference, not a part of the robot mechanism. The assemblage construct gives us a tool to generate complex, task-specific models, and also a tool with which to structure the interaction of object models and motor actions.

Taking Jclose as a simple example: How the object model, Tact$_i$, parameterizes action is completely specified by the SI Gmove$_i$. More generally, the assemblage provides a way to isolate the exact interaction between sensation and action as in Jset or Gmove. However, for a more complex task, the object model might demand *many* sensor readings and *also* built-in object knowledge. This concept has some relation to Henderson's *logical sensors*: "A logical sensor either maps directly onto a physical sensor, or it provides a description of how one or more logical sensors are combined to produce the desired data" [29]. The problem with this is that it is oriented towards sensed data only; whereas a good object model will should consist of a mix of built-in object knowledge *plus* sensor readings. Constructing an object model using the assemblage makes it completely transparent whether the data comes from sensors, built-in knowledge, or some complex mix of the

two.

Consider the object model for the informal task of reaching out to touch a glass with some finger i of a robot system. Visual input combined with knowledge of the size and shape of the glass allow an initial estimate of the position of finger i. If the robot has proximity sensors on finger i, then as the finger approaches the glass, the proximity sensor data should take over from the visual positioning data. Finally, tactile data can be used to generate final positioning. Externally, the object model should only have two ports: The position of the desired contact point (that is, the object surface) in port p; and whether or not the finger is touching the glass through port c:



The act of moving to touch the glass is now reduced to the complexity of a guarded move, despite the complex sensory data and built-in knowledge which is being used. To summarize, this simplification arises from two concepts: the idea of task-specific object models, and the assemblage construct to build such multi-modal/built-in knowledge models. The notion of task-specific sensation is analogous to the definition of ***perception*** versus ***sensation*** in psychology, and for that reason we have previously referred to robot programming based on such object models as *Perceptual Robotics* [34].

The assemblage construct also gives us a way to describe how such task-specific object models are incorporated into actions: Assemblages are used as the basic units with which the task is structured. For this reason we shall refer to assemblages which obey the strict formating rules we devise for task structuring, as ***task-units***.

Within such a task-unit, some SIs function as object models and some as basic actions to implement the task that the task-unit represents. We call the SIs which are the object models *perceptual SIs*, after Arbib's use of the term and our definition of perceptual robotics. Those SIs which represent basic actions (from the point of view of the task being implemented) we call *motor SIs*. The connection map C in the task-unit assemblage

connects motor SIs to perceptual SIs. Task-units may be arbitrarily nested; in which case, they describe a *sensori-motor hierarchy* (similar in fashion to Albus [2] and Tsukune's SMS [85].) With each task-unit we associate a *precondition* schema. This precondition schema will instantiate the task-unit when some specified triggering conditions are met.

We introduce a *shorthand notation* for describing assemblages and task-units (which, after all, are simply assemblages singled out for special attention because they better help us structure a robot task). This shorthand will allow us to *outline* task-units and assemblages without giving too much detail. It is *not* a programming notation, and does *not* provide enough information for a compiler program to operate on. We develop a full schema specification language in the next section. In this dissertation, the shorthand will play the role of a top-down design tool, allowing the gradual development of the full schema specification.

We use the notation $\|_c(\mathtt{A}_i, \ldots)$ to indicate that the SIs $\mathtt{A}_i, \ldots$ are connected together as described by the map C. We denote an assemblage by a list of SI names between square brackets, $[\|_c(\mathtt{A}_i, \mathtt{B}_j, \ldots)]$. This foreshadows our use of square bracket to enclose the definition of a schema; externally, remember, an assemblage SI is identical to any other computing agent. In order to allow this shorthand to represent port connections (when necessary), we will optionally allow each schema to be written $\mathtt{T}_i(a_1, a_2, \ldots)(b_1, b_2, \ldots)(v_1, v_2, \ldots)$, where the a's are input port names, the b's output port names, and the v's variables. This, essentially, represents an *external* view of the schema; the view which other schemas get of it. In our shorthand we will be flexibile about how much detail we provide with the schema; the rule will be that only those ports or variables directly necessary to describe the schema's operation will be included.

A task-unit is a specially structured assemblage, which is denoted as:

$$pre : [P \stackrel{c}{\longleftrightarrow} M]$$

where pre denotes the precondition SI which will create the task-unit, P denotes the perceptual SIs, M the motor SIs and $\stackrel{c}{\longleftrightarrow}$ indicates the connection map. When it is necessary to describe the task-unit in detail, we can augment P and M to include ports, and also specify some details of the connection map C.

Consider the basic structure of the grasping problem as described in Chapter 2. From some object description or object pointer, the correct primitive visual SI representing this object (or some part of it) must be located, and the information in it must be

used to start the object model for the grasping process. We implement this ***search*** by a precondition SI, `locate$object`. The object model, the perceptual SI for the grasping task, is represented by the SI `object`; again, it is a task-specific model. The grasping process consists of determining which grasp is appropriate and instantiating that grasp. We use $\mathbf{grasp}^g$, where $g \in \{\mathcal{E}, \mathcal{L}, \mathcal{P}\}$, to denote the grasp task-unit for the ***Encompass***, ***Lateral*** and ***Precision*** grasps respectively, from Chapter 2. We represent ***grasp selection*** as the precondition SI `grasp$select` for the $\mathbf{grasp}^g$ SI:

$$locate\$object : [object \overset{c}{\longleftrightarrow} grasp\$select : grasp^g] \qquad g \in \{\mathcal{E}, \mathcal{L}, \mathcal{P}\}$$

We can expand the assemblage $\mathbf{grasp}^g$ further into its reach, preshape, acquire, and manipulate task-units. Each task-unit will have an object model culling data specific to its task. The reach object model, `RO`, need only be concerned with the ***position*** and ***orientation*** of the object as given by visual data. In similar fashion, the preshape object model, `PO`, need only be concerned with object ***size*** and ***shape*** visual information. The acquisition object model, `AO`, however, needs tactile and force information from the closing fingers, as well as some approximate size information. The manipulation object model `MO` is primarily built-in knowledge parameterized by the gripping positions of the fingers; different grasps will have different built-in manipulation structures (this was one of the main points of Chapter 2).

The acquisition task-unit should not begin until the reach and preshape phases have placed the hand near the object; this we represent with a precondition `close`. Similarly, the manipulation phase cannot begin until the object is stably acquired; this we represent by the precondition `acquired`. These preconditions provide the ***ordering*** of the components of the grasp:

$$locate\$object : [O \overset{c}{\longleftrightarrow} grasp\$select : [\begin{array}{rl} & [RO \overset{c}{\longleftrightarrow} R] \\ & [PO \overset{c}{\longleftrightarrow} P] \\ close : & [AO \overset{c}{\longleftrightarrow} A] \\ acquired : & [MO \overset{c}{\longleftrightarrow} M] \end{array}]]$$

We can expand these task-units still further. Some of these, particularly relating to the ***Virtual Finger*** concept from Chapter 2, we shall use as examples in the next section. The precondition mechanism will be seen as an essential tool in representing virtual finger in a versatile manner. Chapter 8 presents the detailed grasping programs.

To summarize, we have seen two important uses for the assemblage structure in $\mathcal{RS}$: One was the construction of task-specific object models. The other was as a structuring convention for assembling robot programs.

§2. Part II: The $\mathcal{RS}$ Model

In this section, we describe $\mathcal{RS}$ by listing and explaining the five basic properties of the model: These are the *Distributed, Instantiation, Fan, Classification* and *Aggregate* properties. We also detail a set of primitive sensory and motor schemas. Each subsection consists of zero or more repetitions of each of: the statement of a particular property or concept in $\mathcal{RS}$; followed by a syntax or notation definition, and illustrated by some examples.

§2.1 *Distributed Property*

Computation is performed in a distributed model by the *interaction* of a number of *concurrent* computing agents. Two properties of distributed computation are, therefore, *Concurrency* and *Communication*.

Concurrency: A number of computing agents, foci of computation, are concurrent if they are not *necessarily* linked by any temporal constraints on their respective computations.

Communication: To allow concurrent computing agents to communicate, we equip each computing agent with a set of communication objects called *ports*. Each computing agent has a set of *input ports* and a set of *output ports*; data is sent from a computing agent by writing to an output port, data is received into a computing agent by reading an input port. Communication *between* a set of concurrent computing agents is described *by a network map*, C, which maps output ports to input ports for the set of computing agents.

Messages are sent *synchronously* from an output port on one SI to an input port on another SI, across a connection specified in the map C [4]. Informally, whenever a computing agent reads an input port, or writes to an output port, then *no further computation* will be done by that computing agent until a write, or a read, respectively, has been carried out on the port to which this port is connected, as described by the map C.

In asynchronous communication, communicating agents do not wait for mutually coincidental reception or transmission. This is the more fundamental form, since synchronous communication can be built by asynchronous primitives. However, we shall demonstrate that the addition of the instantiation operation and fan-in and fan-out [1] in C extends synchronous communication such that asynchronous primitives can be approximated. This is a fact we shall prove formally in the next chapter. Essentially, it means we get the best of both worlds; communication can be modeled formally, and the power of the asynchronous primitives can still be used.

§2.2 *Instantiation Property*

Given the observations on the usefulness of the *prototype* concept at all levels of the robot programming domain, we introduce the *instantiation* operation as the mechanism whereby computing agents are created. A *schema* is a generic specification of a computing agent; it describes a *class* of computing agents. A computing agent is created from a schema by the *instantiation operation*.

We indicate the instantiation of a schema to create a computing agent as $\mathbf{S}^j \rightsquigarrow \mathbf{S}^j_i$; i is referred to as the *instantiation number* of the computing agent. The instantiation operation takes as parameters the schema name, initial values for internal variables of the schema v, and a port connection map c for ports on the schema. The term computing agent and the term *schema instantiation*, or SI for short, are synonymous.

Example 1

Consider implementing the recursive factorial function in $\mathcal{RS}$:

$$F(n) = \begin{cases} 1 & if\ n \geq 1 \\ n * F(n-1) & else \end{cases}$$

Each recursive step in the computation of $F(n)$ for some n, can be represented by a single computing agent. The behavior of each computing agent would be, simply, to determine if it was computing the base case ($n = 1$), and if not, to *create* a new computing agent to handle the next recursive step. Communication between computing agents would consist of the transfer of a continued factorial request to new computing agents, and the passing back of partial results from each recursive step. This network can be described by a single schema, `Factorial`, which recursively creates instantiations of itself until the base case is reached. ■

[1] Either is sufficient on its own.

Example 2

Consider the following simple sensory-based robot task. Assume we are given a two-fingered, parallel-jawed gripper, mounted on a Cartesian arm, and equipped with tactile sensors on the inside of each jaw of the gripper. The gripper is placed so that a target object is somewhere between the gripper jaws. Implement the *CENTER* command [62]: The jaws of the gripper close concurrently on the object until one of the tactile sensors is activated, at which point the robot wrist is moved to center the gripper above the object. This process is repeated until both jaws of the gripper touch the object simultaneously.

This problem can be represented by two schemas, `Wrist` and `Finger`; three computing agents, one instantiation of `Wrist` and two of `Finger`. Concentrating on what must be passed between SIs, it is clear that to move the wrist correctly, the wrist SI must know *the position* of each finger and the *status* of its touch sensor. Assume this information is available to the `Wrist`. Let us further simplify by assuming the gripper is oriented with its jaws along the $x-axis$; let the object width be $2d$, and the gripper span be $2G$. If `Wrist` receives a position g on *contact* from one of the `Finger` SI, then the wrist must move $(G - g - d)$ in the direction of the touching finger to recenter the gripper. ■

Our syntactic unit of distributed computation is the *schema*. A *schema* description consists of the following: a list of input and output ports, an internal local variable list and a behavior section. The behavior section is a program which loops continuously, once the schema has been instantiated as an SI, until that SI *deinstantiates*. The program instructions can synchronously read from or write to the ports, access internal variables, instantiate other *schemas*, or deinstantiate.

Definition 3

We use the following syntax to define a schema:

```
[  Schema-Name:        N
   Input-Port-List:    (ip)
   Output-Port-List:   (op)
   Variable-List:      (v)
   Behavior:           (b)    ]
```

- N is an identifying name for the schema.
- ip, op are lists of input and output port names respectively, and the port data-types. Each element in the list must be in the form:

$$\langle Portname\rangle:\langle Porttype\rangle$$

where the allowable data-types are *Integer, Real, Vector* and *Matrix*.[2] For our purposes we restrict the latter two to represent 4 element vectors and 4 by 4 element matrices.

- v is a list of internal variable names and their data-types, in the same syntax as above.
- b is a specification of behavior.

All five components of the schema description must be present; however, components other than the name may be empty, which we indicate by empty parentheses ( ). ■

Definition 4

We define the behavior section by the following syntax:

$$
\begin{array}{lcl}
Behavior & ::= & \wedge \mid \langle Stat \rangle ; \langle Behavior \rangle \\
Stat & ::= & \langle Assign \rangle \mid \langle If \rangle \mid \langle Instn \rangle \mid \langle Dinstn \rangle \\
 & & \mid \langle For \rangle \mid \langle Forall \rangle \\
Assign & ::= & \langle Var \rangle := \langle Expression \rangle \mid \\
 & & \langle OutputPort \rangle := \langle Expression \rangle
\end{array}
$$

■

We embed reading and writing into the syntax of *Assign*; an input portname occurring in *Expression* is a read to that port, and an output portname on the left hand side of an *Assign* is a write to that port. Apart from this, we assume standard syntax and semantics for an *Expression*, with the addition of certain vector and matrix operations.

Definition 5

We introduce operations to indicate vector and matrix component-wise addition and subtraction, matrix multiplication and vector-matrix multiplication. We also introduce notation for writing vector/matrix constants, and for element retrieval for a vector or matrix.

We write a vector or matrix constant by listing the elements between parentheses, e.g., $(0, 0, \delta, 1)$ is a vector constant. These can be combined with vector/matrix values from vector/matrix ports and variables in a standard way. For example, if p and po are vector ports, then the result of the assignment $po := p - (0, 0, \delta, 1)$ is to write the vector which has been input on the p port to the po port having subtracted component-wise the constant $(0, 0, \delta, 1)$ from it.

If A and B are two vector variables, then the expression $A.B$ is the matrix multiplication of A and B. If v is a vector variable, then the expression $A : v$ is the vector obtained by multiplying

[2]These types were adequate to represent the examples in this dissertation. We are not making the statement that these are the only data types relevant to robotics. The formal semantics of Chapter 5 can be used with any set of port types.

the vector v and the matrix A. The expression $A(1,1)$ is the element in the first row of the first column of the matrix A, and, similarly, $v(2)$ is the second element of the vector v. ■

Definition 6

```
If   ::=  If ⟨Condition⟩ Then ⟨Behavior⟩
            Else ⟨Behavior⟩ Endif
For  ::=  For ⟨Index⟩ = lb...up
            ⟨Behavior⟩ Endfor
```

■

We assume standard syntax for *Condition*, and constrain *Index* to be an internal integer variable for simplicity. Apart from this, the semantics of **If** and **For** are as one would expect.

Example 7

The schema **Summate** describes a generic computing agent which accepts a single input, adds it to some internal variable, and generates the result on its output port. This is similar to the behavior of a synchronous register; in commercial terms it might describe the behavior of a simple *bank-balance* computing agent.

```
[  SUMMATE
   Input-Port-List:     ( n:Real )
   Output-Port-List:    ( sn:Real )
   Variable-List:       ( sum:Real )
   Behavior:            ( sum := sum + n;
                            sn := sum;      )  ]
```

Example 8

The **Gmove** schema describes a computing agent which receives a joint position and a tactile sensor status through its input ports, and produces, through its output port, the next desired position of the joint, subject to the constraint that the joint stop moving *when the tactile sensor indicates contact*. It can be seen that this is an example of the GUARDED MOVE command common in robot programming languages.

```
[  GMOVE
   Input-Port-List:     ( tstatus:Integer pos:Integer )
   Output-Port-List:    ( newpos:Integer )
   Variable-List:       ( incre:Integer )
   Behavior:            ( If (tstatus = 0) Then
                             newpos := pos + incre;
                             Else newpos := pos;
                           Endif     )                  ]
```

The *instantiation operation* takes as input a *schema* and some *instantiation parameters*, and produces a *schema instantiation* or SI. The *instantiation parameters* consist of initial values for the internal variables, and a *connection map* specifying connections of the ports on this *schema* to ports on other SIs. We shall extend this definition presently; the shorter form of it suffices for now.

Definition 9

$$\begin{aligned} Instn &::= \langle Schemaname \rangle_{(Inst)} (\langle CI \rangle)(\langle CO \rangle)(\langle V \rangle) \\ Dinstn &::= \textbf{Stop} \end{aligned}$$

CI is a list of input port names, and the list denotes port connections *between the named ports and the corresponding (by position) port names of the created SI.* In similar fashion, CO is a list of output port names and denotes connections between the named ports, and the corresponding (by position) ports of the created SI. In both of these lists, the *types* of connected ports must match. The V parameter for the instantiation operation is a list of initial variable values of *Schemaname*; these are assigned *by position* to the internal variables of *Schemaname* as they are specified in its V list (the same syntax as is usual for procedure/function parameters).

Inst is an optional integer parameter which specifies which local[3] instantiation is to be created. If *Inst* is *not* specified, then a default number is picked according to the simple algorithm: The next instantiation number for schema *schemaname* is the highest current instantiation number of *schemaname* plus 1. An SI can internally reference its *own* instantiation number by calling the special function $\#I$. ■

Note the similarity between the schema definition format, and the format of the instantiation command. To illustrate schema syntax we present the Factorial example discussed earlier.

Example 10

This example gives the details of the `Factorial` schema of Example 1. When instantiated and passed a value on x, an SI of `Factorial` creates a network of processes recursively (see Figure 13).

---

[3]Ignore this word for now.

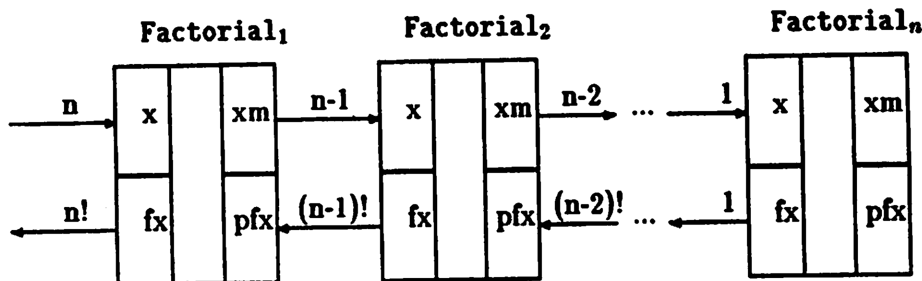


Figure 13: Recursive Network of Factorial.

```
[  FACTORIAL
   Input-Port-List:     ( x:Integer pfx:Integer )
   Output-Port-List:    ( xm:Integer fx:Integer )
   Variable-List:       ( temp:Integer )
   Behavior:            ( temp := x
                          If ( temp ≤ 1 ) Then fx := 1;
                            Else FACTORIAL(xm)(pfx)();
                                 xm := temp - 1;
                                 fx := pfx * temp;
                                 Stop;
                          Endif; )                      ]
```

The behavior of the schema is as follows: A value is read from the port x and stored in the variable *temp*; this is value whose factorial will be evaluated. If the value returned was less than or equal to one, then (by definition) one is returned as its factorial. The ports x and fx function as the main input and output routes respectively.

Otherwise, the instantiation instruction is used to create another instantiation of this schema connected as follows (and see Figure 13): The xm port on the executing (creator) SI is connected to the x port of the new (created) SI (remember, connection is made by positional correlation in the port list). Similarly, the fx port of the new SI is connected to the pfx port of the creator SI. The ports pfx and xm function as the continuation path for the evaluation of the factorial.

The factorial of *temp* − 1 is passed back to the creator SI through the pfx port, and this (by definition) multiplied by *temp* is the value of *temp*! Note the use of the temporary variable *temp* to read initial input from x. This is necessary since communication is synchronous: If x were substituted for *temp* throughout, then each read of x would demand a new value in x; *temp* acts as a buffer to ensure that the whole factorial algorithm is carried out for *each* value written to x.

■

§2.3 Primitive Schemas

In order to express robot control programs in $\mathcal{RS}$ we need some way to represent basic sensory and motor actions. We predefine a set of primitive sensory and motor schemas which embody the *interface* between $\mathcal{RS}$ and the world/robot. The number of these predefined schemas and their internal structure are implementation considerations. In this section we present a sufficient set of primitive schemas to address our examples. In the chapter on verification, we shall associate assertions with each primitive schema defined, which will relate events in the the world to computations with $\mathcal{RS}$. Primitive schemas are considered *local to all assemblages*, they can be referenced and instantiated in any assemblage.

The two primitive schemas Jpos and Jmot were introduced briefly before, we now specify them in more detail. Jpos is a primitive sensory schema. It describes the current position of a degree of freedom on the robot mechanism according to the numbering convention motivated and described before. That is, the *ith* instantiation of Jpos reports, through its output port x, the position of the *ith* degree of freedom on the robot mechanism. The specification of the behavior of Jpos is implementation dependent, and we shall present only the port definitions here:

Definition 11

```
[  JPOS
   Input-Port-List:    ( )
   Output-Port-List:   ( x:Real )
   ...                              ]
```

Since instantiation numbers are local to assemblages, it is possible that there could be many Jpos_i for some i (only one per assemblage though!). They can each function without interfering with each other, since they only inspect the joint position.

The ith instantiation of the Jmot schema accepts input through its input port y and has an implementation-defined effect on the *ith* joint of the robot mechanism for some defined, physical numbering system. For a stepper motor, Jmot might control the number of steps to take, for a DC-servo motor it might control torque. An important point to grasp is that to control the ith joint, one must make a local ith instantiation of the schema Jmot.

From our earlier discussion of assemblage we note that instantiation numbering is local to an assemblage. The reason for this was to facilitate our instantiation numbering

convention for hooking up SIs to appropriate primitive sensory and motor SIs. If more than one $\texttt{Jmot}_i$ exists across all assemblages, there is an ambiguity problem – which SI actually controls the *ith* joint? It is feasible to use an *activation level* approach, and say that the SI with the highest 'activation level' (in some defined sense) gets to control the joint. This is in the spirit of [6]; we will not follow this up here, but, instead, limit the number of $\texttt{Jmot}_i$ to *exactly one across all assemblages.*

Definition 12

```
[  JMOT
   Input-Port-List:    ( y:Real )
   Output-Port-List:   ( )
   ...                              ]
```

Note that we can define primitive sensory and motor schemas for almost any sensor or actuator in this fashion. We consider two other robot senses, a tactile sense and a visual sense. An SI $\texttt{Tact}_i$ reports on the status of the *ith* tactile sensor, for some numbering of the tactile sensors. In our first example using Tact, we let this numbering system be the same as that of the robot limbs – it does not have to be. The instantiation numbering convention allows us to index tactile sensors or sensor arrays in any way we choose. We shall assume the output of a $\texttt{Tact}_i$ SI through its port *c* is a measure of the normal force being exerted on the sensor – a realistic assumption [69].

Definition 13

```
[  TACT
   Input-Port-List:    ( )
   Output-Port-List:   ( c:Real )
   ...                              ]
```

Vision is a more difficult sense to deal with in the fashion in which we have developed position and touch. The fundamental units produced by a visual sensor are pixels; the desired logical visual units are lists of object characteristics. There is no easy mapping between pixels and objects. Part of this mapping consists of the bottom-up (from pixels) segmentation of the image into regions with similar feature values; part consists of the use of knowledge about specific object characteristics (e.g., shape, size) to determine if these objects are represented in the image; and part consists of the use of knowledge about what task is to be performed to determine what to consider as an object (for *this* task). To render the problem more manageable we shall not consider the first two parts above. The third part we shall consider, and use it as a basic structuring concept (task-unit, Definition 23).

We shall rely on a visual preprocessing system to go from the raw visual image to a list of candidate objects, and our primitive sensory schema for vision will be an SI representing an object in the visual field. The preprocessor could be a simple industrial vision system, or a complex knowledge-based system such as Weymouth's [86]. The difference between these two extremes, apart from speed, will be the faithfulness with which the transition from pixel values to object characteristics is made. We assume that the visual preprocessor we have mentioned acts to ***instantiate*** and ***deinstantiate*** dynamically some number of instantiations of the EOB schema (Environmental OBject schema). Each EOB SI represents the result of processing the visual scene to extract a list of objects, and, as we have said, we do not currently consider the details of this highly complex task. We demand that object continuity is maintained for each EOB SI; that is, that once EOB_i, say, has been instantiated, it always corresponds to the same external object. These EOB SIs are the visual system's 'best guess' at what corresponds to objects in the outside world. We shall consider ways, later on, in which these unclassified objects can be classified and/or amalgamated in the presence of task information.

The EOB schema is defined as having no input ports, and one output port per feature measured (it could then be described as an assemblage of **Feature** SIs, each of which measures one feature, and provides one port on the assemblage). The EOB schema has ports which can most extensively be described as a *port array*; we denote this by subscripting the ports (Definition 18) in the assemblage definition.

Definition 14

```
[  EOB
   Input-Port-List:     ( )
   Output-Port-List:    ( f1...f3:Vector )
   ...                                   ]
```

In order to use this in conjunction with the material in Chapter 2, we construct the following description of the feature ports. It is by no means the only possible definition for an EOB. Each of the three ports are of type vector, i.e., a four-element, 1-D array. They represent object characteristics as follows: f_1 is the visual estimate of the object's position $(x, y, z, 1)$; f_2, the visual estimate of orientation $(rx, ry, rz, 1)$; and f_3 is a feature vector giving numerical data *(shape,length,width,height)*. Shape is 1 for round, 0 for flat. Length, width and height are the length, width and height of the object, from which the size and length (binary) classifications of Chapter 2 can be deduced.

§2.4 Aggregation Property:

An *Assemblage SI* is a computing agent whose behavior is defined as the interaction of a number of communicating SIs. This aggregate SI can be considered the instantiation of a single *schema*, an *assemblage schema*, which must contain information on how the individual SIs are created and connected, and how the ports of the component SIs appear as the ports of the assemblage SI.

The necessity of this form of grouping mechanism is evident from Observations One, Three and Four of Chapter 3. Observation One details the necessity to group the degrees of freedom into groups based on physical and logical criteria. Observation Three pointed out the fact that a robot task is a complex aggregate of sensory and motor actions. In Observation Four the construction of task-appropriate object models is described. Obviously, a powerful way to build such models is by aggregating SIs representing particular sensory input and those representing task-dependent object knowledge into a single composite object-model SI.

An *assemblage schema* description consists of a schema name and input and output port lists, the same as a schema. These describe how the assemblage SI appears to other SIs. Additionally, an assemblage schema description contains a list of component schemas; these will form the network of SIs which comprises the assemblage. We will refer to these schemas as *local* schemas. The network is initialized and maintained by the initialization behavior section. The behavior section uses the instantiation operation, as described, to create and connect local SIs together to form the assemblage. In addition to the *network map* detailing the connections of SIs *within* the assemblage, an assemblage schema also has to specify an *equivalence* list of assemblage port names with component SI port names.

We further strengthen modularity by defining instantiation numbering to be *local* to the assemblage in which the SI is a component. Only local schema names can be referenced in the assemblage, and only in their local instantiations.

We define the instantiation number to be an optional parameter to the instantiation operation. This is the mechanism by which we can produce the appropriate instantiations of primitive sensory and motor schemas to connect up. A basic rule which we will attempt never to violate, is to specify *instantiation numbers with variables* in the instantiation operation.

An assemblage SI terminates when all its component SIs terminate, or if it is explictly

deinstantiated by some other SI. In the case of a task-unit assemblage we shall modify this definition slightly for convenience.

Definition 15

We use the following syntax for assemblage definition:

```
[  N ASSEMBLAGE
   Input-Port-List:       (ip)
   Output-Port-List:      (op)
   Variable-List:         (v)
    Component-Schemas:     (s)
   Initialization:        (ib)   ]
```

- N, ip, and op are the assemblage name and the lists of its input and output ports respectively. These determine how the assemblage appears to other SIs. v is an internal variable list.
- s is a list of schemas which are local to this assemblage, i.e., those schemas which will be instantiated to form the SI network.
- ib initializes the network of SI; it creates and connects the SIs which form this assemblage. We do not constrain it to terminate when the network has been initialized, and it can be used to maintain the assemblage network dynamically.

■

In order to allow the ib component to initialize the assemblage accordingly, we extend the definition of the instantiation command in the following manner: We constrain the schema name which occurs in the instantiation command to be a local schema name in the assemblage. In addition to allowing the CI and CO lists to contain local port names (that is, port names of the SI in which the instruction occurs), we extend it to allow local SI names (that is, names of SIs which occur in the assemblage).

The two connection lists CI and CO are both positional lists; a name in some ith position in the list indicates a connection between this name and the ith port of the schema being created. This has some advantages over a more explicit notation; for example, listing origin and image port and SI names with an arrow between them to indicate the direction of the connection. Our method unambiguously states the direction of the connection by having two lists, and allows unambiguous port and SI naming, by eliminating the the need to specify the ports of the newly created SI.

It does have some disadvantages which we need special notation to deal with. Empty positions in the list is the most obvious; and our solution to this is standard. We shall

place commas between items in the two lists, a missing item is denoted by two adjacent commas. Fan-in and fan-out of connections is a topic we shall deal with in the next section, however, we anticipate this by mentioning that we shall need to introduce special notation to indicate two or more connections to the same port in the CI and CO lists. We use $\|^+$, and $\|$, to indicate multiple connections to a port; we shall explain this more thoroughly in the next section.

Finally, we need special notation to represent the assemblage port equivalence list. Given the network of SIs which comprise the assemblage, we need some way to indicate which ports on which SIs in this network will be taken as the external ports on the assemblage, which is, after all, the network viewed as a single SI. We denote an equivalence between an assemblage port and a port on a created SI by placing the assemblage port name in the port connection lists at the appropriate place and preceded by the $\equiv$ symbol.

It is important to point out that an equivalence between an assemblage port and some component SI port is *not* a connection: If any SI connects to this assemblage port, *then* it is equivalent to connecting to the port on the component SI. Equivalences go from input port to input port, or output port to output port.

Definition 16

The full instantiation definition is:

$$
\begin{array}{lcl}
\langle Inst\rangle & ::= & \langle local-sname\rangle_{\langle inst\rangle}(\langle CI\rangle)(\langle CO\rangle)(\langle V\rangle) \\
\langle local-sname\rangle & ::= & name\ of\ schema\ local\ to\ assemblage \\
\langle CI\rangle, \langle CO\rangle & ::= & \langle connection-list\rangle \\
\langle connection-list\rangle & ::= & \Lambda \mid \langle item\rangle\langle connection-list\rangle \\
\langle item\rangle & ::= & \langle connection\rangle \mid \langle connection\rangle \parallel \langle item\rangle \\
 & & \mid \langle connection\rangle \parallel^+ \langle item\rangle \\
\langle connection\rangle & ::= & \langle local-port-name\rangle \mid \langle local-SIPname\rangle \mid \langle equiv-port\rangle \\
\langle local-port-name\rangle & ::= & a\ port\ of\ the\ creating\ SI \\
\langle local-SIPname\rangle & ::= & \langle local-sname\rangle_{\langle inst\rangle}\langle port\rangle \\
\langle equiv-port\rangle & ::= & \equiv \langle local-port-name\rangle
\end{array}
$$

■

For now, we give trivial example to illustrate assemblage syntax: We shall assume *dangling connections*, connections to non-existent SIs are filtered and ignored (we will dwell on this later). This filtering occurs at the network-map level rather than the schema level; an SI does not know to what its ports are connected. Writes and reads to ports

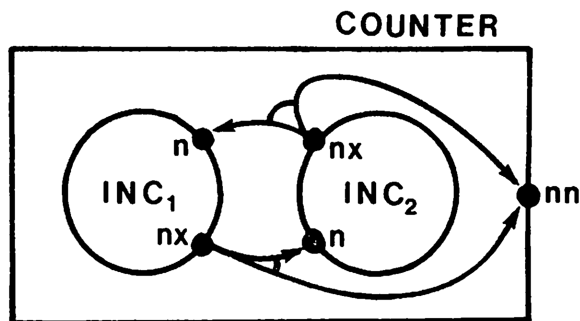


Figure 14: The Counter Assemblage.

will go ahead regardless of whether the ports have been connected or not. This helps tremendously with generic specification, and we will see this in the next example. We restate the fact that an SI can reference its *own* instantiation number by calling the special function $\#I$ (see page 67). In this way the *ith* instantiation of some assemblage SI such as `move$joint` (from our earlier examples) can internally create appropriately named local networks.

Example 17

The assemblage **Counter** produces all the whole numbers, successively, on its output port *n*. Internally, it consists of two communicating SIs. First we define the schema Inc:

```
[  INC
   Input-Port-List:      ( n:Integer )
   Output-Port-List:     ( nx:Integer )
   Variables:            ( buffer:Integer )
   Behavior:             (      nx := buffer + 1;
                           buffer := n;            )  ]
```

We now construct the assemblage **Counter** based on this schema (and see Figure 14):

```
[  COUNTER Assemblage
   Input-Port-List:      ( )
   Output-Port-List:     ( nn:Integer )
   Variables:            ( i:Integer )
   Component-Schemas:    ( Inc )
   Initialization:       (   For i := 1 ... 2
                                Inc_i (Inc_i+1 (nx))(Inc_i+1 (n)  ||+≡nn)(i-1);
                          Endfor  )                                          ]
```

The new feature in this schema is the use of $||^{+}$ to indicate fan-out in the connections of *Inc* (indicated in the diagram by branching connections at each *nx* port). There is also fan-in to the port *nn*; each output port is defined equivalent to the assemblage output port *nn*. We have not

detailed the semantics of fan-in yet, but if a connection was made to port *nn*, whenever *either* Inc SI transmits a value through its output port, then such a value would be received.

Note also that because we ignore 'dangling' connections we can use the For loop to describe the connections in a generic fashion (based on the loop index). It can be seen that Inc_1 will produce all the *odd* numbers, and Inc_2 will produce all the even numbers. Note that we can give an explicit network connection map and equivalence list for this assemblage (as we will be able to do for any assemblage) while this will be useful in verification (Chapter 6), it can, in general, be lengthy.

■

In an *implementation* of the assemblage construct we would expect that all the component SIs are close together in communication space; this is a process distribution criterion for a distributed computer system. We would also expect that an assemblage can be started and stopped as if it were a single SI and that all the component SIs of the assemblage are treated equally with respect to resource access.

§2.5 *Fan property:*

The fan properties describe how our model deals with fan-in and fan-out of network connections. A good deal of power will result from the $\mathcal{RS}$ fan-in and fan-out definitions.

Fan-Out: Fan-out occurs when the output port of some SI is connected to more than one input port on other SIs; if there are *n* such connections, the fan-out is said to be of degree *n*. For example, let port *a* on A be connected to both *b* on B and *c* on C, i.e., in explicit form (and see Figure 15(b)):

$$\begin{array}{lcl} C(c) & \leftarrow & A(a) \\ B(b) & \leftarrow & A(a) \end{array}$$

where A,B and C are schemas local to some assemblage, then the instantiation command to create the fan-out connection in a network such as the above might be:

$$A()(C(c) \parallel B(b))()$$

Any value written to *a* by A will be passed *both* to *b* and *c*. However, it is necessary to consider whether the write at *a* terminates when *one* of *b* or *c* is read, or whether both must be read before the write will terminate. The former is called OR-semantics (either read), and the latter, AND-semantics (both read). In general, there are uses for both kinds. We shall explore this issue in some depth and more formally in the next

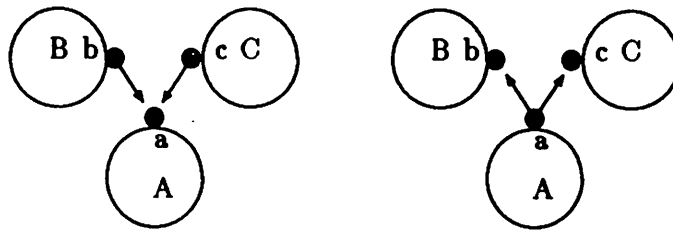


Figure 15: Fan-in (a) and Fan-out (b).

chapter. Suffice it to say now, that we shall choose OR-semantics as the basic semantics of both fan-in and fan-out; that is, if the fan-in or fan-out occurs implicitly (not explicitly specified, as at *nn* in the previous example), or if it is specified in the port connection list using ‖. We will use $\|^+$ to denote AND-semantics; this must be specified explicitly (as at the port *nx* in the previous example). Our previous example would not have worked very well if we had used OR-semantics (‖) instead of AND-semantics ($\|^+$) at the port *n*. In that case an SI connected to the **Counter** assemblage could only read values at *nn* by 'stealing' them before one of the **Inc** SI got them. As soon as this happened the whole system would go into deadlock.

Fan-In: Fan-In occurs when an input port on some SI has more than one output ports from other SIs connected to it; again, if there are *n* such connections, we say the fan-in has degree *n*. For example, let port *b* of **B** and port *c* of **C** both be connected to input port *a* of **A** (and see Figure 15(a)):

$$\begin{array}{lcl} A(a) & \leftarrow & C(c) \\ A(a) & \leftarrow & B(b) \end{array}$$

An instantiation command to create the fan-in connections in such a network might be:

$$A(B(b) \,\|\, C(c))()()$$

Any value written by *either* **C** or **B** will be received at *a*. The basic semantics of fan-in are OR-semantics, i.e., a read to *a* by **A** will complete if either **C** reads *c*, or **B** reads *b*. In Chapter 5, we will discuss AND-semantics for fan-in; however, they do not appear as useful as AND-semantics in fan-out.

Port Arrays: As a further aid in expressing generic network specifications, we introduce the port array concept. With our current notation, we can only denote single ports in a port list. A port array is an array of ports; a particular port is specified not by

a name, but by the array name plus an index. In Chapter 5, we shall show how this concept emerges in a uniform fashion from the assemblage construct, and we will detail its formal semantics.

Definition 18

We declare a port array in the input or output port list by writing:

$$P_{lb} \ldots P_{ub} : \langle Data - Type \rangle$$

where *ub* and *lb* are the upper and lower bounds of the port array. Within a schema, we refer to an element of a port array, which is a port, by using the port array name with a subscript between *ub* and *lb*. ■

Example 19

We now specify the assemblage to carry out the CENTER task. We shall assume that the inputs to the assemblage are each finger position and the wrist position, and the output from the assemblage are new finger positions and a new wrist position. The `Wrist` schema accepts a current wrist input position and returns a new desired wrist position. It also accepts finger positions and tactile sensor status from the SIs controlling the gripper fingers. The `Finger` schema takes as input a current finger position and sensor status, and produces as output the new desired finger position. In a previous example (Example 2) we have outlined the details of the `Finger` schema and `Wrist` schema, so here we shall only present their port lists, not their internal behavior.

```
[ FINGER
  Input-Port-List:      ( ts:Integer fp:Integer )
  Output-Port-List:     ( nfp:Integer )
  ...                                            ]

[ WRIST
  Input-Port-List:      ( wp:Integer ts1...ts2:Integer )
                          fp1...fp2:Integer )
  Output-Port-List:     ( nwp:Integer )
  ...                                                    ]

[ CENTER Assemblage
  Input-Port-List:      ( wp:Integer fp1...fp2:Integer
  Output-Port-List:     ( nwp:Integer nfp1...nfp2:Integer )
  Variable-List:        ( i:Integer )
  Component-Schemas:    ( Finger Wrist Tact )
  Initialization:       ( Wrist (≡wp) (≡nwp) ();
                          For i:=1..2
                            Tact_i () (Wrist(ts_i)) ();
                            Finger_i (Tact_i(c),≡fp_i) (≡nfp_i || Wrist(fp_i)();
                          Endfor;   )                                         ]
```

■

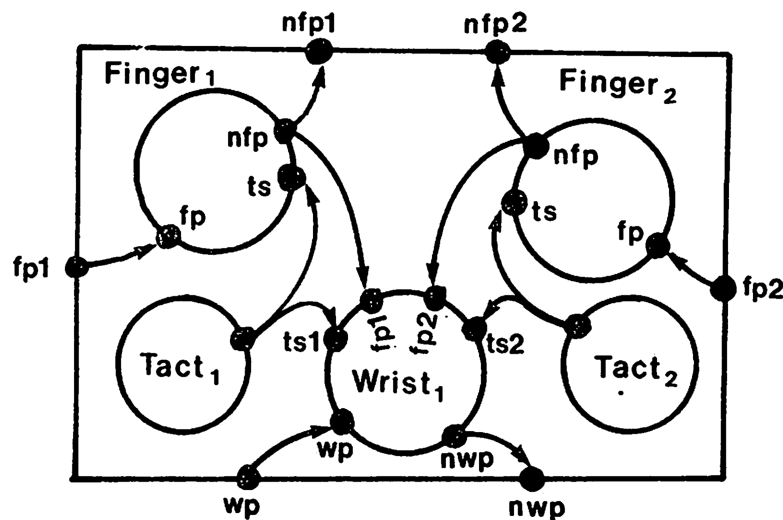


Figure 16: The Center Assemblage.

§2.6 *Classification Property:*

We define the classification property as the ability to communicate with all or some specific class of computing agents. Broadcast (where a messsage is sent to all agents) and multicast (where a message is sent to a specific subset of all agents) are examples of this, for a message-passing system.

As a motivating example, consider the problem we mentioned earlier with respect to classifying EOB SIs. For example, suppose we had some test C_m which could be used to determine, from the values of the ports of an EOB SI, if the object represented by that SI was a mug. We can use the Forall operation to carry out this classification as follows: Let us embody the test C_m into a schema TestCm which, when instantiated and connected to the ports of an EOB SI, will determine if that SI represents a mug. Its action, on success, is to create an instantiation of the MUG schema connected to the appropriate EOB SI; its action, on failure, is simply to deinstantiate. The Forall operation is then used to create (and connect) *one instantiation of* TestCm *for each instantiation of* EOB *as follows*:

```
Forall EOB_i :
    TestCm(EOB(f1),EOB(f2),EOB(f3))()();
Endforall
```

More formally we define the syntax of the Forall operations as:

Definition 20

Let there be some number of instantiations of the schema S, the numbering not necessarily contiguous, and let S be declared local to some assemblage, A. The Forall statement has the syntax:

Forall	::=	**Forall** S_i :
		⟨*Instlist*⟩ **Endforall**
Instlist	::=	Λ | ⟨*Inst*⟩⟨*Instlist*⟩

This statement uses a schema name as an *index* for iteration. The commands in the loop-body ⟨*Instlist*⟩ will be executed *once for each instantiation of S which currently exists in the assemblage* A. The statement can be in the body of A, or the body of a local SI of A. The scope of the Forall is bounded by the assemblage. i refers to an arbitrary instantiation number of S, and can be used throughout the scope of the Forall to describe generic actions to be performed on all instantiations of S. ∎

A primary use of the Forall statement is in constructing *preconditions* for task-unit assemblages. Consider the problem of instantiating some SI, T, when a certain object, or object configuration, is perceived. The object is characterized as a set of visual features; object perception is characterized as the values of the feature ports of an EOB SI. We define an SI Obj? whose function is to search all EOB instantiations in parallel for one whose feature port values correspond to values of internal variables of Obj?.

Obj? creates one instantiation of another SI Test for each instantiation of EOB; the internal variables of Test are initialized to describe the *desired* object, and the ports of Test are connected to ports on the corresponding EOB instantiation. Each Test_i also has a result output port connected to the input port *res* on Obj? (a fan-in situation).

Each Test_i simply tests the values it receives on its input ports against the values stored in its internal variables; if they match, a one is written to the result port, otherwise a zero is written to the result port. In either case Test_i terminates after testing EOB_i. Having started all Test instantiations, all Obj? needs to do is to wait for a one to be written to its input port *res*, and instantiate T_j when that happens.

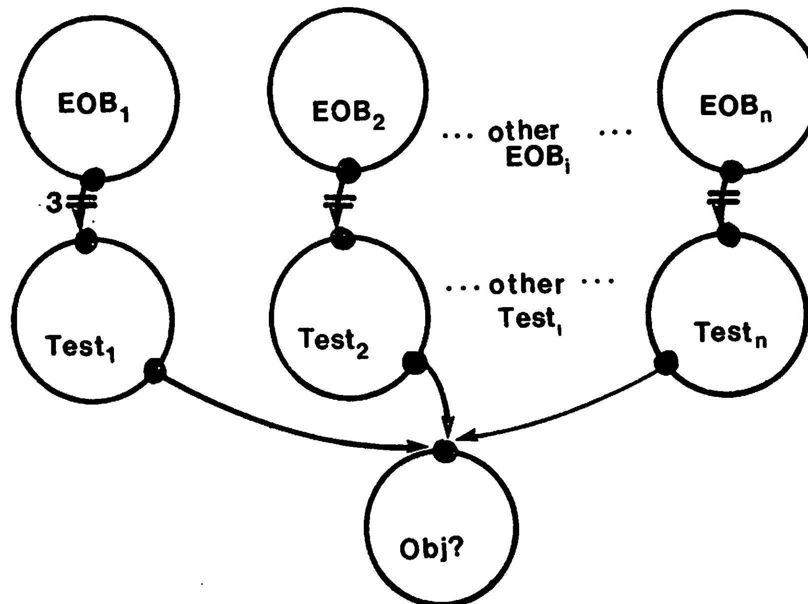


Figure 17: The Obj? Precondition.

Example 21

```
[   Test
    Input-Port-List:     ( f1...f4:Integer )
    Output-Port-List:    ( judgment:Integer )
    Variables:           (df1...df3:Integer )
    Behavior:            (If (f1 =df1)&...&(f3 =df3)
                             Then judgment:=1;
                             Else judgment:=0;
                           Endif;
                           Stop;  )                    ]

    [   Obj?
        Input-Port-List:     ( res:Integer )
        Output-Port-List:    ()
        Variables:           (  df1 ...df3:Integer )
        Behavior:            ( Forall EOBi :
                               Testi(EOB(f1),EOB(f2),EOB(f3))
                                           (res) (df1,df2,df3);
                              Endforall;
                              If res=1 Then
                               Tj(...)(...);
                               Stop;
                              Endif;    )                    ]
```

Note that Obj? will cycle through its behavior section continually creating Test processes. The unnecessary proliferation of Test processes is stopped since Obj? specifies the instantiation number of Test. In this way, if Obj? tries to create more than one SI to examine some EOB_i, we

can recognize the case. ■

To connect the instantiated SI T to the triggering EOB requires a minor modification to the previous example. Obj? creates one instantiation of Test for each instantiation of EOB and then dies. Each Test tests its EOB: If it fails the test the Test SI dies, otherwise the Test SI creates the task-unit. Since every particular instantiation of Test is connected to only one EOB, the SI can be connected to the triggering EOB. We can combine both of these approaches and have a precondition which waits around for some object to exist and when it does, creates an instantiation of some SI T connected to the EOB describing the object.

In all these examples, we have assumed that there will exist an EOB SI which represents the target object. This, of course, is a simplification. It may be necessary to explicitly search for an object by moving around the robot sensors. We shall not address here the issues involved in moving sensors to locate a particular object, although we recognize that this is an area which will need to be addressed. Indeed, in Chapter 6 we shall present a *perception axiom* which hides this aspect of perception. Future research with $\mathcal{RS}$ in this area can proceed by replacing this axiom with a much weaker one which allows this form of searching.

In Chapter 2, we discussed the virtual finger mechanism which allows us to specify motor actions in terms of some *logical* unit, the VF, which will later be mapped on to a real mechanism. One way to look at the VF translation mechanism would be as a form of variable fan-out: Motor commands issued to a VF are passed along to each physical finger which is considered part of that VF. But the important point is that we want to use the VF mechanism to specify *similar* responses to sensory input, as well as identical actions for members of the VF. For example, a guarded move command for a VF should translate to a guarded move command for each physical finger considered part of the VF.

Assume we have some task-unit N, describing action to be carried out by the VF (and hence, in turn, by each finger of the VF):

$$N = [T_i - F_i]$$

written in general terms of some finger control SI, $\mathtt{F}_i$, where i indicates which physical finger (according to some numbering of the robot mechanism) is being controlled, and is represented by some internal variable in N. The allotment of physical fingers to virtual fingers is a function of object and hand characteristics. The task to be accomplished is described in terms of VFs, which are mapped to appropriate physical fingers when the

task is executed for a particular object(s). So if N describes a task to be done, then the function of the virtual finger mechanism is to instantiate N for each finger F_j which is included in some designated virtual finger VF_k.

Example 22

Let us define a schema EVFk (Element of VF k), which we shall use to represent VF_k. Let EVFk have no input or output ports, no internal variables, and have the behavior of a null process.

```
[ EVFk
  Input-Port-List:    ()
  Output-Port-List:   ()
  Variables:          ()
  Behavior:           ()   ]
```

We indicate that some (not necessarily consecutive) set of fingers are to be included in VF_k:

$$\mathcal{F}_k = \{i \mid F_i \textit{ is considered to be in } VF_k \textit{ for this task}\}$$

by making a corresponding instantiation of EVFk for each $i \in \mathcal{F}_k$. The virtual finger to physical finger mapping is accomplished by making appropriate *local* instantiations of VFk. For example, if EVF2_1, EVF2_2 and EVF2_4 exist in some local context that indicates that for this context $\mathcal{F}_2 = \{1, 2, 4\}$. The Forall statement can be used to make one instantiation of N for each instantiation of EVFk:

$$\texttt{Forall}\ \ EVFk_i:\ \ N()()(i);\ \ \texttt{Endforall}$$

If, for example, $k = 2$ and $\mathcal{F}_2 = \{1, 2, 4\}$, then this will result in three instantiations of N with i equal to 1,2 and 4 respectively. Since i determines which finger N controls, the Forall above has the required virtual to physical mapping behavior. ■

In all our examples, we have used Forall to *create* SIs which carry out some requisite operations. For notational simplicity we could extend the definition of Forall so that its body can consist of any number or kind of statement. This body can then be compiled into a single schema. Execution of this extended Forall statement simply results in an instantiation of this schema being created for each instantiation of the index schema of the Forall statement. We choose not to use this upgraded Forall operation yet; we shall discuss this matter again in Chapters 5 and 10.

§2.7 *The Task-Unit Assemblage*

We differentiate the task-unit assemblage from the general definition of an assemblage. This is done to *structure the way programs are built in $\mathcal{RS}$*. Object models and task

programs are made of the same units: SIs. However, task programs have some constraints which can be usefully embodied in the task-unit assemblage.

A task-unit represents an 'atomic' robot task, and the components of the task-unit are general features of robot tasks at any level: An object model, primitive motor actions, the way in which these actions are parameterized by the object model, and triggering conditions for the task. We define a special syntax for task-unit specification only as a notational aid, since task-units are in all respects identical to any other assemblages. Task-units can be nested to form a sensori-motor hierarchy; the lowest nodes in such a hierarchy are the primitive sensory and motor schemas.

Definition 23

We use the following syntax to define a task-unit N:

```
[  N TASK-UNIT
   Input-Port-List:        (ip)
   Output-Port-List:       (op)
   Variables:              (v)
    Perceptual-Schemas:     (ps)
    Motor-Schemas:          (ms)
   Initialization:         (ib)    ]
```

- N,ip, and op are the name, input and output port lists of the task-unit assemblage, v is an internal variable list.
- ps and ms together compose the list of component schema of the task-unit.
- ib initializes the task-unit SI network.

■

Previous examples such as `move$joint`, `Jclose`, `Center`, etc., can simply be rephrased as task-unit assemblages. Let us consider a more complex example within our example domain of dextrous manipulation. Consider an object grasped in a *precision grasp*. This is described for any hand and object by the actions of three virtual fingers; these VF may, in turn, map to any number of physical fingers. A manipulation of the grasped object can be described in terms of the actions of these three VF. Let the transformation matrix M describe a manipulation in object-centered coordinates as described in Chapter 2; that is, if a matrix O describes the current position and orientation of the grasped object in some world coordinate frame, then the desired object configuration after manipulation is given by $M.O$, in world coordinates.

Consider the physical fingers in a particular virtual finger; consider that for this VF, the fingers are to maintain their relative object position during the manipulation (for example, rotating a ball between your finger-tips). For each finger, we can describe the desired new finger end-point position, to effect the desired manipulation, by simply transforming the current finger end-point by the matrix M. We can write this as a task-unit assemblage using i as a generic finger index; the VF precondition mechanism discussed in the previous section can then create one appropriately parameterized task-unit for each physical finger in the VF.

Example 24

Let `Fend` be a schema, such that the *ith* instantiation reports on the Cartesian end-point of the *ith* finger (in some prearranged numbering) in world coordinates through its output port `p:Vector`. Let `move$finger` be a schema such that the *ith* instantiation takes as input on its port `d:Vector` a new desired Cartesian end-point for finger i in world coordinates, and moves the finger end-point to that position.

```
[ FEND
  Input-Port-List:    ()
  Output-Port-List:   ( p:Vector )
  ...                                    ]

[ MOVEFINGER
  Input-Port-List:    ( d:Vector )
  Output-Port-List:   ()
  ...                                    ]
```

We first describe a schema *Manip* which will be the link between perception (`Fend`) and action (`move$finger`) in this task-unit. This schema takes a finger position on its port fe, a manipulation matrix on its port m, and produces the new desired finger end-point on it port nfe. To do this, it must have knowledge of the object transformation O. We introduce the following vector and matrix operations: $INVERSE(.)$ produces the inverse of a transformation matrix if it exists; $A.B$ premultiples matrix B by matrix A; $A : v$ multiples the vector v by the matrix A.

```
[ MANIP
  Input-Port-List:    ( fe:Vector m:Matrix )
  Output-Port-List:   ( nfe:Vector )
  Variables:          ( T_o:Matrix T_o^-1:Matrix )
  Behavior:           ( T_o^-1 := INVERSE( T_o );
                        nfe := T_o^-1.m.T_o : fe;  )  ]
```

We shall describe the task-unit *Fmanip*, the *ith* instantiation of which will input a manipulation matrix M as described above and in Chapter 2, on its input port `m:Matrix`, and cause the

ith finger to adopt the desired end configuration to effect the object manipulation described by *M*.

```
[  FMANIP ASSEMBLAGE
   Input-Port-List:          ( m:Matrix )
   Output-Port-List:         ()
   Variables:                ( i:integer )
    Perceptual-Schemas:      ( FEND )
    Motor-Schemas:           ( MOVEFINGER MANIP )
   Initialization:           (i:=#I;
                              Fend_i ()()();
                              Movefinger_i ()()();
                              Manip_i (Fend(p),≡m)(Movefinger(d))();
                                                                    ]
```

■

This ends the informal treatment of $\mathcal{RS}$. In the following chapter, we shall develop the formal syntax and semantics of the model, and investigate some useful properties.

CHAPTER V

FORMAL SEMANTICS

§1. Introduction

In the previous chapter, a syntax and informal semantics for $\mathcal{RS}$ was presented. In this chapter, the formal semantics is constructed. This construction will render our model well-defined in the sense that details omitted from informal semantics can be inferred from the mathematical objects which comprise the formal semantics. Equally important, verification and exploration of behavior cannot proceed formally until the precise effects of constructs in the model have been captured.

The importance of some formal aid in program development and verification has been recognized since the middle 70's [23, 1]. This need is exacerbated when *real-time control* is involved. Purely in terms of need therefore, one would imagine verification of robot programs to be in the forefront of the field. This is not true for a number of reasons. The two main ones being: There are so many other pressing research areas in robotics that verification has not surfaced yet, and when it does surface, robot models of computation are rarely so well developed that formal verification is possible.

In this chapter, we will develop an *operational semantics* for $\mathcal{RS}$ based on the *Port Automaton Model* of Steenstrup et al. [1983]. In Chapter 6, we shall use this automaton semantics as the interpretation for a Temporal Logic [57]. This chapter is concerned with the construction of the basic model of computation which is the semantics of $\mathcal{RS}$, while Chapter 6 is concerned with specifying and verifying the behavior of programs in the model.

This chapter is organized as follows: First, the *Port Automaton Model* is presented, and a version of it constructed which is appropriate for our purposes; second, the mapping from the syntax of $\mathcal{RS}$ to this version of the port automaton model is presented; this mapping is the formal semantics of the distributed model; finally, we present several interesting extensions to the basic structures of the $\mathcal{RS}$ model.

§2. The Port Automaton Model

In this section, Steenstrup et al.'s port automaton model is presented. A version of their basic port automaton is constructed which will act as the mathematical object which we will use as the semantics of a schema. We begin by presenting their definition of a port automaton and a port connection.

Definition 25

A *port automaton* is a collection of objects and maps:

$$P = (L, Q, X, Y, \tau, \delta, \beta) \tag{V.1}$$

where,

L is the set of *ports*,

Q is the set of *states*,

$X = (X_i : i \in L)$, where X_i is the *input set* for port i,

$Y = (Y_i \cup \{\perp\} : i \in L)$, where Y_i is the *output set* for port i,

$\tau \in 2^Q$ is the set of *initial states*,

$\delta : Q \times \bigsqcup_{i \in L} X_i \to 2^Q$ is the *transition function*,

where $\bigsqcup_{i \in L} X_i = \{(x, i) : x \in X_i\}$ is the disjoint union of the X_i's,

$\beta = (\beta_i : i \in L)$, where $\beta_i : Q \to Y_i$ is the *output map* for port i.

Notes:

1. All subject to the *port activation axiom* (for technical simplicity) that for each $q \in Q$

$$\{x \in X_i : \delta(q, (x, i)) \neq \emptyset\} = \emptyset \text{ or } X_i \tag{V.2}$$

that is, in state q either *all* inputs on port i can be accepted or *none* can.

2. Port i of P is *activated* in state q if forall $x \in X_i$, $\delta(q, (x, i)) \neq \emptyset$. Otherwise, port i is *inactivated*; and $\beta_i(q) = \perp$ *for* $\delta(q, (x, i)) = \emptyset$.

3. *Communication* over a port consists of an *exchange* of values through the port; one or both of these values may be the *trivial value*, denoted by '#' (we shall expand mightily upon this definition presently). ■

Definition 26

This definition presents Steenstrup et al.'s port connection automaton, and is summarized from [80]. Let P^1 and P^2 be two port automata, where

$$P^j = (L^j, Q^j, X^j, Y^j, \tau^j, \delta^j, \beta^j), \quad j = 1,2 \tag{V.3}$$

Let $S^j \subseteq L^j$. The map $c : S^1 \to S^2$ is a port connection map if c is a bijection such that for each $i \in S^1$, $Y_i^1 \subseteq X_{c(i)}^2$, $Y_{c(i)}^2 \subseteq X_i^1$. We call i and $c(i)$ *complementary ports*; $c(i)$ is the complement of i. This is illustrated in Figure 18.

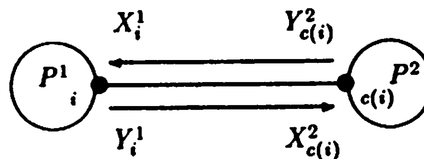


Figure 18: The Port Connection Map.

A port connection of two port automata yields a port automaton such that the connected ports are subsumed and no longer visible to the environment. The port connection of P^1 and P^2 for port connection map c is $P^1\|_c P^2$ a port automaton $P = (L, Q, X, Y, \tau, \delta, \beta)$, where

$$\begin{array}{rcl} L & = & (L^1 - S^1) \cup (L^2 - S^2) \\ Q & = & Q^1 \times Q^2 \\ X & = & (X_i : i \in L), \\ & \textit{where} & X_i = X_i^j, i \in L^j - S^j \\ Y & = & (Y_i : i \in L), \\ & \textit{where} & Y_i = Y_i^j, i \in L^j - S^j \\ \tau & = & \tau^1 \times \tau^2 \in 2^Q \end{array} \tag{V.4}$$

$$\beta_i \;=\; Q \xrightarrow{pr_1} Q^j \xrightarrow{\beta_i^j} Y_i^j \quad \textit{for} \; i \in L^j - S^j,$$

where pr_i represents the projection map from a set product onto its ith component.

We next determine if it is possible to express the transition function of the port connection as a port automaton transition function. We must derive an expression for δ which captures the interactions of P^1 and P^2. Exploiting the symmetry in port automaton communication, we assume $i \in L^1 - S^1$; the complementary case is done similarly.

In a network of port automata, *internal* communications may take place in the interval between *external* communications. To determine what internal communications are possible between

P^1 and P^2, we must know which of the connected ports are activated in some state $q \in Q$; let these be described by the set Z_q, the set of S^1-components of the set of simultaneously activated pairs of complementary ports. Let $q = (q^1, q^2)$:

$$\begin{array}{rcl} Z_q & = & \{k \in S^1 : \exists x \in X^1_k \textit{ such that } \delta^1(q^1,(x,k)) \neq \emptyset, \\ & & \textit{and } \exists \bar{x} \in X^2_{c(k)} \textit{ such that } \delta^2(q^2,(\bar{x},c(k))) \neq \emptyset\} \end{array} \quad \text{(V.5)}$$

Let us observe what happens to P^1 and P^2 as a result of one internal communication via complementary ports k and $c(k)$, where $k \in Z_q$. We define the function $\hat{\delta} : Q \rightarrow 2^Q$ as:

$$\begin{array}{rcl} \hat{\delta}(q) & = & \hat{\delta}(q^1,q^2) \\ & & \{(\bar{q^1},\bar{q^2}) : \bar{q^1} \in \delta^1(q^1,(\beta^2_{c(k)}(q^2),c(k))), \\ & & \textit{and}, \bar{q^2} \in \delta^2(q^2,(\beta^1_k(q^1),k)), k \in Z_q\} \end{array} \quad \text{(V.6)}$$

Since multiple internal communications can occur between two external communications, we must extend $\hat{\delta}$ to its transitive closure, $\hat{\delta}^*$. For $j \geq 0$, let $\hat{\delta}^j$ be equivalent to the set of states reachable from q by j internal state transitions:

$$\begin{array}{rcl} \hat{\delta}^0(q) & = & \{q\} \\ \hat{\delta}^j(q) & = & \hat{\delta}^{\#}(\hat{\delta}^{j-1}(q)) \\ & & \textit{where, } ^{\#}() \textit{ is extension by union} \\ \hat{\delta}^*(q) & = & \bigcup_{j=0}^{\infty} \hat{\delta}^j(q) \end{array} \quad \text{(V.7)}$$

$\hat{\delta}^*$ represents *all* states reachable from q as a result of exclusively internal communication. From this we can define the state transition directly affected by the input on port i. First we define this transition function for a single state $q \in Q$ as:

$$\tilde{\delta} : Q \times \bigsqcup_{i \in L} X_i \rightarrow 2^Q \quad \text{(V.8)}$$

where,

$$\begin{array}{rcl} \tilde{\delta}(q,(x,i)) & = & \tilde{\delta}((q^1,q^2),(x,i)) \\ & = & \{(\bar{q^1},\bar{q^2}) : \bar{q^1} \in \delta^1(q^1,(x,i))\} \end{array} \quad \text{(V.9)}$$

Note that this definition implies that the port activation axiom which holds for P^1 and P^2 expands naturally to hold for their port connection P. The complete definition of the transition function is achieved by extending $\tilde{\delta}$ to the set of states obtained from internal transitions.

$$\delta(q,(x,i)) = \tilde{\delta}^{\#}(\hat{\delta}^*(q),(x,i)) \quad \text{(V.10)}$$

This formalizes the conjecture that a port connection of two port automata should be a port automaton. ∎

Definition 27

Steenstrup et al. define communication within the port automaton model to occur as follows: Each port automaton has a set of ports through which all communication takes place. Communication consists of a simultaneous exchange of values through a port; that is, a process *receives* a value from the environment at the same time as the process *sends* a value to the environment; both sending and receiving occurring at the same port. The flow of information may be *unidirectional* (a fact we attend to in detail later), in that one of the exchanged values may be the trivial value '#'. In order for communication to occur on complementary ports i and $c(i)$, from P^1 to P^2, the following conditions must hold simultaneously for P^1 and P^2.

P^1	P^2
in state q^1 $\beta_i(q^1) = y^1$ $\delta(q^1, (y^2, i)) \neq \emptyset$	*in state* q^2 $\delta(q^2, (y^1, c(i),)) \neq \emptyset$ $\beta_{c(i)}(q^2) = y^2$

(V.11)

In the case where no other port except i is *activated*, P^1 cannot undergo any internal state transition until some data is received on port i. This blocking action gives this type of communication, *Synchronous Communication*, its name [4].

In contrast, in *Asynchronous Communication*, a sender transmits its message regardless of the state of the receiving process, and does not wait to receive an acknowledgement from the receiver. In a similar fashion, a receiving process which wishes to receive a message by asynchronous communication will terminate directly with a special error return if no message had been sent – no blocking is involved, and if a message is received, then no acknowledgement need be returned.

Synchronous and *Asynchronous* denote classes of communication mechanisms, rather than particular methods of communication. Port automata communication can be classified as a form of synchronous communication where:

1. All ports have both transmit and receive abilities.
2. The receiving process can respond to a transmitted message by either a trivial response ('#') or a non-trivial message.
3. Ports are communication objects with the following properties:
   (a) *Single-Buffered*; they can hold a single message for transmission or reception.
   (b) *Structured*; the input and output sets, X and Y, allow *data types* to be associated with ports. Only ports which have the same type can be connected by a communication

link, that is, two ports can only be connected if their input and output sets are matched. ∎

Definition 28

We extend the definition of the port connection automaton to define the *network automaton* as follows. The network automaton of a set of m port automata $\{P^j : j = 1, \ldots, m\}$ and a set of port connection maps $\{c_k : k = 1, \ldots, m-1\}$ is a single port connection automaton constructed in the following manner: Select P^1 and P^2, let $\tilde{P}^1$ be the port connection automaton $P^1 \|_{c_1} P^2$. Continue constructing port connection automata in the ordered sequence; $\tilde{P}^j = \tilde{P}^{j-1} \|_{c_j} P^{j+1}$ until $j = m-1$. This final port connection automaton $\tilde{P}^{m-1}$ is the network automaton. By Definition 26, it is also a port automaton. This connection scheme is illustrated in Figure 19.

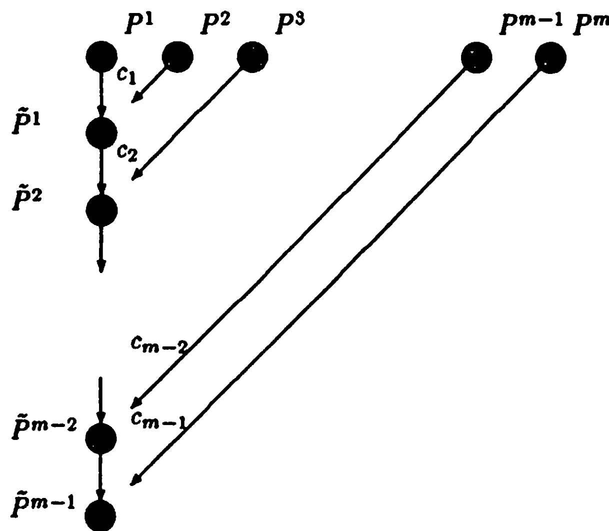


Figure 19: Construction of the Network Automaton.

Theorem 29

For a set of m port automata, a single network map can be constructed which carries the same information as the set of port connection maps described in Definition 28, but in terms of connections between the component automata, not their port connection automata.

To see this difference consider Figure 20(a), which is the port connection version, versus Figure 20(b), which is the network map version of a network of port automata.

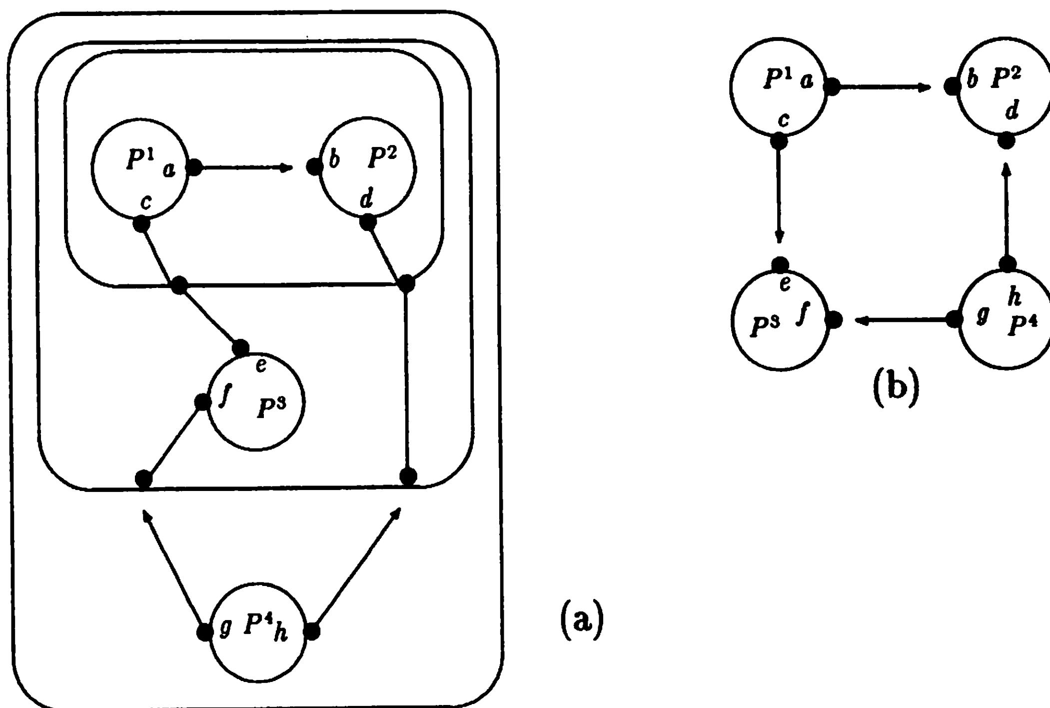


Figure 20: Network Described by Port Connections (a) Versus Network Map (b).

Consider the sequence of port connection automata described by Definition 28; let us write this:

$$\tilde{P}^k = \tilde{P}^{k-1}\|_{c_k} P^{k+1}, k = 1 \ldots m-1, \ \ and \ \ \tilde{P}^0 = P^1 \tag{V.12}$$

where $\tilde{P}^k$ are the port connection automata, and P^{k+1} are the component port automata, and it is understood that the sequence of definitions of port connection automata starts with $k = 0$ and proceeds sequentially to $m-1$. Let us superscript all the components of a port automata $P^j = (L^j, Q^j, X^j, Y^j, \tau^j, \delta^j, \beta^j)$, $j = 1 \ldots m-1$. In that case, for some particular port connection automaton, $\tilde{P}^k$; in this sequence, its ports are given by the equation:

$$\tilde{L}^k = \bigcup_{i=0\ldots k-1} (\tilde{L}^i - \tilde{S}^i), \quad \tilde{L^0} - \tilde{S^0} = L^1 - S^1 \tag{V.13}$$

where S^i are the ports subsumed by communication in P^i. This equation describes the ports of $\tilde{P}^k$ as the set of all ports not yet subsumed by communication on all the component port automata which were connected to yield $\tilde{P}^k$.

From this we shall construct the network map n which relates a pair denoting an automaton and a port to another pair denoting an automaton and a port, and is to be interpreted as meaning that there is a communication connection between these ports:

$$n : \bigsqcup_{k \in \{1,\ldots,m\}} S^k \rightarrow \bigsqcup_{j \in \{1,\ldots,m\}} S^j \tag{V.14}$$

subject to the restriction that:

$$if \ n(k,i) = (j,h), \ then \ Y_i^k \subseteq X_h^j \ and \ Y_j^h \subseteq X_i^k \tag{V.15}$$

where i and h are complementary ports in the sense of Definition 26. To construct n we inspect (V.12) as k goes from 0 to $m-1$, and at each k consider each $\ell \in \tilde{L}^k$ as described by (V.13) and construct n pairwise:

$$n(i,\ell) = (k+1, c_k(\ell)) \tag{V.16}$$

Since for each connection described in the set of port connection maps, n also describes the same connection, and n consists of only these connections, the networks described by n and by Definition 28 are the same.

This process can be reversed to generate a set of port connection maps from the network map. Proceed as follows: From n determine the mutual connections between P^1 and P^2, and construct $c_1 : S^1 \rightarrow S^2$

$$for \ all \ i \in S^1, \ j \in S^2, \ if \ n(1,i) = (2,j) \ then \ c_1(i) = j \tag{V.17}$$

Construct the port connection automaton $\tilde{P}^1 = P^1 \|_{c_1} P^2$. Isolate all the connections in n between the nonsubsumed ports of $\tilde{P}^1$ and P^3, from these generate the map c_2. Continue this process to generate the $m-1$ connection maps. ∎

We shall now proceed to construct the version of the Steenstrup et al. port automaton which we shall use as the semantics of an SI. The basic change we make is the introduction of separate *input and output* ports. Although we label these as input and output ports, they are still bidirectional ports as in Definition 25. However, input ports are constrained to transmit only the *null value* (they can receive any value in their input set), and output ports are constrained to receive only the null value (they can transmit any value in their output set). This change has two advantages: It allows an easier mapping to the schema descriptions of Chapter 4, and it allows us to formalize synchronous communication more explicitly than in Definition 27. This formalization will allow us to construct several useful extensions to communication in $\mathcal{RS}$.

Definition 30

In this definition a version of Steenstrup et al.'s port automaton is constructed which will be the basis for our formal semantics. The main change is the introduction of separate input and output ports. For technical simplicity we also restrict the port type (see Note #2).

A *port automaton* is a collection of objects and maps

$$P = (L_x, L_y, Q, X, Y, \tau, \delta, \beta)$$

where,

L_x is the set of *input ports*, and

L_y is the set of *output ports*,

and we use L to denote $L_x \cup L_y$.

Q is the set of *states*,

$\tau \in 2^Q$ is the set of *initial states*,

$X = (X_i : i \in L)$,

where X_i is the *input set* for port i,

and where $X_i = \{\#\}$ for $i \in L_y$,

$Y = (Y_i : i \in L)$,

where Y_i is the *output set* for port i,

and where $Y_i = \{\#\}$ for $i \in L_x$.

$\delta : Q \times \coprod_{i \in L} X_i \to 2^Q$ is the *transition function*,

where $\bigsqcup_{i \in L} = \{(x, i) : x \in X_i\}$ is the disjoint union of the X_i's,

$\beta = (\beta_i : i \in L)$, where $\beta_i : Q \rightarrow Y_i$ is the *output map* for port i.

Notes:

1. Notes 1 to 3 of Definition 25 apply, with the important change that in Note 3, all communication proceeds with a flow of information from output port to input port and a return of the trivial value from input port to output port. This is formalized by the following two *synchronous communication axioms:*

$$\begin{array}{ll} \forall q \in Q,\ \exists i \in L_x,\ x \in X_i,\ \delta(q,(x,i)) \neq \emptyset & \supset \beta_i(q) = \# \\ \forall q \in Q,\ \exists i \in L_y,\ y \in Y_i,\ \beta_i(q) = y & \supset \delta(q,(\#,i)) \neq \emptyset \end{array} \quad (V.18)$$

2. For *all* port automata we set:

$$X_i \in \mathcal{T},\ i \in L_x,\ \ and\ \ Y_i \in \mathcal{T},\ i \in L_y \quad (V.19)$$

where by $\mathcal{T}$ we denote the set of legal port types in $\mathcal{RS}$; in Chapter 4 we have set:

$$\mathcal{T} = \{Real, Integer, Matrix, Vector\} \quad (V.20)$$

This *prohibits* input ports from being connected to input ports, since for every port automaton:

$$(X_i \in \mathcal{T})\ \neq\ (Y_i = \{\#\}), for\ \ i \in L_x$$

This same mechanism prohibits output ports from being connected to output ports. ∎

In order to satisfy ourselves that no power has been lost by the introduction of separate input and output ports, it is necessary to show that for any port automaton with bidirectional ports (bPA), it is possible to construct a port automaton with unidirectional ports (uPA) which, in some defined sense, 'does the same computation', and viceversa. We formalize 'does the same computation' by the concept of input-output trace.

Definition 31

A *port event* is a tuple (i, x, y); where i is a port name and x and y are data values which are simultaneously transmitted ($y \in Y_i$) and received ($x \in X_i$) on that port. ∎

Definition 32

An *Input-Output Trace*, B, of a port automaton P is a finite sequence of port events (i, x, y) over the ports of P. A port automaton is charaterized by its set of traces. The *length* of a trace

B is the number of events in it, and is denoted $|B|$. A trace of length n is also written B_n. All processes have the empty trace B_0 in their set of traces. ∎

If we consider a port automaton which is the port connection of two port automata, then it is clear that two ports on the port connection may simultaneously engage in communication (e.g., one port on each port automaton). Two such simultaneous events are recorded in the input-output trace for the port connection in some non-determinate order. If n such simultaneous events occur, then we represent this in the input-output trace as an non-deterministic interleaving of the n events.

Definition 33

For any two traces B and B', $B = B'$, if they are the same length and if the *ith* event in B is equal to the *ith* event in B'. Two events are equal if they are on the same port and if they exchange the same data values. We describe the trace B' obtained from trace B by a port renaming function F as B/F; where F substitutes for portnames $i, j, k, \ldots$ portnames $a, b, c, \ldots$. Two traces B and B' are considered equal if there exists a bijective renaming function F, such that $B' = B/F$. ∎

A cruicial point in understanding the connection between the trace of a uPA and a bPA, is the connection between the trace of length 1; $B_1 = ((i, x, y))$, and the trace of length 2 $B'_2 = ((i, \#, y), (i, x, \#))$. [1] Any information contained in the event in B_1 is also contained in the two events in B'_2.[2] We shall say that traces which are alike in the way that B_1 and B'_2 are, are *equivalent* traces.

Definition 34

Two traces B and B' are equivalent if $B = U(B')$, where we let $B_{1:n}$ denote the trace B with events $e_1, \ldots, e_n$, and U is defined as:

$$
\begin{array}{ll}
B_{0:n} = U(B'_{0:m}) & if\ n = m = 0 \\
 & or\ \exists k \leq m\ \ B_{k:n} = B_{k:m}, and \\
 & \quad e'_{k-1} = (i, x, y)\ in\ B',\ and \\
 & \quad e_{k-2} = (i, \#, y), e_{k-1} = (i, x, \#)\ in\ B,\ and \\
 & \quad B_{0:k-3} = U(B'_{0:k+2})
\end{array}
$$

∎

---

[1] Where i, x, y have the same values in B_1 and B'_2.

[2] Indeed, B'_2 contains more information, since the two events could be used to indicate the start and end of some operation.

Observation 35

If $B = U(B')$, then $|B| \geq |B'|$. This comes directly from the definition of U; if the traces are equal, then they have the same length, otherwise B is at least one event longer than B'. ∎

Definition 36

Let i be a port on a port automaton P. Let B be a trace of P. Consider the trace B restricted to just the events on some port i. If all the events in this restricted sequence of events are of the form $(i, \#, y)$ or $(i, x, \#)$, then we say that i is a *separable* port. ∎

Intuitively, a separable port is a port on which data is only ever passed in one direction; however, the direction may change from event to event. This is the sort of port usage which Steenstrup et al. use in their examples.

Definition 37

If a trace B has separable ports $a, b, c, \ldots$, then the events in the trace which consist of just an input of information $(a, x, \#)$ and those with just an output of information $(i, \#, y)$ can be renamed separately without changing the trace. Let FI denote a function which maps $a, b, c, \ldots, \ldots$ to $a_x, b_x, c_x, \ldots$, just in the case where $a, b, c, \ldots$ are separable ports, and just in those events where there is an input of information; then we say $B = B/FI$. ∎

To see why this definition holds we must look at the transition map for a port automaton. Let us be in a state q such that $\delta(q, (x, i)) = \{\bar{q}\}$ and $\beta_i(q) = \#$, and let $\bar{q}$ be a state where $\delta(\bar{q}, (\#, i)) = \tilde{q}$, and $\beta_i(\bar{q}) = y$. That is a state q in which there is an input of information on i followed by a state $\bar{q}$ on which there is an output of information on i. It is clear that a new port j could be substituted (in δ and β) [8] for the port i in the first state, and as long as j had not already been defined in δ and β, and as long as $\bar{q}$ is the next state, it does not change anything. The input and output events on a separable port do not *interfere* with each other, in the same sense that input and output on different ports do not interfere with each other.

Observation 38

If $B = B'/FI$ for any trace B of port automaton P and B' of port automaton P', for some fixed FI, then the cardinality of L, the set of ports in P, must be greater than or equal to that of L'. This comes directly from definition of FI. Either B' will have no input events, in which case no port renaming will be done, and the trace B has exactly the same ports named in it as B'. Otherwise, there is at least extra port i_x which comes from renaming the port in the input

---

[8]The environment would have to know of course.

event on port i to i_x. In this case there is at least one more port named in B than B', and any port automaton of which B is a trace must have at least one more port than P'. ■

We can now estimate the effect of restricting ourselves to dealing with port automata with defined input and output ports, uPA, rather than the more general port automata with bidirectional ports, bPA.

Theorem 39

For any uPA we can construct a bPA, such that if B is a trace of the uPA, then B is also a trace of the bPA.

This is trivial, since we have defined uPA to be special kind of bPA. Hence, any trace of a uPA is, by definition, also the trace of a bPA. ■

Theorem 40

For any bPA we can construct a uPA, such that if B is a trace of the bPA, then the uPA has an equivalent trace $U(B)/FI$, for a simple separable port renaming function FI.

We prove this as follows: We give a construction to generate an appropriate uPA, P', from a bPA P. In the course of this, we define the renaming function FI. We then prove by induction on trace length that, if B is a trace of P, then $B' = U(B)/FI$ is a trace of P'.

Construction of P': Let the constructed P' have the same set of states and set of initial states as the original P, we will then alter the transition and output maps of P in an appropriate fashion.

$$\begin{aligned} Q' &= Q \\ \tau' &= \tau \end{aligned} \tag{V.21}$$

Next we traverse L and construct L'_x and L'_y as follows:

1. If for all $q \in Q$, $\beta_i(q) = \#$, then i is a port used only for input communications, therefore include i_x in L'_x, and alter δ and β so that instead of i, i_x is named.

2. If for all $q \in Q$, $\beta_i(q) \neq \#$, but $\{x \in X_i : \delta(q,(x,i))\} = \{\delta(q,(\#,i))\}$, then i is a port used only for output communications, therefore include i in L'_y.

3. If there exists a $q \in Q$ such that $\beta_i(q) \neq \#$ and $\{x \in X_i : \delta(q,(x,i))\} \neq \{\delta(q,(\#,i))\}$, then this port i is a port which is used for bidirectional communication, so we must construct extra ports in P' to handle just the output part and just the input part of its communications. For each such i construct its *input equivalent port* i_x and its *output equivalent port*

i, and for each $q \in Q$ in which i is *active*, augment Q to include some $\tilde{q}$, and augment δ and β such that:

$$\delta'(q,(x,i_x)) = \{\tilde{q}\} \qquad \beta'_{i_x}(q) = \# \tag{V.22}$$

$$\delta'(\tilde{q},(\#,i)) = \delta(q,(x,i)) \qquad \beta'_i(\tilde{q}) = \beta_i(q) \tag{V.23}$$

and add i to L'_y and i_x to L'_x.

4. Let L'_x consist only of ports added by 1 and 3a above.
   Let L'_y consist only of ports added by 2 and 3b above.
   Let δ' and β' only be augmented from δ and β by 3 above.

P' is a uPA: The automaton P' thus constructed from P obeys the restrictions of Definition 30. To prove this consider δ' and β'. Ports can be members of L'_y by Condition 2, in which case for all $q' \in Q'$, $\{x \in X_i : \delta(q,(x,i))\} = \{\delta(q,(\#,i))\}$ already, or else they are the output equivalent port of some port i in which case V.23 above enforces this condition. In similar fashion, ports can only be members of L'_x by Condition 1, in which case, for all $q' \in Q'$, $\beta'_i(q') = \#$ already, or else they are the input equivalent port of some port i, and V.22 above enforces this condition. Hence this automaton is a uPA.

Definition of FI: (From the above construction) Let FI be a separable port renaming function which renames ports $i, j, \ldots$ to $i_x, j_x, \ldots$, just in the case that they are each separable ports, and just in the events in which they are involved in the input of information.

Inductive Proof of Theorem: We consider any trace B_0 of the bPA P of length zero of P, and show that there is a behavior of length zero B'_0 of the uPA P' such that $B'_0 = U(B_0)/FI$. This is the basis step. We then show that, if for some traces B_n of P and B'_m of P' the theorem holds and $B'_m = U(B_n)/FI$, then, if P extends its trace by one event e_{n+1} to give B_{n+1}, then there is a trace of P' which extends B'_m by at most two events for which the theorem again holds $B'_k - U(B_{n+1})/FI$, where $m+1 \leq k \leq m+2$. This is the inductive step.

Basis Step: All traces of length 0 are equal, hence the theorem holds initially.

Inductive Step: Assume that the theorem holds for a trace B_n of P and B'_m of P'. If P extends its trace by one event to give B_{n+1}, then this event is either:

1. $(i, x, \#)$ input only.
2. $(i, \#, y)$ output only.
3. (i, x, y); input and output.

If the event is 1. above: By 1. of our construction, the uPA trace is to extend B'_m by the same event. Since up to n and m U holds on these traces, and $e_{m+1} = e'_{m+1}$ (ports used just for the output of information will have the same names under FI), then we have $B'_{m+1} = U(B_{n+1})/FI$.

If the event is 2. above: By 2. of our construction, the uPA trace is to extend B'_m by the event $e'_{m+1} = (i_x, x, \#)$; we have preserved the theorem by the same argument as above, except that FI maps i to i_x in B'_{m+1}, and again we have $B'_{m+1} = U(B_{n+1})/FI$.

If the event is 3. above: By 3. of our construction, the uPA trace is to extend B'_m by two events $e_{m+1} = (i, \#, y), e_{m+2} = (i, x, \#)$. The definition of U says U holds over all B and B' if it holds over our extensions above, and if it holds over the remainder of each trace. Our inductive assumption is that it holds over the remainder, and the above construction ensures that U holds for the last event of B and the last two events of B'. Hence it holds that $B'_{m+2} = U(B_{n+1})$. ∎

We have formalized the differences in trace between bPA and uPA, and we have satisfied ourselves that we can construct an equivalent uPA for any bPA. Observations 35 and 38 tell us that the constructed uPA may need more ports and more communication events then the bPA.

§3. Construction of the Formal Semantics

We have now completed the version of the port automaton model we shall use to construct the semantics of our sensory-based model of distributed computation. The next step is to construct the mapping from the syntax, described in an informal and expository way in the previous chapter, to the port automaton model. For technical clarity in the definition of a state vector, we restrict the set of legal port data-types $\mathcal{T}$ to be the set of computable reals [43], denoted by $\Re$:

$$\mathcal{T} = \{\Re\} \qquad \text{(V.24)}$$

For clarity we preface these formal semantics with a review of the syntax.

Definition 41
We can abbreviate the syntax of schema definition of Chapter 4, Definition 3, by omitting the keywords:

$$[\ N\ (IP)\ (OP)\ (V)\ (B)\ ]$$

where,

- N is a unique name for this schema,
- IP is a list of input port names,
- OP is a list of output port names,
- V is a list of internal variable names,
- B is a program description of the computational behavior of the schema.

All 5 elements of the schema description *must* be present, however, except for the name, all may be empty, which we denote by the empty parenthesis '()'. The input port list, output port list and variable list are of the form:

$$\begin{array}{lcl} list & ::= & \wedge \mid \langle pair \rangle \langle list \rangle \\ pair & ::= & \langle name \rangle : \langle type \rangle \\ type & ::= & t, \text{ such that } t \in \mathcal{T} \end{array}$$

In the *ip* and *op* lists *name* declares the name of an input port or output port respectively. In v, name declares an internal variable name. We have defined $\mathcal{T}$ to be the set of legal data types in $\mathcal{RS}$; *type* for both *ip*, *op* and v lists must be an element of $\mathcal{T}$ under V.24.

The program description of the schema consists of sequence of statements which are executed in a continuous cycle (an implicit 'while true do' loop). We define the behavior section by the following syntax:

$$\begin{array}{lcl} Behavior & ::= & \wedge \mid \langle Stat \rangle \langle Behavior \rangle \\ Stat & ::= & \langle Assign \rangle \mid \langle If \rangle \mid \langle Instn \rangle \mid \langle Dinstn \rangle \\ & & \mid \langle For \rangle \mid \langle Forall \rangle \\ Assign & ::= & \langle Var \rangle := \langle Expression \rangle \mid \end{array}$$

A statement *Stat* can be any one of the following:

Assignment

An assignment statement has the form:

$$\langle Var \rangle := \langle Expression \rangle$$

where *Expression* is an arithmetic statement consisting of:

- arithmetic operators $(+, -, /, *, **, (,), mod, div)$, that is: add, subtract, divide, multiply, power, parentheses, and integer remainder and division.

- real number constants.
- internal variable names.
- input port names.

An assignment statement has the following syntax:

Var	::=	$\langle variable-name\rangle$	$\mid \langle output-port-name\rangle$
$Expression$	::=	$\langle Term\rangle$	$\mid \langle Term\rangle\ \langle OP1\rangle\ \langle Expression\rangle$
$OP1$	::=	$+ \mid -$	
$Term$	::=	$\langle Factor\rangle$	$\mid \langle Factor\rangle\ \langle OP2\rangle\ \langle Term\rangle$
$OP2$	::=	$/ \mid * \mid div \mid mod$	
$Factor$	::=	$\langle Primary\rangle$	$\mid \langle Primary\rangle * * \langle Factor\rangle$
$Primary$	::=	$(\langle Expression\rangle)$	$\mid -\langle Expression\rangle \mid$
		$\langle real-number\rangle$	$\mid \langle variable-name\rangle \mid$
		$\langle input-port-name\rangle$	

Informal Semantics:

If a variable name occurs in the expression, then the value stored at that variable is used to evaluate the expression. If an input port name occurs in an expression, then a read occurs on that port, and the value returned is used to evaluate the expression. Var is the name of an internal variable in the variable list V. The assignment statement is interpreted as meaning that *expression* is evaluated according to standard arithmetic rules and the result placed in the internal variable denoted by Var. If Var is an output port name, then the assignment statement is understood to mean that the expression is evaluated and written to this port.

Definite Iteration

$$\textbf{For}\quad \langle Var\rangle := LB \ldots UB \quad \langle Body\rangle \quad \textbf{Endfor}$$

where,

- Var is the name of an internal variable,
- LB and UB are integer expressions,
- where $Stat$ is any valid program statement.

$$Body ::= \wedge \mid \langle Stat\rangle\ \langle Body\rangle$$

The nonterminal *Body* describes a sequence of zero or more valid statements, we shall use *Body*, as defined above, in later constructs also. We refer to the statements of *Body* as the *body* of the loop, to Var as the loop *index variable* and to LB and UB as the *lower* and *upper bounds* of the

loop respectively.

Informal Semantics:

The statement is interpreted as meaning that the body of the loop is to be executed *at least once*[4], with Var given the value LB, and then repeated incrementing Var to value $Var + 1$, if $UB > LB$ or $Var - 1$, if $UB \leq LB$, but terminating when $Var = UB$.

Conditional

```
If (⟨Booleanop⟩) Then ⟨Body1⟩ Endif
If (⟨Booleanop⟩) Then ⟨Body1⟩ Else ⟨Body2⟩ Endif
```

where,

- *Body*1 and *Body*2 are sequences of zero or more statements as defined earlier.
- *Booleanop* consist of a sequence of *Comparisons* connected by any of the logical connectives: & (and), ! (or), ¬ (not).
- *Comparison* is defined as:

$$\langle Expression1\rangle \ \langle COP\rangle \ \langle Expression2\rangle$$

  where,

  - *Expression*1 and *Expression*2 are arithmetic expressions and,
  - *COP* is any one of the standard comparison operators $(=, >, <, \neq, \geq, \leq)$.

Informal Semantics:

The conditional statement is interpreted as meaning that *Booleanop* is to be evaluated, and if true then the statements in *Body*1 are to be executed, followed by the execution of the first statement after the `Endif`. If *Booleanop* evaluates to false then the statements in *Body*2 are to be executed if there are any, to be followed by the execution of the first statement after the `Endif`.

Instantiation and Deinstantiation

Postponed until Definition 47.

---

[4]The body executes once if $LB = UB$, twice if $LB = UP \pm 1$, etc.

FORALL

Postponed until Definition 56.

■

In the next section, we shall establish a port automaton as the semantics of a schema instantiation. Before we do this, we need to introduce some concepts. Our approach in this section will be operational, in keeping with the port automaton model. However these same definitions will provide the basic form of the *transition axioms* [56] employed in a later section for verification purposes. First let us consider how the internal state of a schema can be represented.

The internal variable list of schema N is a list of variable names $(V_1, V_2, \ldots, V_k)$, denoting the internal variables of N, which we shall call a k−variable schema. Each variable V_i has a value v_i. At any stage, we can write out the contents of all the variables of N as $(v_1, v_2, \ldots, v_k) \in \Re^k$, because of equation V.24.

We assign a unique label or index to all the statements in the program description part of N from the set of natural numbers by scanning the statements of N in sequential fashion, starting at the first statement after the opening parenthesis of the program specification, and finishing with the last statement before the closing parenthesis. If the largest label assigned to N in this fashion is n, then we refer to N as a k−variable schema of length n.

Definition 42

A *state of computation* for a k−variable schema is a k dimensional vector over the reals [5]. We shall also call it a state vector. ■

Definition 43

A *port event* on a schema instantiation N_j is a tuple; where i is an input port, $(i, x, \#)$ is an input event; and where i is an output port, $(i, \#, y)$ is an output event. An *input-output trace* on a schema instantiation N_j is a sequence of input and output port events on the ports of that schema. ■

---

[5]From Kfoury et al. [1982].

Definition 44

If N is a k-variable schema of length n then a *strict computation* by an instantiation of N is a sequence, possibly infinite, of the form:

$$a_0 e_0 A_0 a_1 e_1 A_1 \ldots a_i e_i A_i \ldots$$

where the a_i terms are k dimensional state vectors, the A_i terms are the statements of N, and the e_i terms are port events (defined identically to port events for a port automaton) on the ports of the schema, all subject to the consistency constraints:

1. If a computation C of a schema instantiation N_j is restricted to just port events, then the resulting sequence must be an input-output trace of N_j.
2. If A_i is the nth statement of N, ST_n, then A_{i+1}, the next statement to A_i, is ST_1, the first statement of N. [6]
3. If A_i is a statement other than ST_n, then the next statement A_{i+1} is given by the individual semantics of A_i. ■

If we skip forward for a moment and imagine we have constructed the mapping from a schema instantiation to its port automaton semantics, and that the state of this port automaton is represented by the state vector of the schema instantiation, then, intuitively, a computation 'fills out' the input-output trace with 'internal' knowledge from the structure of a *particular* port automaton. Any two port events are punctuated by the internal state and schema instruction linking those two events. However, it is convenient to relax the constraint that port events punctuate every state; this simplifies describing the semantics of control instructions such as the `If` and `For`. The port connection automaton will provide us with the semantics of this 'abbreviation' of the computation.

Definition 45

An *abbreviated computation* is a computation sequence C in which more than one state-vector and instruction can appear between any two port events. ■

The justification for allowing such *pure state* transitions is as follows: Again, imagine we have constructed the mapping from a schema instantiation to its port automaton semantics P^N. We can consider this port automaton to be a port connection of some number of port automata. Internal communications within can cause state transitions in the component automata and hence in P^N, in the interval between external communications

[6]The exit targets of control statements, such as `If` and `For`, must be structured carefully to obey this.

(for example, see $\hat{\delta}^*$ in V.7). Note that the number of such internal communications is non-deterministic. Hence, between any two adjacent $a_i A_i$ terms in an abbreviated computation, there is a non-deterministic number of internal state transitions in the port connection automaton semantics.

Theorem 46

With each schema N defined according to Definition 41, we can associate a port automaton P which is the semantics of N. A particular *initial state* for P is selected from the set of initial states of P, $\tau_i \in \tau$, when a set of initial values for the variables of N are chosen. By selecting such an initial set of values to define initial state τ_i, we can construct a particular port automaton P_i which has τ_i as its singleton set of initial states. We refer to P_i as the semantics of the *schema instantiation*, ${}_iN$, of N with those particular initial values.

In order to prove this theorem, we construct a mapping from the components of N to those of a port automaton P, $P = (L_x, L_y, Q, X, Y, \tau, \beta)$ which is the semantics of N.

- The schema name N uniquely identifies a port automaton P^N as the semantics of N. For simplicty, we shall omit the superscript through the rest of this theorem, letting it be understood that P refers to P^N.
- The set of input ports L_x of P can be constructed as follows: For each name in the input port list of N, add an input port with this name to L_x. Let these be the only elements of L_x.
- The set of output ports L_y of P can be constructed as follows: For each name in the output port list of N, add an output port with this name to L_y. Let these be the only elements of L_y.
- We have set $X_i = \Re$ for all input ports and $Y_i = \Re$ for all output ports. For input ports $Y_i = \{\#\}$, and for output ports $X_i = \{\#\}$.
- The set of states Q of P is constructed from the state vector of N, and is the set product $\mathcal{N} \times \Re^k$, where $\Re$ is the set of real numbers and $\mathcal{N}$ is the set of natural numbers, and k is the number of internal variables in N:

  $$Q = \mathcal{N} \times \Re^k$$

  where the state vector of N provides the element of $\Re^k$, and the statement label in N provides the element of $\mathcal{N}$. For any N we shall construct the transition map δ of the port automaton which is its semantics by describing how each statement in N affects δ.

- The set of initial states of P is that subset of Q given by:

$$\tau \in 2^{\{1\} \times \Re^{k}}$$

That is, it denotes any assignment of initial variable values in which the next instruction to be executed is the *first* instruction.

- In order to construct the transition map, δ for P, we need to render *explicit* port reading and writing. We now describe a schema translation function Ξ, which takes as input some schema N, with port reading and writing implicit in the syntax of assignment, and produces a schema N' which has the same semantics as N but in which port reading and writing are explicit operations.

  Let $IP(V_i, j)$ denote the reading of a value from input port j, and its storage in variable V_i. Let $OP(j, V_i)$ denote the sending of the value stored in variable V_i to output port j. Whenever we encounter port operations in the program specification of N, we replace them in N' as follows:

$$\begin{array}{c|c} var_i := expression(inport_j) & oport_j := expression \\ \Downarrow & \Downarrow \\ IP(var_{k+1}, j) & var_{k+1} := expression \\ var_i := expression(var_{k+1}) & OP(j, var_{k+1}) \end{array}$$

  where, again, by $expression(x)$ we denote an arithmetic expression featuring x in certain specified places. We add an extra variable var_{k+1} to N' to safely allow this transformation, since using existing variables in N might alter the semantics of N. From now on, we shall assume that this translation function has been applied for schema N to yield $N' = \Xi(N)$. Let Q and τ be based on the augmented value of k for N' once these port operation transformations have all been made, and we shall drop the N' notation and just use N.

- δ is a mapping:

$$\delta : Q \times \bigsqcup_{i \in L_s \cup L_y} X_i \rightarrow 2^{Q}$$

- β is a collection of maps:

$$\beta_\ell, \ell \in L_y$$

We shall construct δ and β by:

1. For each statement ST_i (corresponding to some instruction A_j in a computation of N), we present how ST_j maps from one state a_j, and possibly, (in the abbreviated computation), port event e_j to the next state a_{j+1}.

2. For each ST_i corresponding to A_j, we show how the next statement in the computation A_{j+1} is chosen, given the consistency constraints of Definition 44.

For a particular N, the semantics P^N can be constructed from the *order* of the statements in the program section of N, and from the individual effect of each statement as presented below. We present the effect of each statement as a mapping of one of two types: The first type of mapping is based on the strict computation definition. We present the effect of the statement by writing a state vector and port event, followed by the statement being discussed, followed by a unique state vector which is the result of the statement execution.

$$\begin{array}{c}(i, v_1, \ldots, v_k)(\ell, x, y) \\ ST_i \\ (\bar{i}, \bar{v}_1, \ldots, \bar{v}_k)\end{array} \tag{V.25}$$

The second type of mapping is based on the definition of an abbreviated computation, and is essentially a pure state transition. We write a state vector, followed by the statement being discussed, followed by the unique state vector which is the result of the statement execution (bearing in mind our comments on pure state transitions).

$$\begin{array}{c}(i, v_1, \ldots, v_k) \\ ST_i \\ (\bar{i}, \bar{v}_1, \ldots, \bar{v}_k)\end{array} \tag{V.26}$$

These are to be interpreted as: If the next instruction in the computation of N is A_j, which is statement ST_i in N (i.e., it has statement label i), and the state vector of N, a_j is equal to $(v_1, \ldots, v_k)$, and the port event (if specified) $e_j = (\ell, x, y)$ (where one of x or y *must* equal #), then the next state vector in the computation of N will be $a_{j+1} = (\bar{v}_1, \ldots, \bar{v}_k)$ and the next instruction in the computation of N will be $A_{j+1} = ST_{\bar{i}}$.

If the mapping is specified as type one above (in terms of a strict computation) then we can construct δ and β as follows: If P^N is the semantics of N, then the current state of P^N is given by $q = (i, v_1, \ldots, v_k)$, the output map for ℓ in state q is $\beta_\ell(q) = y$, and the next state of P^N is given by the transition function:

$$\delta(q, (x, \ell)) = \{(\bar{i}, \bar{v}_1, \ldots, \bar{v}_k)\} \tag{V.27}$$

However, if the mapping is of type two above, then we do not have enough information to construct directly δ and β. By considering P^N, the semantics of N, to be a port connection of a number of port automata, we can say that the pure state transition of P^N is caused by internal

communications on the component automata of P^N. And, although there may be a undetermined amount of such communication, eventually P^N will be in the specified final state.

We shall use the notation $next(i) = (i \ mod \ n) + 1$, where n is the length of N; this can be seen as embodying the second consistency condition in Defintion 44.

Input Operation

The semantics of the explicit input operation is given by the mapping:

$$\begin{array}{c}(i, v_1, \ldots, v_j, \ldots, v_k)(\ell, x, \#) \\ IP(var_j, \ell) \\ (next(i), v_1, \ldots, x, \ldots, v_k)\end{array}$$

Output Operation

The semantics of the explicit output operation is given by the mapping:

$$\begin{array}{c}(i, v_1, \ldots, v_j, \ldots, v_k)(\ell, \#, v_j) \\ OP(\ell, var_j) \\ (next(i), v_1, \ldots, v_j, \ldots, v_k)\end{array}$$

Assignment Operation

The semantics of the assignment statement with port operations extracted is:

$$\begin{array}{c}(i, v_1, \ldots, v_j, \ldots, v_k) \\ var_j := expression \\ (next(i), v_1, \ldots, expression, \ldots, v_k)\end{array}$$

Conditional Operation

For convenience we restate the syntax of the conditional statement:

If (*Booleanop*) **Then** *Body*1 **Endif** (V.28)

If (*Booleanop*) **Then** *Body*1 **Else** *Body*2 **Endif** (V.29)

Note that any port operations in *Booleanop* will have been removed by Ξ. Let i_{s1} be the label of the first statement in *Body*1, let i_{s2} be the first statement in *Body*2. Let i_{ef} be the label of the **Endif** clause, and let i_{el} be the label of the **Else** clause if one exists. If *Body*1 is empty and there is an **Else**, then $i_{s1} = i_{el}$, otherwise $i_{s1} = i_{ef}$; if there is an **Else** clause and if i_{s2} is empty, then $i_{s2} = i_{ef}$. We present the semantics of the conditional as three maps: One for the **If** clause, one for the **Else** clause and one for the **Endif** clause (which is the exit target of the statement). These three together in order give the semantics of V.29 above, omitting the **Else** semantics and setting $i_{s2} = i_{ef}$ gives the correct semantics of the V.28 form above.

Let us define the function $C : \{b : b \text{ is a Booleanop}\} \rightarrow \{0,1\}$ as:

$$C(b) = \begin{cases} 1 & if\ b\ is\ true \\ 0 & else \end{cases} \tag{V.30}$$

IF

Assume b is a *Booleanop*, then

$$\begin{array}{c} (i, v_1, \ldots, v_k) \\ \textbf{If } b \textbf{ Then} \\ (C(b) * i_{s1} + (1 - C(b)) * i_{s2}, v_1, \ldots, v_k) \end{array} \tag{V.31}$$

ELSE

Let $(v'_1, \ldots, v'_k)$ be the state vector after executing *Body*1, the semantics of the **Else** clause is just a jump to the exit:

$$\begin{array}{c} (i_{el}, v'_1, \ldots, v'_k) \\ \textbf{Else} \\ (i_{ef}, v'_1, \ldots, v'_k) \end{array} \tag{V.32}$$

ENDIF

Let $(v''_1, \ldots, v''_k)$ be the state vector after executing *Body*1 or *Body*2, it does not matter which since **Else**, like **Endif**, does not effect the internal variables. The semantics of **Endif** is:

$$\begin{array}{c} (i_{ef}, v''_1, \ldots, v''_k) \\ \textbf{Endif} \\ (next(i_{ef}), v''_1, \ldots, v''_k) \end{array} \tag{V.33}$$

Iteration

For convenience we restate the syntax of the iterative statement:

$$\textbf{For} \quad var_j := LB \ldots UB \quad Body \quad \textbf{Endfor} \tag{V.34}$$

Let i_s be the label of the first statement in *Body*; if *Body* is empty, let $i_s = i_{ef}$, where i_{ef} is the label of the **Endfor** statement, the exit target of the **For** statement. Again, notice that any port operation will have been taken from LB and UB by Ξ. Let us define the function $SP : Z \times Z \rightarrow \{+1, -1\}$, where Z denotes the set of integers, as:

$$SP(LB, UB) = \begin{cases} +1 & if\ UB \geq LB \\ -1 & else \end{cases} \tag{V.35}$$

We present the semantics of the iterative statement in two parts: One for the **For** clause and one for the **Endfor** clause.

FOR

$$\begin{array}{l}(i_{for}, v_1, \ldots, v_j, \ldots, v_k)\\ \textbf{For}\ var_j := LB \ldots UB\\ (i_s, v_1, \ldots, LB, \ldots, v_k)\end{array} \quad \text{(V.36)}$$

ENDFOR

After executing *Body* one or more times, let the state vector be $(v'_1, \ldots, v'_j, \ldots, v'_k)$, where

$$(LB \le v'_j \le UB \& SP(LB,UB) = +1)\ OR\ (UB \le v'_j \le LB \& SP(LB,UB) = -1) \quad \text{(V.37)}$$

Also, for clarity, let:

$$\begin{array}{lcl} c &=& c(\text{"}v_j = UB\text{"})\\ \tilde{c} &=& 1 - c\\ i_e &=& next(i_{ef})\\ sp &=& SP(LB,UB)\end{array} \quad \text{(V.38)}$$

The semantics are then:

$$\begin{array}{c}(i_{ef}, v'_1, \ldots, v'_j, \ldots, v'_k)\\ \textbf{Endfor}\\ ((\tilde{c} * i_s + c * i_e), v'_1, \ldots, (v'_j + sp * \tilde{c}), \ldots, v'_k)\end{array} \quad \text{(V.39)}$$

Finally, we have described the mapping from N to its semantics, the port automaton P^N. Note that $\tau \in 2^{\{1\} \times \Re^k}$; for each $\tau_i \in \{1\} \times \Re^k$, the computation done by N for that set of initial variable values will be different and unique. The semantics of the schema instantiation (or SI) ${}_iN$ of schema N is the port automaton P_i^N which is the the port automaton P^N with its τ replaced by a singleton set $\tau = \{\tau_i\}$. Any computation by P_i^N is constrained, therefore, to begin with a_0 constrained by τ_i, thus fixing a_0 for a given computation of N. The port automaton ${}_iN$ describes a computing agent constructed from schema N; that is, an SI N_j for some j. In the next definition we shall discuss the connection between i and j for schema N. ∎

We augment the syntax of schema program specification to include the *instantiation* and *deinstantiation* operations. The instantiation operation takes as parameters a schema name and a set of internal variable values, and makes a corresponding schema instantiation. The SIs are the computing agents in our model. A schema has no unique starting state, so it describes many possible computations; an SI has a unique starting state.

An *instantiation number* (I#) is a natural number which uniquely identifies a particular SI. The I# is part of the implementation details of the model, and thus does not enter directly into our semantics. However, the fact remains that the I# is a useful tool for describing generic networks; so we shall allow I# as a restricted optional parameter to the instantiation operation, and also as part of of the SI name. We constrain these uses of the I# to allow only generic network specification.

I# can be passed as a parameter to the instantiation operation — in which case, it is an error to have more than one SI with the same number, or if omitted, it is understood that a unique I# is automatically assigned to the SI. We indicate the I# of an SI, when it is necessary to do so, by subscript: N_j is the SI of schema N with an I# of j. In Theorem 46, we demonstrate that the semantics of an SI is a port automaton P_i^N with a singleton set of initial states $\{r_i\}$ corresponding to the initial variable values of the SI. Just as N is an arbitrary name for a schema, N_j is an arbitrary name for a computing agent created from N. There is no connection between i and j; and the value j is not part of the semantics of the computing agent. Since N_j can describe any computing $_iN$ (note that i is very much a part of the semantics), we are free to construct whatever useful meaning we see fit for j, and we shall do just this in this definition.

A network connection map, connecting the ports of N_i to ports on other SI can be specified as a parameter to the instantiation operation. The rationale for including this parameter is that, in general, a computation is only useful if it interfaces, somehow, with the environment (other SIs, physical devices, etc.). The instantiation operation can only create a useful SI if it allows for specifying how the new SI is to be joined to other existing SIs.

Definition 47

The instantiation operation has the following syntax:

$$\begin{array}{lll}
\langle Inst \rangle & ::= & \langle schema - name \rangle_{\langle inst \rangle}(\langle CI \rangle)(\langle CO \rangle)(\langle V \rangle) \\
\langle CI \rangle, \langle CO \rangle & ::= & \langle connection - list \rangle \\
\langle connection - list \rangle & ::= & \wedge \mid \langle item \rangle \langle connection - list \rangle \\
\langle item \rangle & ::= & \langle connection \rangle \mid \langle connection \rangle \parallel \langle item \rangle \\
 & & \mid \langle connection \rangle \parallel^{+} \langle item \rangle \\
\langle connection \rangle & ::= & \langle local - port - name \rangle \mid \langle local - SIPname \rangle \mid \langle equiv - port \rangle \\
\langle local - port - name \rangle & ::= & a\ port\ of\ the\ creating\ SI \\
\langle local - SIPname \rangle & ::= & \langle local - sname \rangle_{\langle l-inst \rangle}(port) \\
\langle equiv - port \rangle & ::= & \equiv \langle local - port - name \rangle \\
\langle local - sname \rangle & ::= & local\ schema\ name
\end{array}$$

- *schema – name* is the name of a schema which has been defined using the syntax of Definition 41.
- *inst* is the I# of the new SI, and may be omitted, in which case the I# of the new SI is set to the highest I# of this schema, so far, plus one; it is set to one if no prior instantiations of this schema exist. An attempt to create an SI with the same schema name and I# as some prior SI in the current assemblage is an error, and no SI is created.
- *l – inst* is an instantiation number for a schema instantiation in one of the connection lists. If it is ommitted, then we employ certain *default* addressing rules to remove the ambiguity in the SI reference (see below).
- v is a list of *expressions* assigned by position to the internal variables of the new SI as declared in its schema definition.
- CI is a list of port names, denoting port to port connections by position, between the ports in the list and the input ports of the new SI as declared in its schema definition.
- CO is a list of port names, denoting port to port connections by position, between the ports in the list and the output ports of the new SI as declared in its schema definition.
- We postpone discussion of || until Definitions 49 and 50; and of ≡ until Definition 59.

Default Instantiation Numbering Rules: If the instantiation number is not provided in the $SIPname$ in a connection list, we can deduce an appropriate number as follows:

1. If the schema name is the same as the schema being created, assume that the instantiation number is that of the new SI.
2. If the instantiation instruction is in the body of a `Forall` loop of which the index schema name is the same as the schema name in the unresolved SI, then get the instantiation number from the instantiation number of the index schema.
3. If none of the above hold, then assume instantiation number one (e.g., when there only ever will be one instantiation of a particular schema in a given assemblage).

The formal semantics of instantiation can be viewed as follows: Consider a network of SIs, whose semantics is defined by a network automaton as in Definition 28. If a new SI is now added to this network by one of its component SIs executing an instantiation command, then the semantics of the network (the structure of the network automaton) changes. Let P^N be the network automaton which is the semantics of the network of SIs *before* the instantiation operation. Let P^j be the semantics of the newly created SI, with unique starting state r_0^j. Let c be the port

connection mapping specified in the instantiation command by CI and CO between ports on P^j and those on P^N. The semantics of the new network is the network automaton P^M, which is the *port connection* of P^N with P^j under c:

$$P^M = P^N \|_c P^j \tag{V.40}$$

Definition 48

The deinstantiation operation has the following syntax:

Stop

The execution of this operation results in the elimination of the executing SI as a computing agent. The operation is understood to be applied to the SI which executes it. The Deinstantiation operation is the *only* way to terminate a computation; the last instruction in the computation must be the deinstantiation command.

We can view the formal semantics as: Consider a network of SIs whose semantics is the network automaton as defined in Definition 28. If some SI in this network deinstantiates, then the semantics of the network (the structure of the network automaton) changes. Let P^N be the network automaton which is the semantics of the network before the SI deinstantiates; let P^j be the semantics of the SI which deinstantiates. Once P^j ceases to exist, then any connections which had been made to P^j are now undefined! In order to capture the correct semantics, we dismantle P^N back into its components and built up a new network automaton P^M in which P^j has been replaced by a 'dummy' P^{*j} which has the same ports as P^j (hence any connections to P^j are still defined). Additionally, this method allows us to specify through P^{*j} how we choose to handle SIs which receive port communication 'after' they have deinstantiated.

Let the semantics of each of the component SIs of the network N be the port automata P^i, $i = 1, \ldots, m$, and let P^j, $j \in \{1, \ldots, m\}$ be the semantics of the SI which deinstantiates, and assume the set of port connection maps $\{c_k : k = 1, \ldots, m-1\}$ as discussed in Definition 28. The semantics of the network before P^j is deinstanjated is $P^N = P^{\tilde{m}-1}$ as given by:

$$\begin{aligned} \tilde{P^0} &= P^1 \\ \tilde{P^k} &= P^{\tilde{k}-1} \|_{c_k} P^{k+1} \quad k = 1, \ldots, m-1 \end{aligned} \tag{V.41}$$

When the deinstantiation command is executed we replace P^j by some automaton P^{*j} which has the same input and output ports as P^j. The semantics of the network after the deinstantiation

command is the network automaton $P^M = \tilde{P^{m-1}}$, where:

$$\begin{array}{lllll} \tilde{P^0} & = & P^1 & P^j = P^{*j} \\ \tilde{P^k} & = & \tilde{P^{k-1}} \|_{c_k} P^{k+1} & k = 1, \ldots, m-1 \end{array} \quad \text{(V.42)}$$

The behavior of P^{*j} determines how 'dangling connections' are handled. Two possible behaviors for P^{*j} are:

1. The Black Hole. The transition and output maps for P^{*j} are both the null map. Thus communication with an SI, after it has deinstantiated, will never return any replies; and unless some form of time-out is used (which we will discuss later) a communicating process is permanently blocked (perhaps causing deadlock). This is the simplest form of P^{*j}.

2. The Taped Message. The output maps for P^{*j} all produce a unique error symbol for all ports for all states, and in all states all ports are active:

$$\begin{array}{ll} \forall q \in Q,\ \forall i \in L_x,\ \forall x \in X_i, & \delta(q,(x,i)) \neq \emptyset \\ \forall q \in Q,\ \forall j \in L_y, & \beta_j(q) = \text{"}ERROR\text{"} \end{array} \quad \text{(V.43)}$$

   Any communication to P^{*j} is answered immediately by a unique error message. A communicating process can choose to watch out for this error message or not. ∎

In the next few definitions, we shall address the meaning of fan-in and fan-out in network connection mappings. Port automaton communication is the semantics of all SI communication. For single port to port connections this is trivial, completely defined by Definitions 26 and 27. With branching connection maps (either fan-in or fan-out) it is possible to attribute either AND- or OR-semantics to fan-in and fan-out cases: If the read or write operation on the port with fan-in or fan-out does not complete until an individual read or write with *each* branch completes, this is called AND-semantics; if a read or write need only occur with *one* branch, this is called OR-semantics. We describe both semantics for each case. We identify one as the *basic* semantics for both fan-in and fan-out, by which we mean the semantics which it will have if it is not otherwise indicated – this is necessary, since fan-in and fan-out can occur implicitly, and we must ensure that we define what semantics these implicit cases will have.

Definition 49

When an output port in some SI is connected to more than one input port on other SIs, then this condition is called *fan-out*, and the number of input ports is called the *degree* of fan-out.

In the instantiation operation fan-out is denoted by *two ports names in the same position* in the CO list with || or $||^+$ placed between them. Fan-out can also occur implicitly with multiple instantiation commands.

Consider an SI, N, which has output port a fan-out connected to ports on n other SIs. The port automaton P^N which is the semantics of N has n output ports instead of output port a, named uniquely $a1, \ldots, an$. Whenever N outputs a value on a, P^N outputs the same value on all n of its ports $a1, \ldots, an$.

AND-semantics: A write to an output port with fan-out of degree n cannot terminate until all the n input ports have been read, and n simultaneous exchanges of y for the trivial value # must occur. That is, a write to port a by the SI N has the semantics of n successive writes by the port automaton P^N, in some undefined order, to its ports $a1, \ldots, an$. AND-semantics in fan-out fulfills part of the intuitive notion of a broadcast, i.e., that all intended recipients get the message.

OR-semantics: Definition 25, Note 2, defines when a port is *active*: Informally, it says a port is active if any input on that port will cause a defined state transition. Steenstrup et al. place no limit on the number of ports which can be simultaneously active in a given state. However, the first port to engage in communication[7] is the one which dictates the next state of the automaton. They note that it is not possible to *predetermine* which port will actually engage in communcation; for this we must consider the environment to which the ports are connected. A write to port a of N is equivalent to a write to each of the ports $a1, \ldots, an$, and they all become *simultaneously active*. Thus, a communication on any one of these ports will terminate the write. ■

Definition 50

Fan-In occurs when an input port on some SI is connected to more than one output port on other SIs. The number of output ports is called the *degree* of the fan-in. Fan-in is denoted in the instantiation operation by *two ports in the same position* on the CI list, with || or $||^+$ between them. Fan-in can also occur implicitly with multiple instantiation commands.

Consider an SI, N, with semantics P^N: If N has input port a which has fan-in of degree n, then P^N has n input ports unambiguously named $a1, \ldots, an$. A read to a by N has the semantics of a read to each of $a1, \ldots, an$.

---

[7]Steenstrup et al. limit to one, without loss of generality, the number of ports which can engage in communication at the same time.

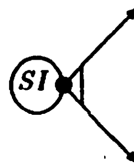
AND-semantics: A read to a will not terminate until n reads have occurred, one from *each* of the ports to which a is connected. The value returned in a is the value of the last read to occur. It is clear that AND-semantics in fan-in are useful mainly for synchronization purposes, since only one of the n data can actually be held in a at the termination of the read.

OR-semantics: Using the same reasoning as in fan-out, we define OR-semantics for fan-in as follows: A read by N to a has semantics of making the n ports of P^N *simultaneously* active. Thus, if a read operation occurs on *any one* of the n ports, then the read to a will terminate. ■

Notice that OR-semantics is quite useful in both fan-in and fan-out, but AND-semantics is not that useful in fan-in. Also notice that AND-semantics can be recreated by removing the fan-in/out and simply adding $n-1$ extra ports to the schema, since it is in exactly these terms that AND-semantics is defined. The only advantage to keeping a notation for AND-semantics is for brevity. We shall define OR-semantics as the basic semantics of both fan-in and fan-out, and will associate AND-semantics with them only when explicitly indicated.

Definition 51

OR-semantics is the basic semantics of fan-in and fan-out in a network connection map. If the fan-in/out occurs implicitly, then it has this basic semantics. If it arises from an explicit network connection map using the $\|$ operator, then it also has this basic semantics. It only has AND-semantics when this is explicitly indicated in the connection mapping by the symbol $\|^+$. In diagrams of SI networks, we shall indicate AND-semantics in the standard way, with a 'tie' between all the branches of the fan-in/out, e.g.:



■

Note that the non-determinism in fan-in/out, when more than one branch of the fan-in/out is connected to an active port (i.e., a port which can participate in communication) is (by the above semantics) a particular case of the non-determinism discussed with respect to the input-output trace definition.

Definition 52

In this definition we describe the *parallel search problem of synchronous communication*. The problem is a result of the *blocking* effect of synchronous communication, Definition 30.

To reiterate briefly, synchronous communication involves the synchronizing of sender and receiver at some specific point in their code in order for data transfer to occur. If one should reach the synchronization point first, it 'waits', blocks, until the other also reaches its synchronization point. Only when they are thus synchronized will communication occur (hence the name). In asynchronous communication, messages are passed between sender and receiver without any such explicit synchronization. The advantages of synchronous over asynchronous communication are:

1. When a sending agent finishes its transmission, it is guaranteed that the receiver has the message (otherwise the sender would still be blocked).
2. No queue handling is necessary.
3. It can be modeled nicely by a transition function.

On the other hand, a disadvantage of synchronous communication is that the blocking of computing agents while they communicate can exacerbate deadlock and also cut down on the parallelism between computing agents (since they spend time waiting for each other). It is this latter point that we shall call the *parallel search problem.*

Consider an SI, A_k, whose function is to search a range of n elements, $R = r_1 \ldots r_n$, for an element r_i that fulfills some criterion C, i.e., to find r_i in R, such that $C(r_i)$. To exploit the speedup available in parallel processing, this SI creates two other *search* SIs, each of which will search one half of the range R. Let these be B_i which searches $r_1 \ldots r_{\lceil \frac{n}{2} \rceil}$ and D_j which searches $r_{\lfloor \frac{n}{2} \rfloor} \ldots r_n$. Since both of these operate in parallel, there exists the potential for them to find an r satisfying $C(r)$ faster than a single SI sequentially searching through R.

Let B_i and D_j behave as follows: Whenever they find a value of r_i, such that $C(r_i)$, they immediately terminate their search and communicate this value back to A_k. If they complete their search and do not find an appropriate value, then they return a unique *failure* value. It is *how* this communication must be done which embodies the parallel search problem. Consider that the parent SI has an input port a connected to output port a' of B_i, and an input port b connected to output port b' of D_j. When either search SI has found a result, they write the value to their respective output port. The parent SI must first read one of the ports a, say, and then read the other, b, to determine how the search has completed.

Let A_k have semantics P^A, B_i semantics P^B and D_j semantics P^D. The network map is:

$$\begin{array}{lcl} c: \coprod_{\ell \in \{A,B,D\}} S^\ell & \rightarrow & \coprod_{\ell \in \{A,B,D\}} S^\ell \\ c(A,a) & = & (B,a') \\ c(A,b) & = & (D,b') \end{array} \tag{V.44}$$

After the read to a, let P^A be in state q, and

$$\beta_b^A(q) = \# \quad \delta^A(q,(x,a)) \neq \emptyset, \quad x \in X_a^A(= \Re) \tag{V.45}$$

Unless P^B now, or subsequently, reaches a state $\tilde{q}$ such that:

$$\beta_{a'}^B(\tilde{q}) = y \quad \delta^B(\tilde{q},(\#,a')) \neq \emptyset, \quad y \in Y_{a'}^B(= \Re) \tag{V.46}$$

then P^A cannot undergo any further state transitions; it is 'blocked'. Even if P^D is ready to communciate, P^A cannot do anything until P^B reaches an appropriate state (if it ever does).

The parallel search problem is, therefore, that an SI such as A_k:

1. May never terminate if any *one* of its search SIs fail to return a result.

2. May have to wait for *both* SIs to terminate before it can get a result. ∎

Theorem 53

A network of a parent SI and two search SIs can be constucted, using port fan-in, which does not have the parallel search problem as defined above.

Let us assume the construction in the above definition: Parent A_k with semantics P^A; search SIs B_i and D_j with semantics P^B and P^D respectively. But, let us use the following connection mapping:

$$\begin{array}{rcl} P_j(a) & \leftarrow & S_i(a') \\ P_j(a) & \leftarrow & S_j(b') \end{array}$$

where the parent SI now *just* reads port a to get its result. Since we have not indicated otherwise, the fan-in has OR-semantics. However, now the semantics of A_k, P^A, has two ports $a1, a2$, connected:

$$\begin{array}{rcl} c(A,a1) & = & (B,a') \\ c(A,a2) & = & (D,b') \end{array} \tag{V.47}$$

and a read to a by A_k is equivalent to activating *both* ports in P^A:

$$\begin{array}{l} \beta_{a1}^A(q) = \# \quad \delta^A(q,(x,a1)) \neq \emptyset, \quad x \in X_{a1}^A(= \Re) \\ \beta_{a2}^A(q) = \# \quad \delta^A(q,(x,a2)) \neq \emptyset, \quad x \in X_{a2}^A(= \Re) \end{array} \tag{V.48}$$

It follows directly from the definition of communication, Definition 27, that now *either* branch of the fan-in will cause a state transition. Thus P^A is *not blocked* as long as one of P^B, P^D can communicate. Thus we have that any A_k:

1. Will only fail to terminate if *both* SI fail to return a result (and we attack this problem in the next theorem).
2. Has only to wait for the fastest search SI to terminate before it gets a result.

Notes

1. In the case that the search SI which terminates first produces the failure signal, then another read to a is necessary. This second read to a will return the result from the second SI (which, remember, was searching in parallel with the first). If communication overhead (i.e., the time to execute read port instructions etc.) is discounted, then our fan-in semantics produces the best possible result in this case also, which is the time for the longest (or next longest, if more than two search SIs are involved) SI to terminate.
2. In the case where both SIs terminate simultaneously, then since our semantics define that only one port communication can occur at a time (Definition 27), we let these simultaneous communications be ordered arbitrarily. Therefore either we find r, or Note 1 above applies. ∎

Theorem 54

A connection can be constructed between an input port on one SI and an output port on another SI, which has the property that a read to the input port will *always* terminate, even if the output port is never written to. [8]

Let there be two SIs, A_i and B_j, with semantics P^A and P^B, respectively. Let output port a of A_i be connected to input port b of B_j. A read to a will now leave P^A blocked as discussed in Definition 52. We shall use the same construction as in the previous Theorem 53. Let us introduce an SI, T_k with semantics P^T, with output port c, which is *fan-in connected* to b on B. Recalling how the semantics of fan-in require P^B to now have two ports $b1$ and $b2$, the network connection map:

$$\begin{array}{lcl} c: \bigsqcup_{\ell \in \{A,B,T\}} S^\ell & \rightarrow & \bigsqcup_{\ell \in \{A,B,T\}} S^\ell \\ c(A,a) & = & (B,b1) \\ c(T,c) & = & (B,b2) \end{array} \qquad \text{(V.49)}$$

Since we have not indicated otherwise, fan-in has its basic OR-semantics. We now simply construct T so that it is *guaranteed* to unblock B. If we further constrain T to transmit only a special error

[8] Specifically, this is the form of time-limit on reception discussed in Streenstrup et al. [1983, p46, Note 3]. However, they assume it without construction; here we construct it from basics.

symbol on c, then B can recognize the fact that the message received was from T and not from A.

It is important for T to give A a 'fair shot' (fairness) at getting its message through. Yet, without a concept of global time, it is not possible to relate the 'rate' at which T 'does its processing' to that of A (or any other SI either). Hence, we can only give the notion of fairness a local meaning – we can insist that T counts up to some value (i.e., its semantics cycles through some chain of states) before it tranmits its unique *time-out* value.

Let us rename this SI `Time$out`; it has the following behavior: `Time$out` is created with its output port `t$out` fan-in connected to the output port upon which the time limit is being set. It has one internal variable `frequency`, which is set to some value upon creation of the *SI*. `Time$out` will cycle `frequency` (hence a local timing) times around its program section, and then write the trivial value to its output port.

```
[  Time$out
   Input-Port-List:     ( )
   Output-Port-List:    ( t$out:Real )
   Variables:           ( frequency:Integer )
   Behavior:            ( frequency:=frequency-1;
                         If (frequency≤ 0) Then
                          t$out:=#;
                          Stop;
                         Endif;    )             ]
```

Thus, if no message is received by the port to which this SI has been fan-in connected by `frequency` counts, then the trivial response is guaranteed to arrive. Let P_i be some SI with input port a. Before P_i read a value from a, it creates an instance of `Time$out`, fan-in connected to a with the statement:

`Time$out()(a)(n)`

where n is some natural number which effects how long P_i is willing to wait for a value to arrive on a.

The ability to set a time limit on how long a receiving process waits for input is a characteristic of asynchronous communication. Hence, the fan-in semantics of Definition 50 allow the emulation of a form of asynchronous communication. As Steenstrup et al. note, this time limit ability also improves deadlock behavior.

Note: While this mechanism of time out is fine for the *analysis* of schemas, it is too involved for the *synthesis* of schemas. Having demonstrated that we can indeed generate time out SIs from our basic tools, the most sensible thing to do now is to adopt it as an additional basic operation. ■

Observation 55

Using the synchronous communication operations and the instantiation operation, it is possible to duplicate completely an asynchronous communication operation.

The defining property of asynchronous communication is that the sender does not wait (its semantics do not block) for the receiver to complete its side of the operation.

This is quite straightforward: If some SI, S_j, wants to send a message synchronously to some other SI, N_i, then S_j simply creates another SI, M_k, with its port(s) connected to the port(s) of N_i over which the message is to be received, and passes on the message to M_k via its initial variable values. M_k then synchronously transmits the message, while S_j continues uninterrupted after carrying out the instantiation operation. This is exactly the defining property of asynchronous transmission.

We augment the program specification of Definition 41 to include the **Forall** statement. This statement carries out some action across all instantiations of some schema, regardless of how many (if any) there are.

Definition 56

The following syntax describes the **Forall**:

$$
\begin{array}{lcl}
\mathit{Forall} & ::= & \mathbf{Forall}\ S_i : \\
 & & \langle \mathit{Instlist} \rangle\ \mathbf{Endforall} \\
\mathit{Instlist} & ::= & \Lambda \mid \langle \mathit{Inst} \rangle \langle \mathit{Instlist} \rangle
\end{array}
$$

where,

- *Inst* is a schema instantiation statement.
- We refer to *Instlist* as the *body* of the **Forall** Loop.
- We refer to S, which is a schema name, as the *index schema* of the **Forall** loop.
- i is a local variable, and, within the **Forall** loop, it holds the instantiation number of the instance of S being considered.

The **Forall** loop executes its body *once* for each instantiation of its index schema which currently exists. All executions of the loop-body occur simultaneously, not serially like a **For** statement. If the index schema name occurs in the body of the **Forall** loop, then it is understood to refer to the instantiation of the index schema for which the current execution of the loop-body occurs (since it occurs once for each instantiation). Additionally, if the symbol i occurs in the loop-body, then it is understood to refer to the instantiation number of the index schema for which the current execution of the loop-body occurs.

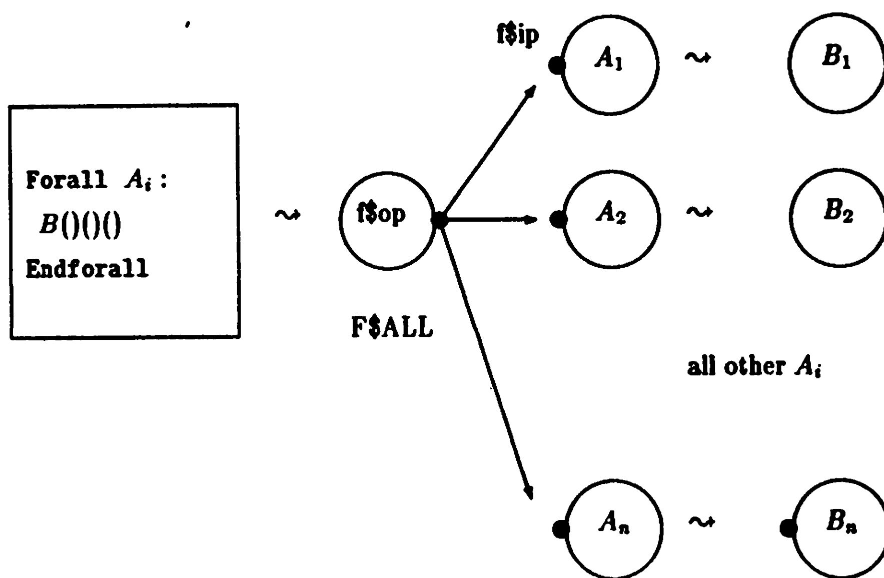


Figure 21: The Semantics of the Forall Statement.

The formal semantics of the Forall statement is provided by the port automaton model. Without loss of generality, let us consider the body of the Forall loop to be a single *Inst*. We use the SI created in *Inst* to describe the action we wish to perform on each instance of the index schema. Let us associate an index $I(S)$ with every schema S, and $S(I(S)) = S$. Let the port automaton which is the semantics of a schema be augmented as follows:

1. Let it always possess an input port unambiguously named *f$ip*.
2. Let it always have the following code: *f$ip* is read, if a # value is passed to it, nothing is done; but if a non-trivial value is passed, then that value, say i, is used to make a schema instantiation of schema S, such that $i = I(S)$.

When some SI, A, with semantics P^A executes a Forall statement, this is equivalent to A executing an *instantiation* of the special schema *F$ALL*, and to setting an initial variable value of this SI to the value of the schema which is mentioned in *Inst* (let this be B).

F$ALL is a special predefined schema with one output port *f$op*; when F$ALL is created by the execution of a Forall operation, it is fan-out connected to all instances of the *index schema*'s (let this be A) *f$ip* (default) port. The semantics of F$ALL is therefore a port automaton $P^{F\$ALL}$, which has one output port for each instantiation of A which exists. F$ALL writes the index of the name B, $I(B)$, to all its output ports. Thus each instance of A will receive (by 2. above of our augmented semantics) this index, i say, on its *f$ip* port. Its action is then to create an instance of the schema $S(i)$, the schema specified in the *Inst* statement of the original Forall (Figure 21). ■

Observation 57

The Forall statement provides for a daunting amount of parallelism; it is limited only by the assemblage scoping rules. A more 'sequential' form of the Forall can easily be constructed. This sequential form behaves as follows: It executes its body once for each instantiation of its index schema which currently exists. The body can only consist of instantiation statements, as we have discussed. For simplicity, let us assume there is only one instantiation command in the body of the Forall, and, again, let us assume it is an instantiation of the schema B, and that the index schema of the loop is A. So far, we have discussed nothing new.

The sequential version of the Forall specifies that each instantation of B created is connected to the previous instantiation (or more clearly, the instantiation with the next lowest instantiation number. This connection is from a default port called the *closedown* port on an instantiation of B to a default port called the *startup* port on the instantiation of B with the next highest instantiation number. We now constrain the first action of an instantiation B to be a read of the

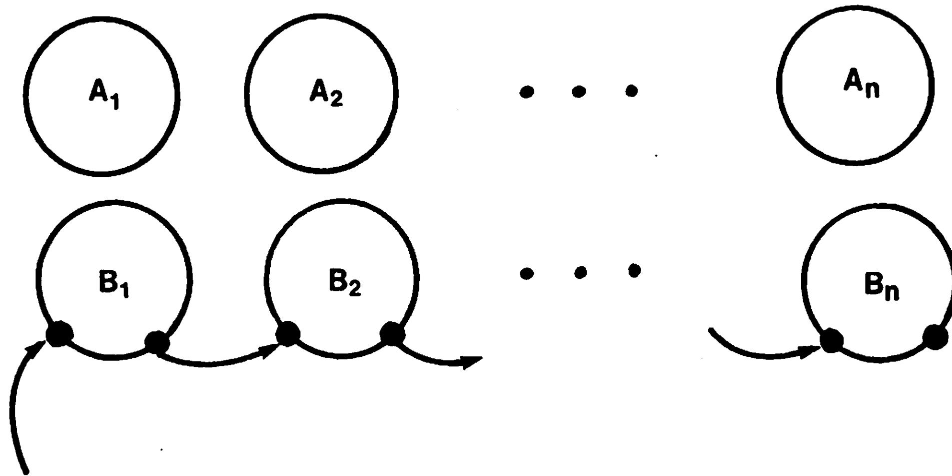


Figure 22: The Sequential Version of the Forall Operation.

null value from its *startup* port, and the very last action of any instantiation of B to be a write of the null value to its *closedown* port. We shall avail of these default ports once again in Chapter 6, to define a temporal logic semantics of instantiation. ■

Notice now that the `forall` has created a systolic structure of SIs. Any particular instantiation of B cannot begin processing until all instantiation with lower instantiation numbers have deinstantiated. In order to solve the bootstrap problem, that is, to start off the instantiation of B with the lowest instantiation number, we must ensure that the SI executing the `Forall` writes a null value to the *startup* port of this instantiation. This sequential version of `Forall` has the benifit of limiting the parallelism of the general `Forall`.

Observation 58

It has already been noted in Chapter 4 that we can construct a 'super' `Forall` statement which has an arbitrary loop-body, rather than being restricted to having simply instantiation statements in the loop-body.

Given any arbitrary sequence of statements, which obey the syntax definition of behavior as we have defined it, as the body of a `Forall` loop, we construct a loop-body schema description from those statements as follows: Any local variables mentioned in the loop-body, and not mentioned elsewhere in the schema containing the `Forall` (which we will call the main schema), are considered unique local variables of the loop-body schema. Any local variables mentioned in the

loop-body which are mentioned elsewhere in the main schema are considered local variables of the loop-body schema which are parameterized on instantiation of the loop-body schema by the corresponding variables of the main schema. Any port names of the main schema mentioned in the loop-body are considered port names of the loop-body schema; and on instantiation of the loop-body schema these ports are connected to the corresponding ports on the main schema. Any reference to the ports of the index schema are considered references to the ports of the loop-body schema; and, on instantiation of the loop-body schema, these ports are connected to the corresponding ports of the index schema.

Having constructed the loop-body schema, the semantics of a Forall statement with arbitrary body is equivalent to the semantics of a Forall loop which has the instantiation of the loop-body schema (derived and instantiated as per the above instructions) as its body. We can additionally define a 'super' form of the sequential Forall statement by constraining the loop-body schema to use *startup* and *closedown* as described in Definition 57. ■

An *Assemblage* is a computing agent whose behavior is defined in terms of the behavior of a number of other computing agents, which can, in turn, be assemblages or schema instantiations. This definition exploits the fact that, for every network of port automata, there exists a single network automaton whose behavior is equivalent to that of the network. It uses this fact to allow the specification of a schema as the behavior of a network of computing agents. The motivation behind this is that some programs can more easily be described as a network of interacting computing agents. Internally, the assemblage is described by a set of SIs, each with its own semantics, and a network connection mapping. Externally, the assemblage has as its semantics the single network automaton composed of the individual semantics of all the component SIs of the assemblage.

Definition 59

An assemblage schema description of Chapter 4, Definition 15, can be abbreviated by omission of keywords (as we did for the schema definition) :

$$[\, N\ (IP)\ (OP)\ (V)\ (S)\ (IB)\,] \qquad \text{(V.50)}$$

where,

- N is a name uniquely identifying this assemblage schema,
- IP is a list of input port names and their types,
- OP is a list of output port names and their types,
- V is a list of internal variable names and their types,

- S is a list of the component schemas in this assemblage; we shall say they are *local* to this assemblage.
- IB is a behavior section which creates and connects the SIs which compose this assemblage. From IB we can deduce two maps: C, a network connection map for the assemblage; and P, an equivalence map for component SI ports and assemblage ports. We use the prefix $\equiv$ followed by an assemblage port name in the CI and CO list of instantiation commands in IB to describe the map P. It has similar semantics to port connection in CI and CO; but instead of denoting a connection between the named port and the port in that position in the schema definition of the created SI, it denotes that the named assemblage port is *equivalent* to the port on the SI.

Notes:

1. An instance of an assemblage schema is denoted N_i, as for a schema. Assemblage instantiation is identical to that of a schema. Once the assemblage is instantiated, its IB component immediately starts executing to set up the assemblage network.
2. Deinstantiation of N causes immediate deinstantiation of all the component SIs.
3. Deinstantiation of all the component SIs of N results in the deinstantiation of N.
4. We define the *Assemblage Locality Property of Instantiation Numbers* to state that the instantiation number of any SI is *local* to the assemblage in which that SI was instantiated (since, in Definition 47, we established our freedom to construct arbitrary meaningful semantics for instantiation numbers).

   Thus S_1 might exist in assemblages A_1 and A_2 as a unique SI name, and denote a different SI in each assemblage. The *globally unique* name of any SI is a creation pathname, starting with some (always present) root assemblage $TopLevel_1$ and containing the names of all the parent assemblages of S_i down to its immediate parent followed by the schema name and instantiation number of S_i. We define $TopLevel_1$ as the root of our creation hierarchy, and, from now on, it is understood to always exist. So, in previous definitions, where we mentioned some SI, S_j, the globally unique name of that SI must now be $TopLevel_1 S_j$ (this is very similar to directory and file naming in a standard file system). Apart from this definition we shall make no further use of global names, and from now on it is understood that any instantiation number is *local* to the assemblage in which the SI is created (that is $TopLevel_1$, if no other assemblages are present).
5. From IB we can deduce two maps, P and C, characteristic of the assemblage. C is the network connection map. P is *not* a connection mapping in the sense that C is. It maintains

a set of port equivalence definitions; it does not connect port i on some assemblage, A_j, to port k on one of the component SIs of A_j, instead it states that port i on assemblage A_j *is* port k on the component assemblage (the port automaton semantics will make this clear).

6. We do not rule out IB continuing to function after the network has been set up, adding or subtracting SIs according to some rule. We shall have to embody this explicitly in the assemblage semantics, however. One way to include an existing SI into an assemblage is to have IB recreate a copy of the desired SI. There does not seem to be any easy way to share SIs between assemblages, other than to provide each with a local copy (thus preserving modularity). Also there is no mechanism to create an assemblage from a set of existing SIs, i.e., *bottom-up*.

We define the formal semantics for the assemblage as follows: Definition 28 extends the port connection automaton of Definition 26 to a network automaton, and Theorem 29 states that any such network is essentially a port connection automaton, and can be described by a set of port automata and a network connection map.

The S and IB components define a set of SIs, in that IB operates upon the elements of S according to some internal rules, and produces a set of SIs. The semantics of each such component SI in the assemblage N is a port automaton (Theorem 46). So IB and S define a set of port automata, $\mathcal{AS}$.

From IB we can construct a network connection map describing how the port automata in the set $\mathcal{AS}$ are connected to each other. The network connections, as provided by C, are mappings from a port on one SI to a port on another SI (subject to the port direction rules). From this, we can simply construct the mapping between the port automata, which are the semantics of each SI.

We can, therefore, identify the port connection automaton described by the S, IB and C components of the assemblage definition as the port automaton which is the semantics of the assemblage. Consider how this affects the other components of the assemblage definition.

The port sets of the port connection automaton are composed of, by Definition 26, those ports on component automata which have not been subsumed by connections. In general, we may want to denote these ports by *different names* when they are referenced as part of an assemblage, rather than when they are referenced on their own schemas (since the assemblage will probably carry out a different function of these port values). Hence, we provide the assemblage definition with lists of input and output ports IP, and OP, and with an equivalence mapping function from these lists to port names on component automata of the assemblage, P. Yet, by the port connection automaton semantics of the assemblage, the assemblage ports are *not separate* ports

from the component SI ports; they are simply different names for these ports.

Previously, we have mentioned that we wish to have IB continue to operate once the assemblage has been set up. We split IB into two sections: one section, IB_i, which sets up the assemblage, and one section, IB_c, which continues to operate once the assemblage has been set up. Previously, therefore, the sematics has been constructed around the IB_i component; its duty is to transparently set up the network – it is not a part of the 'computations' done by the assemblage. IB_c is, however, part of this computation in some sense, in that it may 'react' to incoming data, or the internal state of the network, and add or subtract SIs to or from the network.

We represent IB_c formally as a port automaton, P^{ibc}; thus, the set of port automata actually comprising the port connection automaton, which is the semantics of the assemblage, is the set:

$$\mathcal{AS} \cup \{P^{ibc}\} \qquad \text{(V.51)}$$

We extend C and P by whatever connections are necessary to allow IB_c to operate. ∎

Definition 60

We define a task-unit simply as a special instance of the assemblage construct; we now establish this more formally. The syntax of task-unit definition (again, abbreviated by keyword omission) is:

$$[\, N\ (IP)\ (OP)\ (V)\ \ (PS)\ (MS)\ \ (IB)\,]$$

where,

- N is a name for the task-unit,
- IP is a list of input port names and their types,
- OP is a list of output port names and their types,
- V is a list of internal variable names and their types,
- PS is a list of perceptual schema names,
- MS is a list of motor schema names,
- IB is a behavior section which initializes the task-unit network.

The only differences between a task-unit and an assemblage are that a task-unit explicitly labels sensory and motor components, for the purpose of creating a sensori-motor hierarchy, and that a *precondition* can be associated with a task-unit. We have to establish that these features can be constructed with an assemblage.

Consider a task-unit N; the assemblage N' which is equivalent to N can be constructed as follows:

$$\begin{array}{lcl} N' & = & N \\ IP' & = & IP \\ OP' & = & OP \\ S' & = & PS + MS \end{array}$$

The list of component schemas in N' is simply the concatenation of the perceptual and motor schemas in N, and we denote list concatenation by +. A task-unit deinstantiates when all its component SIs have deinstantiated.

We can associate a *precondition* $Pr(c)$ with a task unit, T, which has the following effect: If some condition, c, is fulfilled, then task unit T is instantiated. We prove this by building a schema which obeys this behavior. Let ρ be a schema which computes the condition c. We can instantiate ρ with the name, instantiation number, variables and connections we want T to have, represented in ρ's internal variables. Since ρ computes c, it can determine the appropriate time to instantiate T from this information.

We have thus formalized the conjecture that a task-unit is simply a special case of the assemblage construct. ∎

Definition 61

We now define the *port array*. Sometimes it is more useful to specify a port on a schema by means of a numerical index, rather than as a symbolic name. We augment the syntax of schema declaration as follows:

$$\begin{array}{lcl} schema & ::= & [\ N\ \langle IP\rangle\ \langle OP\rangle\ \langle V\rangle\ \langle B\rangle\ ] \\ IP, OP & ::= & \langle PortList\rangle \\ PortList & ::= & \wedge \mid \langle Pname\rangle\langle PortList\rangle \mid \langle Parray\rangle\langle PortList\rangle \\ Pname & ::= & \langle Portname\rangle : \langle Type\rangle \\ Portname & ::= & Character\ String \\ PortArray & ::= & \langle Portname\rangle_{lb} \ldots \langle Portname\rangle_{up} : \langle Type\rangle \mid \langle Portname\rangle_{lb} \ldots : \langle Type\rangle \\ lb, up & ::= & Integer\ Numbers \\ Type & ::= & t,\ such\ that\ t \in \mathcal{T} \end{array}$$

∎

For example, a valid port definition is: $(f_1 \ldots f_9 : Real)$, meaning that there are nine ports, each of which can then be referenced individually as some f_i, $1 \leq i \leq 9$, and each of which will deal with a real number. This bounded array notation is simply related to our schema semantics, since it defines a list of ports as easily as does the list of names. We treat the unbounded array notation very differently.

An unbounded declaration, such as ($f_1 \ldots$: *Real*), indicates that there are an *undefined* number of ports, any of which can be referenced as $f_i, 1 \leq i$, and which is a real number. We limit the unbounded notation *to use with assemblages only*, and we define their semantics in terms of the assemblage semantics as follows.

Observation 62

Using the assemblage definition and the port array notation we can construct an SI whose port sets are dynamic in size.

For simplicity, and without loss of generality, we shall consider only the input ports. Consider the following trivial schema description:

```
[  Element
   Input-Port-List:     ( e$in:Real )
   Output-Port-List:    ( )
   Variables:           ( )
   Behavior:            ( )           ]
```

We now have all the prerequisites we need to create an SI with an indefinite number of ports.

```
[  Dynamic Assemblage
   Input-Port-List:     ( d$in1 ...:Real )
   Output-Port-List:    ( )
   Variables:           ( timer:Integer n:Integer i:Integer)
   Schemas:             ( Element )
   Initialization:      ( If ( (time mod n)=0)  Then
                          i := time  div  n
                          element(≡d$in_i )()()
                         Endif
                         time:=time+1     )                  ]
```

Consider the behavior of an SI of Dynamic. Its initialization behavior, depending on how its variable *time* and n are initialized, will make one instance of Element every n counts (ignoring eventual overflow of the *time* variable). Each time an instance of Element is created, an equivalence map is made between the input port on the SI and the appropriate member of the port array of the assemblage. Hence, Dynamic is a case of an assemblage which is continually adding to its network of component SIs.

Notes:

1. We can apply a similar method to deduce the existence of SIs with dynamic output port sizes.
2. And by a similar argument, port sets can shrink as well as expand.

■

CHAPTER VI

VERIFICATION IN $\mathcal{RS}$

In Chapter 5, we used automata theory to develop and explore the semantics of $\mathcal{RS}$. We have thus developed an analytic tool for our model of computation. At the same time, this tool is not as useful in the synthesis of programs in $\mathcal{RS}$, as we shall demonstrate. As discussed in the introduction to Chapter 5, a formal aid in the development and verification of programs is very important, especially where real-time control is involved. We shall develop a method to determine if some schema or instantiation of that schema satisfies a given specification, that is, to *verify* that schema or SI. We shall also, though more informally, deal with developing the specification. The real-time aspect of robotics will be considered only in the sense that $\mathcal{RS}$ programs give rise to real-time robot behavior. We shall not deal with the issues of whether programs in $\mathcal{RS}$ meet specific real-time deadlines, although we do recognise that this is an important area which needs to be considered more fully.

It is necessary to place some direction on the role of verification and formal program design in $\mathcal{RS}$: Since, usually, "the range of computer sophistication of robot users is large: ranging from factory personnel with zero programming experience to Ph.D.'s in Computer Science"[50], we would not expect all programs in $\mathcal{RS}$ to be derived formally. Instead, we would provide a number of building blocks (such as the primitive schemas, and various useful assemblage schemas) with well-understood and quantified behavior, so that useful programs can be constructed informally; while developing a formal proof system which can be used by more sophisticated users to explicitly verify program behavior. The assemblage structure provides an ideal tool for top-down development. We shall also introduce the concept of the ***assertion template*** and the ***abbreviation predicate*** which will be used in Chapter 8 in conjunction with the task-unit shorthand as a top-down design methodology for developing full schema specifications (the full syntax described in Chapter 4).

§1. Network and SI Verification

The basic mode of computation in $\mathcal{RS}$ is the interaction of a number of concurrent SIs. Verification of the behavior of such a network of SIs must proceed in two steps:

Definition 63

1. For each SI in the network S^i_j, characterize its behavior by some formal assertion ς^i_j.

2. From the conjunction of all ς^i_j in the network, and the connection map for the network, deduce a formal assertion describing the behavior of the network. ■

Chapter 5 provides us with the basic machinery to complete these two steps as follows:

1' From Definition 38, in Chapter 5, and the schema description, for each SI in the network construct the *port automaton* which is the semantics of that SI.

2' From the set of port automata which is the semantics of the network component SIs and the network connection map, use Definition 30 of Chapter 5 to construct the *network automaton* which is the semantics of the network.

The network automaton constructed by 2' completely defines the semantics of the network of SIs. However, even for the most trivial of examples, 1' and 2' are complex and difficult to use in program development. So, while we maintain the port automata semantics for exploring the representational and computational power of the model, we adopt a *temporal logic* system[75, 56] for design and verification purposes. This has the advantage of preserving the formal structure developed in Chapter 5, while, at the same time, being easier to understand and use in program development.

Our approach is as follows: The automata-theoretic model of Chapter 5 will be used as an interpretation for a temporal logic. An assertion in temporal logic will then specify the behavior (in some defined sense) of an instantiation of a schema. We start by briefly presenting the basic structure of temporal logic.

§2. Temporal Logic

Temporal logic is a *modal logic*; that is, a logic involving the concepts of *possibility* and *necessity*. A *universe* in temporal logic consists of a sequence of states or worlds;

where state s' is accessible from state s if, through the passage of time, s develops into s'. In *linear-time* temporal logic each state has at most one successor and one predecessor state. A universe is thus a sequence of states of the form: $s_0, s_1, s_2, \ldots$ in which s_j is accessible from s_i, iff $i \leq j$; and for each s_i, there is at most one s_j which is *immediately accessible*, and $j = i + 1$. Linear-time logic can represent the sequence of states through which a program passes in execution, or the sequence of data values input and output from the program in its execution. A program would then be defined completely by the set of all possible such linear *traces* that it can produce. Concurrent events are represented in these linear traces as a non-determinstic interleaving of the events. Both deterministic programs and non-deterministic programs can be represented: For non-deterministic programs the set of traces must be expanded to include one trace for each possible non-deterministic option.

In *branching-time* temporal logic, each state can have more than one possible successor or predecessor state. Thus, a program is represented by a set of execution graphs rather than linear traces. Non-determinism can be explicitly represented in the graph by having a number of possible futures for any state, the actual future being a non-deterministic choice of one of these futures. We shall define that an assertion is associated with a program (a schema in $\mathcal{RS}$) only if that assertion is satisfied by all possible traces of that program (schema). In a linear-time logic, this means that we can reason about what *will* and *will not* happen (i.e., all traces satisfy the assertion or no traces satisfy the assertion); however, it means that we cannot reason about what *may possibly* happen (i.e., a linear trace exists which satisfies the assertion). This mode of reasoning is possible in branching-time logic.

We shall construct a linear-time temporal model for verification and program development in $\mathcal{RS}$. The reasons for this choice are as follows: A linear-time logic has simpler structure and operators than a branching-time logic. Also, we shall primarily be interested in describing what will definitely happen, and what will definitely not happen, in $\mathcal{RS}$ programs. Additionally, linear-time will represent the concurrency and non-determinism inherent in $\mathcal{RS}$. Hence, we do not (yet, at least) require a branching-time logic.

We start by defining the elements of the language of temporal logic. We make use of a set of basic symbols, consisting of individual variable, constant, proposition, function and predicate symbols. This set of symbols is partitioned into *global symbols* and *local symbols*. Global symbols have a uniform interpretation in *all* states, whereas local symbols may

change their meanings and values from state to state.

We use the set of boolean connectives: $\wedge, \vee, \supset, \equiv, \neg$; the equality operator $=$; and the first order quantifiers $\forall, \exists$. The quantifiers can not be applied to local symbols. The triple equivalence $A \equiv B \equiv C$, we define for convenience, to have the same truth value as $(A \equiv B) \wedge (B \equiv C)$. These operators are referred to as the classical operators.

The *modal operators* are $\Box, \bigcirc, \Diamond, \mathcal{U}, \mathcal{N}$, called respectively, the ***always, next, eventually, until*** and ***unless*** operators. The first three are unary operators, and the last two are binary. A logic which has the classical and modal operators we shall refer to as a temporal logic. We also use the *specification* operator $\langle P \rangle$, where P is a schema or schema instantiation name. A logic which has the specification operator we shall refer to as a dynamic logic.

A model or interpretation (I, α, σ) for this temporal logic language is: A global interpretation I, a global assignment α, and a universe σ. The interpretation assigns concrete constants, functions and predicates from a *domain, D,* to the global constant, function and predicate symbols. The global assignment assigns a value to each global free variable. The universe $\sigma = s_1, s_2, \ldots$ is an infinite sequence of states. Each state is an assignment of values to the local free variable, function and predicate symbols. Note that *global* symbols have a uniform interpretation over the universe. However, *local* symbols may assume different meanings and values from one state to another.

The truth value of a temporal formula or term (defined as in 1st order logic) τ under (I, α, σ) is denoted by $\tau \mid_{\sigma}^{\alpha}$. Let $\sigma^{(k)}$ denote the k-truncated suffix of σ, the sequence $s_k, s_{k+1}, \ldots$.

- If τ is a term or classical formula (with no modal operators), then $\tau \mid_{\sigma}^{\alpha}$ is the value of τ given by s_0 under α.

  $$\tau \mid_{\sigma}^{\alpha} = \tau_{s_0}$$

- $$(\tau 1 \wedge \tau 2) = true, \quad \text{iff} \quad \tau 1 \mid_{\sigma}^{\alpha} = true \ AND \ \tau 2 \mid_{\sigma}^{\alpha} = true$$

  and similarly for $\vee, \neg$, etc.

- $$\bigcirc \tau \mid_{\sigma}^{\alpha} = true, \tau \mid_{\sigma^{(1)}}^{\alpha} = true$$

  $\bigcirc \tau$ is pronounced "next τ", and $\bigcirc \tau$ in σ is true only if τ is true in the shifted sequence $\sigma^{(1)}$.

- $$\Box\tau = true, \text{ iff } for\ all\ k \geq 0,\ \tau\ |^{\alpha}_{\sigma(k)} = true$$

  $\Box\tau$ is pronounced "always τ", and it means τ is true in all states.

- $$\Diamond\tau = true, \text{ iff } there\ exists\ k \geq 0,\ s.\ t.\ \tau\ |^{\alpha}_{\sigma(k)}\ istrue$$

  $\Diamond\tau$ is pronounced "eventually τ", and it means τ is guaranteed to eventually become true in some state.

- $$(\tau 1\ \mathcal{U}\ \tau 2) = true, \text{ iff } \begin{array}{l} there\ exists\ k \geq 0, s.t.\ \tau 2\ |^{\alpha}_{\sigma(k)} = true \\ and\ for\ all\ i,\ 0 \leq i \leq k,\ \tau 1\ |^{\alpha}_{\sigma(i)} = true \end{array}$$

  $(\tau 1\ \mathcal{U}\ \tau 2)$ is pronounced "$\tau 1$ until $\tau 2$", and it means that $\tau 1$ is continuously true until $\tau 2$ becomes true and $\tau 2$ is guaranteed to become true eventually.

- $$\begin{array}{l} (\tau 1\ \mathcal{N}\ \tau 2) = true, \text{ iff} \\ \Box\tau 1\ |^{\alpha}_{\sigma}\ OR\ (\tau 1\ \mathcal{U}\ (\tau 2 \wedge \neg\tau 1))\ |^{\alpha}_{\sigma} = \ true \end{array}$$

  $(\tau 1\ \mathcal{N}\ \tau 2)$ is pronounced "$\tau 1$ unless $\tau 2$", and it means that $\tau 1$ is true until $\tau 2$ becomes true and is false thereafter, but $\tau 2$ may never become true. This is a stronger definition of the unless operator than is usually found in the literature. We adopt it here mainly to define the *Inst* predicate in Axiom 69.

- $$\forall x.\tau\ |^{\alpha}_{\sigma} = true, \text{ iff } for\ all\ d \in D, \tau\ |^{\alpha'}_{\sigma} = true$$

  where $\alpha' = \alpha \circ [x \leftarrow d]$ is the assignment obtained from α by assigning d to x; x is a global variable.

- $$\exists x.\tau\ |^{\alpha}_{\sigma} = true, \text{ iff } for\ any\ d \in D, \tau\ |^{\alpha'}_{\sigma} = true$$

  where $\alpha' = \alpha \circ [x \leftarrow d]$ as above.

Quantification is a subtle issue in temporal logic; a quantification must not change a local variable within the scope of some temporal operator, otherwise some odd effects start to happen. For example, from the axiom $\forall x \Diamond(x < y) \supset \Diamond(t < y)$, it is possible to derive $\forall x \Diamond(x < y) \supset \Diamond(y < y)$, which cannot be valid for any local variable y.

Temporal logic allows one to prove *liveness* properties (assertions that desired events will definitely occur) and *safeness* properties (assertions that undesirable events will definitely not occur). For a set of axioms and theorems of temporal logic see [56, 57]. If τ is true in a model (I, α, σ) then we say that τ *satisfies* that model.

§3. The Model

In order to use temporal logic to prove program correctness in $\mathcal{RS}$, we need to define how our semantics can be considered a model for temporal logic (TL). The main thing a model for TL must do is to provide a universe of states, and describe how each state is mapped onto the meaning and values of local variables, predicates and functions.

We base the fundamental structure of our definition of the *behavior* of an SI on the trace model of Nguyen et al. [1984]. However, we shall introduce a number of extensions to their basic definition. The major difference is that we shall introduce a method for verifying that a schema description satisfies some assertion about that schema. The Nguyen et al. model provides only for proving that a process network satisfies some assertion, *given* the assertions describing the component processes of the network.

$\mathcal{RS}$ uses dynamic process creation; the instantiation and deinstantiation of schemas. The Nguyen et al. model deals only with static networks[1], hence we must decide how to represent the creation and death of SIs. $\mathcal{RS}$ is a model of computation for robots; one of our defining characteristics of robots is that they *sense* some external world, and that they can act to affect some aspect of this external world. The ultimate reason for a computation in $\mathcal{RS}$ is to (intelligently) bridge the gap between sensation and action. It is important that our TL model admits a class of *world-descriptive* predicates, which are capable of reflecting the state of the external world. An assertion describing a primitive schema will forge the connection between world predicates and some assertion about SI

[1]This was extended by a more recent paper, Nguyen et al. [1986], to deal with recursive subcomponents.

behavior. In this way, the robot's ***internal model of the world*** is linked to the real external world.

Finally, the Nguyen et al. model does not provide any mechanisms for the ***development*** of assertions and programs — it simply provides for verification of network assertions given the component process assertions. We shall introduce a mechanism, called ***abbreviation predicates***, by which the top-down development of schema programs and their assertions can occur in parallel.

We make one simplification for clarity. Although temporal assertions should rightly be associated with the port automaton which is the semantics of an SI, we shall abbreviate this process and associate the assertion directly with the SI. This simplifies the process of program development, and also unburdens the notation for proving program correctness.

Definition of Behavior:

Definition 64

An *event* is a pair (p, x), where p is a port name and x a data value received or transmitted on that port. It is an *input event* if p is an input port, otherwise it is an *output event.* A full port name is of the form $S_i(a)$, denoting port a on instantiation i of schema S. A trace t on a set of ports is a finite sequence of events on those ports. ∎

Note that this is simply related to the definition of ***port-event*** and ***input-output trace*** in Chapter 5 (we formalize this relationship presently).

Definition 65

An *observation* on a set S of ports is a tuple (t, In, Out), where t is a trace on S; function In maps the input ports in S to $\{T, F\}$, and Out maps the output ports of S to $\{T, F\}$. Informally, these functions denote that at the end of t, the port is now ready, (T), or not, (F), to input (output) a value, *but* has not yet done so. ∎

An input-output trace of Definition 32 is a history of the input and output events of some SI up to some point. This is exactly the information we want to hold in an observation; however, we want to restructure it. We shall use the observation as the state in TL. Our reasons for wanting to include the whole history of the SI, so far, in each observation is twofold: firstly, for expressiveness, and secondly, so that we can make use of Nguyen et al.'s [1984] proof of non-interference, soundness and relative completeness.

Definition 66

If B_n is an input-output trace on an SI, P, then the observation O corresponding to that input-output trace is derived as follows:

The trace t_o of O is of length n and for each port event $e_k=(i,x,y)$, if $x = \#$, then the kth event of t_o is (i,y), and if $y = \#$, then the kth event of t_o is (i,x).

If, after the trace B_n, the port automaton which is the semantics of P is in a state q, then:

$$\begin{array}{llll} \forall i \in L_x & IF\ \ \delta(q,(x,i)) = \{q\},\ x \in X_i \setminus \{\#\} & THEN\ In_o(i) = T \\ & & ELSE\ In_o(i) = F \\ \forall j \in L_y & IF\ \ \beta_j(q) = y,\ y \in Y_i \setminus \{\#\} & THEN\ Out_o(j) = T \\ & & ELSE\ Out_o(j) = F \end{array} \qquad \text{(VI.1)}$$

where $\setminus$ is the set difference operator. In_o and Out_o are the local input and output functions for the observation. Informally, they indicate what events can happen next. ∎

Definition 67

A *behavior* on a set of ports, S, is an infinite sequence of observations, $s_0, s_1, \ldots$, satisfying the following:

1. The trace of s_0 is empty.
2. For $0 \leq k$, the trace of the observation s_{k+1} is the trace of the observation s_k extended optionally by one event (x,i). If i is an input port, then $In_k(i)$ must be true, and if i is an output port, then $Out_k(i)$ must be true.

An SI (the port automaton which is its semantics) is characterized by its *set of behaviors*. ∎

Having defined behavior, it is now necessary for us to define how a behavior may satisfy a temporal assertion; that is, how an observation can be considered as a state. First, we define the *restriction* of a trace to a port, or set of ports, as the subsequence of the trace containing exactly the events occuring on the port, or set of ports. In similar fashion, the restriction of a communication function *In* or *Out* to a port, or set of ports, is the function obtained by restricting the domain of the function to the port, or set of ports. The restriction of observation and behavior to a port, or set of ports, is defined similarly.

Definition 68

An observation can be considered as a state:

1. Assign to each local variable k, the sequence $[a_0, \ldots, a_n]$, where $[(a_0,k), \ldots, (a_n,k)]$ is the restriction of the trace of the observation to port k.

2. Assign to local functions In and Out the communication functions of the observation.
3. Assign to local predicate symbol $\ll$ the "precedes" relation on the trace of the observation: $(h, m) \ll (k, n)$, iff the mth event on port h occurs before the nth event on port k in the trace, and $\ll$ is, therefore, a total ordering. ■

This definition sets up an important convention: Local variables in an assertion about some set of ports *will have the same names as the ports*, and the value of these local variables will be *the sequence of data values output (input) on that port so far.* We now define some elementary sequence notation which we shall make use of later in this chapter: A sequence, s (e.g., the value of a local variable), is denoted $[a_0, \ldots, a_n]$. The length of s is denoted $|s| = n + 1$. The ith element of a sequence is denoted $s(i) = a_i$. The last element of a sequence is denoted $\ell(s) = s(n) = a_n$. We shall use # to denote the null or trivial value. For convenience, we set $\ell([]) = \#$, i.e., the last value of an empty sequence is the null value.

Instantiation and Deinstantiation:

We have developed a universe for TL and a mapping from each state in the universe to the values and meanings of local functions, variables and predicates based on the trace model of Nguyen et al. [1984]. The first necessary extension we make is to consider the dynamic creation (and death) of SIs. We shall have to think more closely about schema names, and how they are mapped onto variables and constants in TL. For any program in RS there is a finite set of schema names, one per schema defined. Schema names are represented by global (i.e., the same in all states) constants in TL. The set of SI names is the Cartesian product of the set of schemas defined and the natural numbers (excluding 0); that is, all possible instantiations of all schemas. An SI name is, again, a global constant in TL.

We introduce the local predicate $INST(S_i)$, which takes as input an SI name, or a variable which ranges over SI names. $INST(S_i)$ is true, if and only if, schema instantiation S_i has been created and has not since deinstantiated. Since $INST(S_i)$ is a local predicate, its value will change from state to state; and we must define a mapping which allows us to determine from an observation what value $INST(S_i)$ will have.

In order to define $INST(S_i)$ within our behavior framework, we need to express it in terms of some form of port communications. The $INST(S_i)$ predicate is taken into our

definition of an *observation* by assuming that no SI S_i will begin its computation until it reads the trivial value (#) from a unique port, *startup*, and will not deinstantiate without writing the trivial value to a unique port named *closedown*. In this way, the instantiation of an SI can be considered to have happened when the SI reads from its *startup* port. The $INST(S_i)$ predicate becomes true with the input event $(startup, \#)$, and becomes false with the output event $(closedown, \#)$. We need to assume that $In(startup)$ is true for every SI initially; so we have:

Axiom 69

$$(startup = [\#]) \supset (\ INST(S_i)\ \mathcal{N}\ (closedown = [\#])\ ) \tag{VI.2}$$

Notice that this embodies the fact that a schema instantiation can continue processing forever; however, if it does ever deinstantiate, then $INST$ becomes false by our stronger definition of unless.

World-Descriptive Predicates:

In any state the world-descriptive predicates (occaisionally abbreviated, for convenience, to world predicates) reflect the structure of the world external to the robot; thus they allow us to give meaning to the effects of computation in $\mathcal{RS}$ on the outside world. For simplicity, the parameters for a world-descriptive predicate will be local variables in our TL model. Remember, local variables have the same name as ports, but their value is the sequence of data input or output on that port so far. For convenience, in world-descriptive predicates, local variable parameters will be *understood to refer to the last* value in the sequence; this saves us writing $\ell(x)$ for local variable x all the time. We shall also allow global constants and variables as parameters to these predicates, and we shall, by context, indicate which is which. The following list includes all the world predicates we shall use:

$POSITION(i,j,\delta)$	true if joint i is at a position k satisfying $\ell(j)-\delta \leq k \leq \ell(j)+\delta$
$FORCE(i,j)$	true if joint i experiences force $\ell(j)$
$WRISTPOS(j,\delta)$	true if the Cartesian position of the wrist $p=(p_x,p_y,p_z)$ satisfies $\ell(j)-\delta \leq p \leq \ell(j)+\delta$. for $\delta=(\delta_x,\delta_y,\delta_z)$
$WRISTANG(r)$	true if the 3 wrist angles $= r = (r_\theta, r_\phi, r_\psi)$
$WORLDOBJECT(i)$	true if object named i is in robot's environment; in which case the following predicates apply to i:
$OBJPOS(i,j,\delta)$	true if the Cartesian position p of object i satisfies $\ell(j)-\delta \leq p \leq \ell(j)+\delta$.
$OBJANG(i,r)$	true if object i has orientation $r=(r_\theta,r_\phi,r_\psi)$
$OBJFEA(i,f)$	true if object i has feature vector f
$OBJSTF(i,k)$	true if object i has stiffness $k=(k_x,k_y,k_z,k_\theta,k_\phi,k_\psi)$ in its object centered frame $\mathbf{T}_o$

Note on precision: A precision parameter δ has been build into the definition of the $OBJPOS$ and $WRISTPOS$ predicates. A more concise definition of world predicates would associate some form of precision with each predicate. The positional precision that we have defined will suffice to demonstrate the use of the concept.

Abbreviation Predicates:

Abbreviation predicates are *local predicates*, they are simply 'short' names for longer temporal assertions (the exact structure of which may not have been developed). These predicates should have port names as parameters, to indicate that they are actually assertions on these ports. However, when convenient, we shall consider these parameters understood. The role of abbreviation predicates is as 'stand-ins' in an SI assertion until the detailed assertion has been constructed. By their name and parameters they can indicate (in much the same way as precedure names in a programming language) what 'purpose' the detailed assertion will fulfill. They also indicate how the detailed assertion will eventually fit into the overall system.

For example, we introduce the abbreviation predicate $PosControl(m,i)$ to indicate that port m exerts position control over object i; this predicate is actually short for some assertion of the form:

$$PosControl(m,i) \equiv \Box(\Diamond In(m) \wedge (OBJPOS(i,m,\delta) \equiv In(m)) \wedge (|\, m \,| = 0 \supset (In(m) \wedge OBJPOS(i,p_0,\delta))) \tag{VI.3}$$

Recall that local variable m has as its value the sequence of data values which have been

input on port m. The first line here tells us that port m exerts position control on the object i. The second line gives us the initial conditions — the joint is at position p_0, initially (i.e., when the local variable m is empty). This initialization is necessary to define the position of the robot before any processing has been done. *PosControl* sums up the notion that position control of the object is possible over port m. Abbreviation predicates allow us to pursue top-down development of schemas and assertion templates for them.

§4. SI Assertions

An SI is the 'machine' responsible for the production of an observation from its predecessor, producing a behavior. Previously, we have noted that an SI is characterized by its set of behaviors. If an assertion, $\mathcal{A}$, is an ***assertion template*** for a schema S, then *every* behavior of any instantiation S_i of S satisfies $\mathcal{A}$. Additionally, a behavior is a behavior of S_i only if it satisfies $\mathcal{A}$. In Section 5, we shall consider how the assertion can be derived from the details of the schema description.

Definition 70

We denote that ς is an *assertion template* for a schema S by writing:

$$\langle S \rangle \quad \varsigma$$

This is interpreted as meaning that *every behavior of an instantation of S satisfies* ς, where ς is a temporal assertion in which:

- The only free local variables are those constructed from the ports of S, as in Definition 66.
- The only local function symbols are In, Out.
- The only local predicate symbols are $\lessdot$ and $INST$.
- No value is bound to any global variables.

If it is important to mention the instantiation number of the schema, then we use the form:

$$\langle S \rangle \quad \forall i.\varsigma(i)$$

where i now denotes the instantiation number, and $\varsigma(i)$ means that the assertion ς may involve i. ■

This definition applies uniformly when S is an assemblage definition (a network of SIs), and the ports occuring in ς are what we have defined as the assemblage ports (i.e., no connected component SI ports are mentioned)[2]. Note that, in dynamic logic for sequential programs, $\langle A \rangle\ b$ usually means b becomes true once A has finished executing. In $\mathcal{RS}$, $\langle S \rangle\ \varsigma$ means ς becomes true once S has been instantiated, and stays true until S deinstantiates.

Definition 71

When a schema S is instantiated to produce an SI, say S_i, then a temporal assertion ς' associated with that SI is produced from the assertion template ς by *binding values to the global variables*, in particular, the instantiation number i, and the assertion $\varsigma*$ generated:

$$\langle S_i \rangle \quad \varsigma* \equiv (INST(S_i) \wedge \varsigma') \ \vee \ (\neg INST(S_i) \wedge \aleph_{S_i}) \tag{VI.4}$$

where $\aleph_{S_i}$ is the assertion of the 'dummy' schema constructed from S_i as described in the semantics of deinstantiation in Chapter 5, Definition 48; and $INST$ is the special instantiation predicate. ∎

Note that we shall not explicitly use $\aleph_{S_i}$, and it was simply included for compatibility with the semantics of deinstantiation. This predicate essentially describes how S_i behaves before it has instantiated and after it has deinstantiated. For simplicity we shall use the predicate $INST*(S_i)$ to mean that S_i has been instantiated and that the assertion $\varsigma*$ now holds.

The primitive schemas are the connection in $\mathcal{RS}$ between assertions which describe port operations and the external world. We shall use the following primitive schemas and their assertions in examples:

Definition 72

The primitive sensory schema Jpos, the *ith* instantiation of which reports on the position of the *ith* degree of freedom on the robot mechanism through its output port x (see Definition 11 of Chapter 4):

$$\langle JPOS \rangle \quad \forall\, i.\ \Box(Out(x) \wedge Out(x) \equiv POSITION(i, x, \delta_i)) \tag{VI.5}$$

Recall that the value of the *local variable* x in some state is the sequence of data output on the *port* x up to that state. Informally, this assertion is read as: Jpos is always ready to output a

[2]Nguyen et al. make this point with respect to their processes and subprocesses.

value on its port x, and it is always true that the current value transmitted by x is the position of the *ith* degree of freedom within the range specified by δ_i,[3] a global variable which specifies how 'well' the *ith* joint can be controlled. i is the unbound global variable denoting the instantiation number of Jpos. ■

The values of all δ_i come from the characteristics of the particular robot system which is being programmed; this data can be stored in tabular form, and retrieved internally in $JPOS$ (since it can find out its instantiation number using the #I operator), or provided as an initial variable value to $JPOS$ by the creating SI (more cumbersome). We shall assume $JPOS$ locates its own δ_i internally.

Definition 73

The primitive motor schema Jmot, the *ith* instantiation of which accepts a position on its port y and sets the *ith* degree of freedom of the robot system to that value within a precision given by δ_i (and see Definition 12 of Chapter 4):

$$\langle JMOT \rangle \quad \forall\, i.\ \Box[\quad (In(y) \equiv POSITION(i,y,\delta_i)) \wedge \Diamond In(y) \wedge ((\mid y \mid\ = 0) \supset (In(y) \wedge POSITION(i,p_0,\delta)))\ ] \qquad \text{(VI.6)}$$

Informally, this is read as: Jmot will always eventually input a value on its port y, and when it does, it will not accept another value on the port until it has set the *ith* degree of freedom on the robot to the value received. Again, i is the unbound global variable denoting the instantiation number of Jmot. Initially, the joint is defined to be at p_0 and the SI is ready to input a position command. ■

Definition 74

The primitive schema Jtact, the *ith* instantiation of which reports via its output port c, whether the *ith* link is touching something:

$$\langle Jtact \rangle \quad \forall i.\Box(Out(c)\ \wedge\ ((\ell(c)=1) \equiv (\exists j \geq 1.FORCE(i,j)))\ ) \qquad \text{(VI.7)}$$

That is, we say that i is in contact with something if link i experiences a force on it. Notice that we are defining tactile sensation in terms of force sensing here, for simplicity. If we were going to need to do tactile pattern recognition this might not be the best definition. j is a global variable, and we use $\equiv$ to stress the fact that c will deliver a 1 if, and only if, there is some positive force on the link. ■

---

[3]See the definition of POSITION, page 143.

Definition 75

The primitive visual sensory schema is EOB, an instantiation of which reports on some external object. While EOB is a primitive schema in the $\mathcal{RS}$ sense, we note that it actually represents quite a *high-level* unit of visual information. It is important that an EOB always stand for the *same* object for as long as that object is in the field of view. We are content (for the time being) with such an obviously high-level definition of visual input, but recognize that this hides much complexity.

$$\langle EOB \rangle \quad \forall i.\Box[\quad \bigwedge_{j=1\ldots3} Out(f_j) \land WORLDOBJECT(i) \land \\ OBJPOS(i, f_1) \land OBJANG(i, f_2) \land OBJFEA(i, f_3)\,] \tag{VI.8}$$

where the ports of the EOB are as defined in Chapter 4, Definition 14. ∎

Note the connection between the instantiation number i and the object – this tie is being used to implement *visual continuity*. If i is an object in the world, then there is only one EOB_i which represents it, and that EOB_i represents *only it*. Recall that this follows from our isolation of the instantiation number from the *semantics* of an SI; we are free, therefore, to define what meaning is most convenient for the instantiation number.

The EOB assertion above is *passive* in that it does not demand that the robot have any EOB instantiated at all; only that, if they do exist, then they correspond to objects. We introduce a *perception axiom*, which *forces* the robot to sense its environment [4].

Axiom 76

$$\Box(\forall i.WORLDOBJECT(i) \supset INST*(EOB_i)) \tag{VI.9}$$

This *total perception axiom* is a good one to work with, in that it relieves the robot from having to search explicitly with its sensory apparatus, e.g., *move around its sensors* to sample parts of the environment when it does not have an appropriate EOB instantiation for some task. Although we recognize that this form of sensory search is an important area, we do not investigate it in our current framework. The perception axiom is essentially a 'place-holder' for future work in this area. A more realistic axiom is:

$$\Box(\forall i.WORLDOBJECT(i) \supset \Diamond INST*(EOB_i)) \tag{VI.10}$$

The EOB assertion guarantees that every EOB relates to an object in the world, and this axiom states that, eventually, the description of the current world will be complete. This

---

[4] Or, more precisely, it allows us to infer that the robot has sensed its environment.

makes explicit the fact that there may not *immediately* be an SI which represents some given object, but that, after some time (in which, for example, our omitted sensory searching might occur), the appropriate EOB SI will come into existence.

§*4.1 Proof Rules*

We take our own versions of the proof rules developed by Nguyen et al. [1984] for their trace model. The differences between these rules and those of Nguyen et al. are mostly cosmetic. In their system, connected ports have the *same name.* We have adapted their renaming rule to allow us to substitute the name of a port for the name of the port to which it is connected (connection renaming) as a special case of the general renaming rule (arbitrary renaming). We have introduced the equivalence port mapping to their assemblage/network rule, for convenience. The same effect can be achieved with their proof rule by applying their renaming rule after their network rule.

1. *Consequence Rule (CR):*

$$\frac{\langle S_i \rangle \ \varsigma, \ \varsigma \supset \rho}{\langle S_i \rangle \ \ \rho}$$

   If any assertion can be derived from an SI assertion, then this new assertion is also an assertion for the SI.

2. *Renaming Rule (RR):*

$$\frac{\langle S_i \rangle \ \ \varsigma}{\langle S_i' \rangle \ \ \varsigma'}$$

   where S', ς' are obtained from S, ς by one of the following:

   (a) If k is a local variable in ς, then k is replaced throughout with k' to get ς' and port k of S_i is replaced with a port k' to get S_i' (arbitrary renaming).

   (b) If k is a local variable in ς, then k is replaced throughout ς by a variable k', where port k of S is connected to port k' of some other SI. S_i' is then S_i with port k replaced by port k' (connection renaming).

3. *Assemblage/Network Rule (AR):*

$$\frac{\langle SI_j \rangle \ \ \varsigma_j, \ \ j = 1, \ldots, n}{\langle A \rangle \ \ \bigwedge \varsigma_j'}$$

where A is the assemblage with a set of component SIs $\{SI_j : j = 1, \ldots, n\}$, and where ς_j' is the assertion for SI_j, ς_j, but in which the following substitution has been made: If some port k of SI_j has been declared equivalent in the assemblage definition to some assemblage port k', then the local variable k' has been substituted for k throughout ς_j.

The proof of soundness (every specification which is provable from a set of schema assertion templates is true) and relative completeness (every specification which is true is provable) for our rules is the same, therefore, as for those of Nguyen et al., and is established in [65]. Before going into the details of producing ς^i for some schema description S^i, we shall give an example of going from a collection of SI assertions to an assemblage assertion (see Parts 1 and 2 of Definition 63). In order to do this, we need to define a *Liveness Axiom*.

Definition 77

We associate a *liveness* assumption (e.g., justice, fairness) with a network of SIs for the purpose of proving correctness. Let Ψ be such an assumption, then an SI is characterized by its set of Ψ-behaviors, behaviors which satisfy Ψ. All previous definitions hold if the set of behaviors of an SI is restricted to the set of Ψ-behaviors. Unless otherwise stated, we shall always adopt a *finite progress axiom* [4] of fairness in the form used by Nguyen et al. for synchronous networks:

$$\Box(|\, k \,| = n \wedge \Box\Diamond(In(k) \wedge Out(C(k))) \supset \Diamond \,|\, k \,| > n) \tag{VI.11}$$

where k is the local variable constructed from some port k, and $C(k)$ gives the name of the port to which k is connected, which is the same as the name of the local variable constructed from that port. ■

Informally, this axiom allows us to deduce that the transfer of information will occur between connected ports if the In and Out functions eventually become true together.

§4.2 *Example*

We shall analyze a version of the `move$joint` assemblage introduced in Chapter 4, which is essentially a position servo network. This schema is typical of many motor schemas we shall use in Chapter 8. It accepts a destination input on its port s, and it will eventually set the position of the ith joint (where i is its instantiation number) to the value of the input on s.

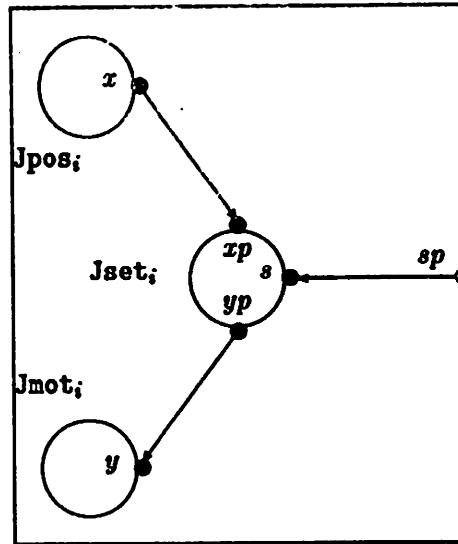


Figure 23: The `move$joint` Assemblage.

The `move$joint` assemblage is shown in Figure 23. Assertions for Jpos and Jmot have already been given in Definitions 72 and 73. The only other schema is Jset, which implements a position servo algorithm.

To circumvent the discussion of various servo algorithms, we define a function $F_v(d,x)$, which implements the servo algorithm. We say that the limit of $F_v(d,x)$ tends to d as x tends to d. This emulates a critically damped servo system with initial position x and desired setpoint d. That is, F_v generates the monotonic sequence:

$$FVD = [x, F_v(d,x), F_v(d, F_v(d,x)), \ldots\ldots] \tag{VI.12}$$

The behavior of Jset can now be formulated: It reads a setpoint on its port s, it then servos the joint i to the position given on port s. This servoing involves continuously reading the input port xp and using the FV function, as we have defined it, to compute a value to write to the output port yp, until the joint is at its desired position.

$$\begin{array}{lll} \langle Jset \rangle & \forall i . \Box (& [In(s) \equiv (\ell(xp) = \ell(s))] \wedge \\ & & [\neg In(s) \supset (\neg Out(yp) \equiv In(xp) \equiv (\mid xp \mid \, = \mid yp \mid))] \wedge \\ & & [\ell(yp) = F_v(\ell(s), \ell(xp))] \wedge \\ & & [\mid s \mid \, = 0 \supset In(s) \wedge \neg Out(yp)] \quad) \end{array} \tag{VI.13}$$

The last line in the assertion is the initialization condition; if no setpoint has been received, then **Jset** will not try to servo. The first line, then, is the servo 'trigger', when a setpoint is received no new setpoints will be allowed until the servoing of the robot to the last received setpoint has been completed (recall, we defined $\ell([]) = \#$, the last value of an empty sequence is a null). The two middle lines implement the servo algorithm; the current position is used in conjunction with the setpoint in F_v to set the joint one step nearer (as defined by F_v) its setpoint.

By (AR) we can now (loosely) write down the assertion characterizing **move$joint** as the conjunction of the three component SI assertions:

$$\langle move\$joint \rangle \qquad (VI.5 \wedge VI.6 \wedge VI.13) \tag{VI.14}$$

The connection map for the **move$wrist** assemblage is:

$$C(yp) = y \qquad C(x) = xp \qquad C(sp) = s \tag{VI.15}$$

This mapping is a simplified and convenient form of the network connection map and port equivalence map for the assemblage. We shall prove that **move$wrist** obeys the following assertion:

$$\langle move\$wrist \rangle \quad \forall i.\Box(\begin{array}{l} [(In(sp) \supset POSITION(i, sp, \delta_i)) \wedge \Diamond In(sp)] \wedge \\ [\mid sp \mid \; = 0 \supset (In(sp) \wedge POSITION(i, p_0, \delta_i))] \;) \end{array} \tag{VI.16}$$

That is, at first the joint is at its initial position, and whenever we feed it a setpoint through sp, eventually the new position will be reached. Our main job here is simply to link the world predicates, as given by the primitive schema assertions, to the assertion which characterizes **Jset**. We shall start by proving that the initialization clause, the second set of square brackets above, holds. Initially, all local variables are of length zero:

1. $\mid s \mid \; = 0 \supset \neg Out(yp) \wedge In(s)$, from $\langle jset \rangle$,

2. But since $C(yp) = y$, $\neg Out(yp) \equiv \neg Out(C(y))$, by (RR),

3. Hence, no communcation can occur, and $\mid y \mid \; = 0$,

4. And by $\langle Jmot \rangle$, $\mid y \mid \; = 0 \supset POSITION(i, p_0, \delta_i)$,

5. And by the network connection $C(s) = sp$, we have
   $\mid sp \mid \; = 0 \supset POSITION(i, p_0, \delta_i) \wedge In(sp)$,

6. And by (CR), this is also an assertion for $\langle move\$finger\rangle$.

Now we want to show that $(In(sp) \supset POSITION(i, s, \delta_i))$ and $\Diamond In(sp)$ is also an assertion for `move$wrist`; that eventually $In(s)$ becomes true, and whenever it does, the robot has been moved to the new setpoint. We shall do this in an inductive fashion.

A
- Assume $\mid s \mid = 0$ and $\mid xp \mid = \mid yp \mid = 0$, and a new setpoint arrives such that $\mid s \mid = 1$, and $\ell(xp) \neq \ell(s)$.
- Show that eventually we have $\mid xp \mid = \mid yp \mid = 1$, and either:
  - $POSITION(i, F_v(\ell(s), \ell(xp)), \delta_i) \wedge (\ell(s) \neq \ell(xp))$, or
  - $POSITION(i, s, \delta_i) \wedge (\ell(s) = \ell(xp))$.

B
- Assume $\mid s \mid = k$ and $\mid xp \mid = \mid yp \mid = m$, and a new setpoint arrives such that $\mid s \mid = k + 1$, and $\ell(xp) \neq \ell(s)$
- Show that eventually we have $\mid xp \mid = \mid yp \mid = m + 1$, and either:
  - $POSITION(i, F_v(\ell(s), \ell(xp)), \delta_i) \wedge (\ell(s) \neq \ell(xp))$, or
  - $POSITION(i, s, \delta_i) \wedge (\ell(s) = \ell(xp))$.

C We have defined F_v in VI.12 to converge, so that by induction on F_v, A and B above imply that eventually we shall have a value of xp such that: $POSITION(i, s, \delta_i) \wedge (\ell(s) = \ell(xp))$.

D But also note that whenever $\ell(s) = \ell(xp)$ is true, then we have $POSITION(i, s, \delta_i)$ (the converse is not always true). Thus we have $\ell(s) = \ell(xp) \supset POSITION(i, s, \delta_i)$. But by $\langle Jset\rangle$, $\ell(s) = \ell(xp) \equiv In(s)$. We can prove, therefore, that an assertion for `move$wrist` is $(In(sp) \supset POSITION(i, s, \delta_i))$ and $\Diamond In(s)$.

The first two steps, A and B, in more detail, are:

1. $\mid s \mid = k + 1$, and $\ell(xp) \neq \ell(s)$, by assumption (and k could be 0; these first few steps are the same for A and B),
2. $\neg In(s)$ by $\langle Jset\rangle$,
3. $In(xp) \equiv (\mid yp \mid = \mid xp \mid)$, by $\langle Jset\rangle$,

4. $In(xp)$, by the assumption of initial zero lengths,

5. From $\langle Jpos \rangle$, $\Box Out(x)$,

6. Now the basis part of our induction is that $|\ sp\ | = |\ yp\ | = 0$;

   (a) And by the liveness axiom, a transmission will eventually $\Diamond\ |\ xp\ | = 1 \wedge POSITION(i, xp)$

   (b) And by $\langle Jset \rangle$ $\Diamond Out(yp)$,

   (c) And $\ell(yp) = F_v(\ell(s), \ell(xp)) \wedge POSITION(i, xp)$,

   (d) And, again, by the liveness axiom, we will eventually get transmission to Jmot, $|\ yp\ | = 1$ and $\Diamond POSITION(i, F_v(\ell(s), \ell(xp)))$.

7. The inductive step is that, if $|\ xp\ | = |\ yp\ | = n$ and we have $POSITION(i, \ell(xp))$, then we shall eventually get $|\ xp\ | = |\ yp\ | = n$ and $POSITION(i, F_v(\ell(s), \ell(xp)))$,

   (a) Repeat steps (a) to (d) above with $|\ xp\ | = |\ yp\ | = n$. This is straightforward.

8. By induction, we get $\Diamond(POSITION(i, \ell(s)) \wedge (\ell(s) = \ell(xp)) \wedge (|\ xp\ | = |\ yp\ | = m))$ for some integer m.

§5. Reasoning about Schema Behavior

We extend the definition of behavior to allow us to verify that a schema description obeys a particular assertion. The extension is to allow a behavior to include events on *internal variables*. We shall associate a *label* and an assertion with each statement in the schema description. In a manner similar to that in which we proved assemblage assertions from the conjunction of the assertions of the component SIs, we shall prove a schema assertion from the the assertions about its statements. From the operational semantics described in Chapter 5, we construct for each type of statement in the schema description syntax, a *transition axiom*, an assertion describing the effects of executing this statement. A particular schema will have one such transition axiom for each statement in its behavior section, each labelled in a fashion similar to the program statements (as discussed in Chapter 5). The basic methodology for verifying sequential programs in this manner is described in Manna [1981]. We constrain the schema assertion to include *no*

reference to internal variables, just as we constrained the assemblage assertion not to mention the connected ports of its component SIs.

§5.1 *Extension of Behavior*

We extend the definition of an event to include internal variables:

Definition 78

An *event* on an internal variable is a pair $e = (x, i)$, where x is a datum and i is an internal variable name. A trace on a set of variables is a finite sequence of events on those variables. ■

Definition 79

An *observation* on a set of ports and internal variables is a tuple (t, In, Out), where t is a trace of events on the ports and variables; In maps input ports and variables to $\{T, F\}$; Out maps output ports and variables to $\{T, F\}$. ■

Definition 80

A *behavior* on a set of ports and variables is defined as per Definition 68, except using the definition of observation in Definition 77 (the previous definition). ■

Definition 81

As in Definition 68, we consider an observation to be a state in the universe for temporal logic. But, now, a local variable has the sequence of data values from the trace for both ports and internal variables. The precedence relation occurs on both internal variable traces as well as port traces. ■

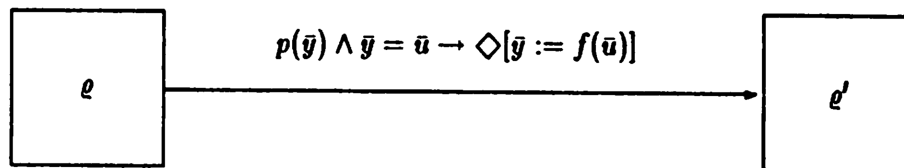
§5.2 *Transition Axioms*

The use of transition axioms to verify sequential programs is presented in Manna [1981]. We shall borrow the *location axiom* and *forward transition axiom* template from that work. The location axiom formalizes the sequential nature of statements in the schema behavior section. We associate a label with each statement in the schema behavior (as discussed in Chapter 5). In any state, the predicate $at\varrho$ is true for only one label ϱ. Let L be the set of labels in the schema:

$$\sum_{\varrho \in L} at\varrho = 1 \tag{VI.17}$$

By which, we mean that *exactly one* at_ϱ for $\varrho \in L$ is true, and all the rest are false. We shall conjunct the assertion about the statement with label ϱ, with the predicate at_ϱ, to indicate when that assertion becomes valid; and each assertion will include which $at_{\varrho'}$ becomes true next (making ϱ' the next statement to be executed after ϱ).

A transition axiom embodies the effect of a single statement; it is, essentially, the *symbolic execution* of the statement. Where $\bar{y}$ is the vector of program variables, the transition axiom template, the basic form of each transition axiom, is as follows:



The forward transition axiom states that if at_ϱ is true and $p(\bar{y})$ holds, and the current values of $\bar{y}$ are $\bar{u}$ with communication functions In and Out; then, eventually, $at_{\varrho'}$ will be true and the current values of $\bar{y}$ will be $f(\bar{u})$ and the new communication functions of this state are In' and Out' (where $\bar{u}$ are auxiliary global variables).

$$F_\alpha : [at_\varrho \wedge p(\bar{y}) \wedge \bar{y} = \bar{u} \wedge In \wedge Out] \supset \Diamond[at_{\varrho'} \wedge \bar{y} = f(\bar{u}) \wedge In' \wedge Out'] \qquad \text{(VI.18)}$$

The details of the effect of each statement we have already presented in Chapter 5. We shall reformat them here as forward transition axioms. Before we do this we need to discuss a schema body liveness axiom, just as we did a network liveness axiom.

Definition 82

If Out or In is asserted on a port p, then eventually a transmission will occur, $In(p) \supset \Diamond\neg In(p)$. ∎

Note that this particular liveness axiom does not facilitate analyzing the deadlock potential of a particular schema, since it states that deadlock will never occur. We justify this by observing that deadlock is actually a property of a particular network connection, rather than a property of any particular schema; hence its analysis is more appropriate at the network level.

We now list the transition axioms we shall use in schema assertion proving:

ASSIGNMENT: The syntax of assignment is as follows:

$$var := expr(a_1, \ldots, a_n)$$

where *var* is an output port or variable, and $a_1, \ldots, a_n$ are input ports or variables, and by *expr*(.) we denote some arithmetic function of parameters (.). Note that $a_1, \ldots, a_n$ are all unique port names; however, each port *could* occur more than once in the expression denoted by *exp*(.), e.g., $y := x * (x + 1)$. The choice, therefore, is either to have each occurrence of a port in the expression denote a *separate* read operation (and, hence, have to define the order of evaluation of the expression very carefully), or to have ***just one read*** occur no matter how many times a port name occurs in the expression (the value must then be buffered and used for each occurrance of the port name). We choose the latter (just one read) since it is much easier to use in programming. The list $a_1, \ldots, a_n$ is, therefore, a list of all the port names in the expression, regardless of how many times (> 1) they are mentioned in the expression.

The transition axiom for assignment is the most complex of all. This is true because of the coarse level of granularity involved in assignment. When considering port automaton semantics, we separated the assignment and port operation, and we isolated port operations into explicit input and output operations. This allowed us to define them easily. For convenience in relating transition axioms to schema statement syntax, we leave the assignment statement in its full form, possibly with implicit input and output operations built in. This is one reason why the transition axiom is so complex.

$$\begin{aligned}
at_{\ell} : (\mid a_i \mid \; = \; & m_i \wedge \; \mid var \mid \; = n) \longrightarrow \\
& \Diamond[(\bigwedge_{i=1\ldots n} In(a_i) \wedge \neg Out(var) \wedge \\
& \Diamond(\bigwedge_{i=1\ldots n} \neg In(a_i) \wedge \mid a_i \mid \; = n_i + 1 \wedge Out(V) \\
& \wedge \Diamond(\neg Out(var) \wedge \; \mid var \mid \; = n + 1 \wedge \\
& \;\; (\ell(v) = expr(\ell(a_1), \ldots, \ell(a_n)) \wedge at\ell'))))]
\end{aligned} \qquad \text{(VI.19)}$$

This is quite involved because it has both input and output operations built into it, and additionally, it embodies a *liveness assumption* for schema body verification. There are essentially three stages to the clause in square brackets above. The first stage simply says that all the input ports have *In* asserted in order to read in the necessary data for the expression. The second part says that eventually *In* will become false for all these input ports, and each of them will be extended by one event. This is the liveness assumption

— since, in order for this to happen, some external environment must communicate with the SI. However, unless we assume such a liveness property, then we shall be unable to reason past the first port operation. At the end of the second part, $Out(var)$ eventually becomes true, in order to write the evaluated expression. The third part again invokes the liveness assumption to say that eventually $\neg Out(var)$ happens; at which point we can say that the value of the last event on var is equal to the $expr$ evaluated on the last event of each of the input ports.

IF-STATEMENT: This has the syntax:

$$IF\ \ C(y)\ \ THEN\ \ \varrho'\ \ ELSE\ \ \varrho''\ \ ENDIF\ \varrho'''$$

where $C(y)$ is a conditional expression which has no port input or output operations, and the ϱs are statement labels. If $C(y)$ did have any port reads or writes, we could abstract them from it and into a previous assignment statement. The transition axiom for the IF is straightforward:

$$\begin{array}{lll} IF & at\varrho \wedge C(y) & \longrightarrow \Diamond[at\varrho'] \\ & at\varrho \wedge \neg C(y) & \longrightarrow \Diamond[at\varrho''] \\ ELSE & at\varrho \wedge true & \longrightarrow \Diamond[at\varrho'''] \end{array} \qquad \text{(VI.20)}$$

DEINSTANTIATION: This has the syntax:

$$Stop$$

If the deinstantiation occurs in instantiation i of schema T, the transition axiom is then:

$$at\varrho \longrightarrow \Diamond[\neg INST(T_i)] \qquad \text{(VI.21)}$$

INSTANTIATION: This has the syntax:

$$T_i(CI)(CO)(V)$$

where CI and CO specify the connection mapping for the new SI, and V specifies initial variable values; we denote the establishment of the new connections with the predicate $C*$, and the variable initialization with $\Phi*$

$$at\varrho \longrightarrow \Diamond[INST*(T_i) \wedge C* \wedge \Phi* \wedge at\varrho'] \qquad \text{(VI.22)}$$

FORALL: This has the syntax:

$$Forall\ \ S_i:\ \ T(CI)(CO)(V)\ \ Endforall$$

where S_i is a schema name and instantiation number variable, and $T(CI)(CO)(V)$ is a schema instantiation command:

$$at\varrho : true \longrightarrow \Diamond[\forall S_i . INST(S_i) \supset (INST * (T_i) \wedge C * \wedge \Phi *) \wedge at\varrho'] \qquad \text{(VI.23)}$$

That is, for all SI names (an infinite set of global constants), if $INST(S_i)$ is true, then assert $INST * (T_i)$.

Invariance Principle: We now discuss the basic tool for proving assertions from schema descriptions: the *invariance* principle [5]. If an assertion holds before the first statement of a schema, and for every statement in the schema body, given that the assertion holds before the statement, and if, after the transition axiom has been applied, the assertion still holds, then that assertion always holds for that schema. In other words, it is an invariant for that schema. If $\mathcal{A}$ is such an invariant, then we can say $\Box \mathcal{A}$.

Initial State, ϕ: It is necessary for us to define what holds in the state *before* the first statement executes. This we shall refer to as the initial state, and the conditions which hold there, we shall denote by ϕ. In the initial state:

1. All $In(v)$ and $Out(v)$ are true, for v a local variable.
2. All $In(p)$ and $Out(p)$ are false, for p an input or output port.
3. All $|v| = 1$, if v is a variable, and the value for each v is given by the map $\Phi*$.
4. All $|p| = 0$, for p an input or output port.
5. $\Diamond at\varrho 1$, the first line of the program will be eventually reached.

§5.3 *Example*

As an example, we shall do a simple buffer-style schema (of which we shall use many in Chapter 8) called `transform`. The schema has one input port p and one output port p_g, and one internal variable t. Part of the initialization conditions will have to be the contents of this internal variable when the schema was instantiated (that is, $\Phi*$, from the instantiation transition axiom). The behavior of the schema is simply to read its

---

[5] As that of Dijkstra, Hoare, Manna, etc.

input port into an internal variable t, and then write the value of t to its output port p_g. In implementing the SSG system, we shall use a number of schemas similar to this one in order to do coordinate transformations between object, grasp, hand and preshape coordinate systems. The schema description of `transform` is:

```
[  transform
   Input-Port-List:     ( p:Real )
   Output-Port-List:    ( p_g:Real )
   Variables:           ( t:Real)
   Behavior:            ( t:=p;          ϱ1
                          p_g:= t; )     ϱ2
                                              ]
```

We demonstrate that the following is an assertion for this schema:

$$\langle transform \rangle \quad \Box(\quad (In(t) \equiv \neg Out(po)) \wedge (Out(t) \equiv In(p))) \quad \Diamond(In(p)) \tag{VI.24}$$

The transition axioms with their statement labels are as follows:

$$\begin{array}{ll} \phi \wedge \Diamond at\varrho 1 & \\ at\varrho 1 \wedge (True \supset \Diamond[& \neg Out(t) \wedge In(p) \wedge \Diamond(Out(t) \wedge \neg In(p) \wedge \\ & \Diamond(\ell(t) = \ell(p) \wedge at\varrho 2))]) \\ at\varrho 2 \wedge (True \supset \Diamond[& \neg Out(p_g) \wedge In(t) \wedge \Diamond(Out(p_g) \wedge \neg In(t) \wedge \\ & \Diamond(\ell(p_g) = \ell(t) \wedge at\varrho 1))]) \end{array} \tag{VI.25}$$

Notice that the third axiom embodies the implicit *infinite loop* of a schema behavior section in $\mathcal{RS}$(Definition 45, Item 2). We shall prove the 'eventually' clause first:

1. $\phi \supset \Diamond at\varrho 1$.

2. $at\varrho 2 \supset \Diamond at\varrho 1$.

3. $at\varrho 1 \supset \Diamond In(p) \wedge \Diamond at\varrho 2$.

Thus, from any statement, we get $\Diamond In(p)$; hence, it is an assertion for the schema.

We use the invariance principle to prove the inside of the always clause; let us refer to it as Q. We shall prove it holds at ϕ, and that, if true before each transition axiom, then it is true after each axiom has been applied. This allows us to say $\Box Q$ is an assertion for the schema.

1. Initially: $\phi \supset In(t) \wedge Out(t) \wedge \neg In(p) \wedge \neg Out(p_g)$, hence, the two equivalence clauses hold.

2. Assuming $Q \wedge at\varrho 1$, we show $\Diamond(Q \wedge at\varrho 2)$: First, $Out(p_g)$ and $In(t)$ are unchanged by this transition axiom, hence they hold. By inspection of the transition axiom, it can be seen that $In(p)$ and $Out(t)$ bear the relationship $In(p) = \neg Out(t)$ in all three clauses[6]. Therefore, we get $\Diamond(at\varrho 2 \wedge In(p) = \neg Out(t))$.

3. Assuming $Q \wedge at\varrho 2$, we show $\Diamond(Q \wedge at\varrho 1)$: First, $In(p)$ and $Out(t)$ are unchanged by this axiom, hence if they were true initially they still hold at the end. By inspection of the transition axiom, it can be seen that $Out(p_g)$ and $In(t)$ bear the relationship $In(t) = \neg Out(p_g)$ throughout. Therefore, we get $\Diamond(at\varrho 2 \wedge In(t) = \neg Out(p_g))$.

§6. Task-Unit Assertions

Since a task-unit has distinct components, we can associate assertion template outlines with each of the components of the task-unit. A precondition will typically test some condition of ports $C(p)$, and create some task-unit T, then, possibly, deinstantiate itself:

$$\langle pre \rangle \quad \Box(C(p) \supset \Diamond(INST * (T) \wedge \neg INST(pre))) \tag{VI.26}$$

Perceptual schemas, in general, perform some processing of sensory input, and/or input to the task-unit. They will thus relate some world-descriptive predicate to their output port values:

$$\langle P \rangle \quad \Box(worldpred \supset Port - Assertion) \tag{VI.27}$$

In some cases, the perceptual schema may simply have the form of a buffer, transforming its input according to some function $F(input)$ and passing it on to its output:

$$\langle P_{buff} \rangle \quad \Box((In(p) \equiv \neg Out(op) \equiv (|\, p \,| = |\, op \,|)) \wedge (\ell(op) = F(\ell(p)))) \tag{VI.28}$$

The motor schemas relate port assertions to world-descriptive predicates. For example, the abbreviation predicate *PosControl* is an example of a motor schema assertion:

$$\langle M \rangle \quad \Box(Port - Assertion \supset worldpred) \tag{VI.29}$$

[6]Refer back to the definition of the assignment transition axiom for a description of the three clauses.

We now have all the material in place to do some extensive program development in $\mathcal{RS}$. Chapter 8 will be a single long example in using all the material we have developed to implement the grasping and manipulation of Chapter 2 as schema programs.

§7. Limitations of the Verification System

Having constructed our temporal logic design and verification methods for the $\mathcal{RS}$ model, we now evaluate them.

Typically, safety properties ("what will not happen") are well represented in trace models; however, such models usually have more problems in expressing liveness properties. Nguyen et al. [1984]'s introduction of the ***observation*** as a generalization of the trace mechanism improves the ability to represent such liveness properties. Within this framework, we can represent anything which can be specified in linear-time temporal logic.

There are some limitations to this verification model. From the point of view of robotics, we have no means of connecting SI behavior to real-time deadlines. Although we can formulate the specification of a servo process in terms of what will ***eventually*** happen, we cannot determine when this will happen. In the case of a servo process we may want to put time bounds on the response of the system. In the case of searching for some object, we may want to give up (and ask for help) after some time has elapsed. There is no simple way to express these constraints in our logic as it stands.

From a more general point of view, we have side-stepped the problem of network deadlock and termination. Our two liveness axioms have prohibited exploration of the deadlock problem. We cannot examine the case when some circular deadlock condition has developed because we have forbidden such a case to occur. This may be more a problem with the liveness axioms than with our temporal logic. We also cannot examine the case of a network which terminates when all its component SIs have neither In nor Out asserted for any port. Indeed, our use of $\Diamond In(p)$ in many schema specifications rules out applying this termination case. However, we note that in a recent paper, Nguyen et al. [1986] extend the framework of Nguyen et al. [1984] to deal with issues of deadlock and termination.

CHAPTER VII

REPRESENTATIONAL ISSUES

There are certain standard programming methods and data types which occur in almost any form of computing. In this chapter, we look at *RS* implementations for a few of the most common of these structures. We start by investigating the form of *recursive* and *iterative* computation in *RS*; we had a preview of this with the `Factorial` schema in Chapter 4. In Section 2, we implement some common data structures. In *RS*, however, data structures lose their 'passive' connotations, since they are implemented 'actively' as networks of SIs. Sets, stacks, pipes/buffers and arrays are the data structures presented in detail.

In Section 3, we implement some AI programming representations in *RS*. AI and Robotics both have the same goal of getting a machine to exhibit intelligent behavior comparable to a human. They differ in that AI usually contents itself with abstract reasoning behavior, while Robotics concerns itself with physical behavior. It can be argued that these are simply two sides of the same coin: Abstract reasoning can be seen as a form of 'internalized' physical behavior; and physical behavior can be considered the 'end result' of abstract reasoning. Since *RS* has been developed in a Robotics context, it will be interesting to see how it handles AI representations typically used in abstract reasoning.

§1. Recursion and Iteration

An informal definition of recursive programming is that of a programming style in which some semantic or syntactic unit constructs a number copies of itself, each solving progressively 'smaller' subsections of the main problem, until eventually some unit is handling the trivial or basis case. The response of each unit is phrased in terms of the response of the unit it has created. The basis case, then, is the *only* case for which

the recursive unit can return a 'straight' answer, a response not phrased in terms of the creation of any other units. The **Factorial** schema of Chapter 5, when given a input x, creates a single copy of itself, and feeds it the input $x-1$. This process continues for each successive instantiation of **Factorial**, until one instantiation is fed the value 1 as input — this is the basis case for the factorial function. The response for the basis case of the factorial function is defined to be 1. This response allows each instantiation of **Factorial** in turn, to resolve its response.

A simple recursive schema is the **Sumfromzero** schema, which implements the following function:

$$SZ(x) = \begin{cases} 0 & if\ x=0 \\ x + SZ(x-1) & else \end{cases}$$

We shall define the **Sumfromzero** schema as follows:

```
[  Sumfromzero
   Input-Port-List:    ( x:Integer Pszx:Integer)
   Output-Port-List:   ( Szx:Integer Px:Integer )
   Variable-List:      ( temp:Integer )
   Behavior:           ( temp:=x;
                        If (x=0) Then Szx:=0;
                         Else Sumfromzero(Px)(Pszx)();
                                Px:=temp-1;
                                szx:=temp+Pszx;
                       Endif;
                       Stop;                             )  ]
```

By analogy with the factorial example: **Sumfromzero** 'continues' its computation recursively whenever it is fed a value in its port x which is greater than zero. The ports Px and $Pszx$ are used to transmit the 'continued' value $(x-1)$, and to receive back the partial answer. The instantiation instruction has connected these ports to the x and szx ports respectively, on the child SI.

Recall from the definition of instantiation in Chapter 5, Definition 48, that we treat the semantics of instantiation by considering how the newly created instantiation changes the semantics of the network of SIs in which it is involved. This is particularly interesting in the case of the recursive instantiation in **Sumfromzero**. In each case, the network consists of the parent **Sumfromzero** SI, and the network created by (and including) the child SI. Let us consider this network as an assemblage SI. We can implement this by instantiating the following assemblage schema **Asumfromzero**, instead of **Sumfromzero**:

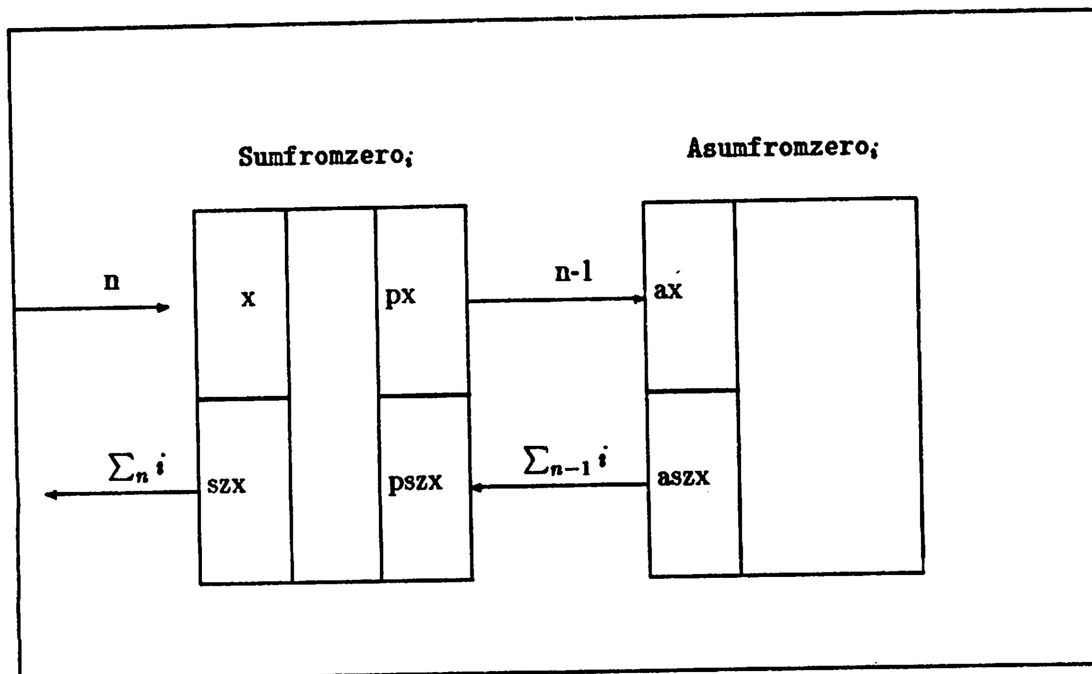


Figure 24: The *ith* Level of Recursion of Sumfromzero.

```
[  Asumfromzero ASSEMBLAGE
   Input-Port-List:            ( ax:Integer )
   Output-Port-List:           ( aszx:Integer )
   Component-Schemas:          ( Sumfromzero Asumfromzero )
   Initialization:             ( Sumfromzero(≡ax) (≡aszx )(); )  ]
```

This assemblage schema makes the recursion more indirect but does not change the way SZ is computed. Note that, now, the semantics of the recursion is exactly that of the definition of the network automaton from Chapter 5. Let P_i be the semantics of the *ith* instantiation of Sumfromzero, and let PP_i be the semantics of the *ith* instantiation of the assemblage Asumfromzero. At each level of the recursion, the semantics of the network of the parent SI and the child assemblage are connected by the map C : $C(px) = ax \;\; C(aszx) = pszx$, e.g., Figure 24; this is described by:

$$P_i \|_c PP_i$$

for the *ith* level of recursion. However, at the $ith - 1$ level of recursion, this, in turn, is an assemblage:

$$PP_{i-1} = P_i \|_c PP_i$$

The above relationship between the assemblage at recursion level $i - 1$ and the network of SIs at level i, is a general description of the semantics of recursion in $\mathcal{RS}$.

Iterative programming is usually defined as occurring when a program iterates through sub-problems over the same body of code in order to solve some problem; as opposed to recursion, where a new copy of the body of code is created for each sub-problem. There are no explicit repetitive constructs in $\mathcal{RS}$ similar to the **while**, **repeat** or **do** loops of many general-purpose programming languages. However, in server-process style, the behavior section of each schema instantiation repeats indefinitely until the SI deinstantiates — an implicit loop. An `If` statement can be used to implement all the standard iterative constructs. Consider the following iterative implementation of the SZ function in Pascal:

```
a:=N;   s:=0;
while (a≠0) do
  begin
  s:=s+a;   a:=a-1;
  end;
```

where a and s are integer numbers, and $a > 0$. It can easily be verified that this loop obeys the invariant:

$$\sum_{i=N} i = \sum_{i=a} i + s$$

Termination can be argued informally by noting that, for an integer $a > 0$, if we subtract 1 from a, then $a - 1 < a$, and thus, eventually, we shall have $a = 0$. And when $a = 0$, the above invariant yields the fact that $\sum_{i=N} i = s$, the desired result. This can be implemented by a schema `Isumfromzero`:

```
[  Isumfromzero
   Input-Port-List:    ( )
   Output-Port-List:   ( szx:Integer )
   Variable-List:      (a:Integer s:Integer )
   Behavior:           ( If (a=0) then szx:=s; Stop;
                         Else s:=s+a;    a:=a-1;
                         Endif;   )                      ]
```

where, now, the values of a and s must be initialized by the SI instantiating this SI. It is easy to see that this schema obeys the same invariant as the **while** loop above.

§2. Some Common Data Structures

This section describes $\mathcal{RS}$ implementations of the following data structures: sets, stacks, pipes/buffers, and arrays. The implementations shown here are not the only

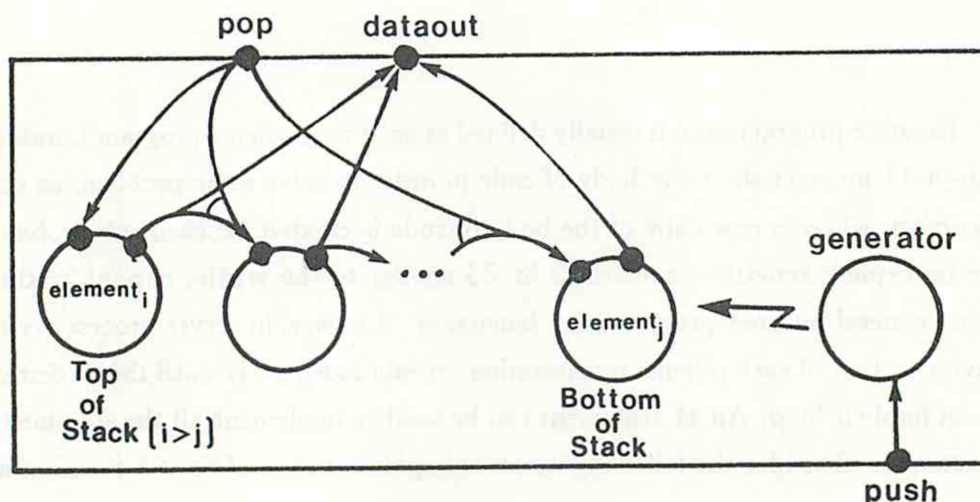


Figure 25: Internals of stack.

possible ones. But, these particular implementations were designed to correspond with the style of processing in *RS*. The one major point in which the *RS* implementation differs from the standard view of these data structures, is that they are all implemented as 'process networks', networks of SIs with little, if any, passive data. A basic advantage of this is that a lot of the *parallelism* inherent in data types such as stacks and arrays can be brought out.

Stack: Two operations can be performed on a stack; the ***push*** operation gives a data value to the stack, a *pop* operation retrieves a data item from the stack. A stack is a LIFO[1] queue; the item retrieved by a *pop* will always be the *last* item fed to the stack with the *push* operation, which has not yet been retrieved from the stack. We implement a stack as an assemblage `stack` with three ports, *pop*, ***push***, and ***dataout***. A write to ***push*** will place that value on the stack; a write to *pop* will result in the top value in the stack becoming available at *dataout*.

Internally, `stack` consists of a number of `element` SIs; one for each element which has been written, so far, to the stack. Every time a value is written to the ***push*** port of `stack`, a new instantiation of the `element` SI is created to represent this data value. When the *pop* port is read, an appropriate instantiation `element` replies with its stored value. It is necessary to ensure that the `element` SIs reply in the correct LIFO order, however. Figure 25 illustrates the connections to implement this ordering (AND- versus OR-semantics being an *very* important difference in this example).

The `element` schema is defined as follows:

[1]Last-In, First-Out.

```
[  element
   Input-Port-List:    ()
   Output-Port-List:   ( data:Real seq:Real )
   Variable-List:      ( store:Real temp:Real )
   Behavior:           ( temp:=seq;
                         data:=store;   )        ]
```

When an instance of element is made, the variable *store* is initialized to hold the data value to be stored. The port *data* is connected to the output port *dataout* of the stack assemblage (Figure 25), and the port *seq* is connected to the *pop* port of stack. Thus, if any value is written to *pop*, then the data value immediately becomes available at *dataout*. In addition, the *data* port of the top element is AND-semantics fan-out connected to the *seq* port *of the previously created instantiation of* element. The fan-in on *seq* ports is also defined to have AND-semantics. Thus, once an element is *underneath* the top of the stack, its read to its *seq* port can only terminate when both *pop* is written *and* the SI representing the value on the stack immediately above this one has written its *data* port (i.e., it has been popped from the stack). Note that *temp* is used within element for the LIFO synchronization only; its actual contents are never used.

The schema which creates element SIs we shall call **generator**. Consider, now, what *underflow* and *overflow* mean in this representation. Underflow occurs when there is no element SI to write to the *pop* port; hence, the reader will suspend until something has been put on the stack[2]. Overflow has no meaning for this representation, unless some explicit upper limit on the number of element SIs is built into generator. Of course, in implementation terms, there cannot be an infinite number of SIs — but, as it stands, there is no limitation on the stack size in the structure of the stack assemblage.

Set: A set is a list of elements with some property in common. We have already had to deal with this concept in representing virtual fingers, where we had a set, $VFset_k$, which represented the finger numbers of those physical fingers considered to be in virtual finger k for some task. There are three components in the representation of sets in $\mathcal{RS}$: the set element schema, the set generator schema, and the set index schema.

There is one instance of the element schema for each element in the set. The generator schema, either initially, or dynamically, sets up appropriate instantiations of the element schema -- hence, it must embody the rule which describes the set. The index schema allows one to determine if a particular element is in the set, once the set has been

[2]Unless it uses a time-out SI, as described in Chapter 5.

initialized.

We shall use the schema sel as the set element schema. sel is the null schema — no ports and an empty behavior section. Its only purpose is to be instantiated and, thus, represent a set element *by its instantiation number*. Based on some internal criteria, the set generator will determine what instantiations of sel to make. For example, the set generator schema primes will construct one set element for all the primes between 20 and 1 (in a brute-force manner).

```
[  Primeset
   Input-Port-List:    ()
   Output-Port-List:   ()
   Variable-List:      ( n:Integer j:Integer noprime:Integer )
   Initialization:     ( For n:=20...1
                           noprime:=0;
                           For j:=n-1...2
                             If (n mod j)=0 then noprime:=1;
                             Endif;
                           Endfor;
                           If (noprime=0) then sel_n()()();
                           Endif;
                         Endfor;
                         Stop;     )                         ]
```

For any prime number found by this algorithm, an instantiation of sel with that instantiation number is created. The next part of the set machinery is the set index schema. There are two possible operations we might want to perform on a set SS: to do some action N for all elements in that set $\forall k \in SS \quad N(k)$; and to determine if a particular element is in that set, $k \in SS$.

The basis of the index schema is the statement Forall $\texttt{sel}_i$: ... Endforall; this allows some SI to be created for each element of the set. If an action such as 'do N for all elements in the set' is being performed, then the body of the Forall in the index schema must create a schema which carries out whatever N describes for each instance of sel. As, for example, when some finger movement is carried out for each member of a virtual finger.

If the desired action is to determine if a particular element k is in the set, then the SI created in the Forall statement needs to create an SI for each element of the set, and the purpose of that SI is to check itself to see if its element is the element k — this is the same mechanism we used in the Obj? precondition, in Chapter 4, to search all EOB SIs for one corresponding to a particular target object.

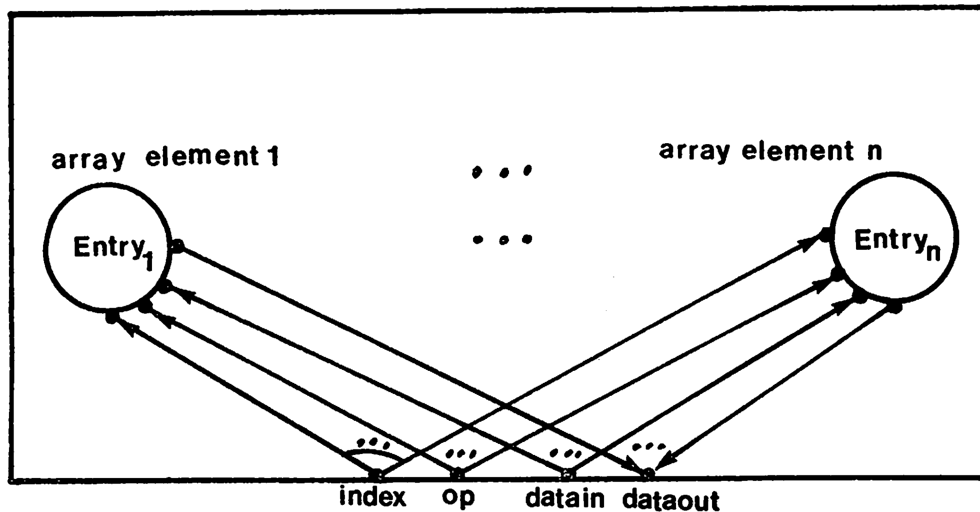


Figure 26: Internals of array.

Array: An array is an indexed set of data locations. The operations which can be performed on the array, are to write a value to a particular location, and to read a value from a particular location. We shall implement an array with the assemblage schema **array**. Internally, **array** consists of a number of **entry** SIs, one for each entry that the array is supposed to have. For simplicity, we shall consider a linear array (a vector); higher dimensions can be constructed by an array schema in which each entry is, in turn, an array schema; or, more simply, by using some mapping function which takes two or more indices and maps them onto a unique single index in the conventional manner.

The assemblage **array** has a port *index* into which the index of the entry to be addressed is written. Internally, this port is AND-semantics, fan-out connected to the *index* port on each **entry** SI. Whenever a value is written to the *index* port, all **entry** SIs read it in parallel. Each **entry** SI stores its own index internally; this value can be given to the SI as an initialization parameter — an alternative implementation to using the instantiation number as the SI's index, though this, too, is possible. When an **entry** SI reads a value from its *index* port, it checks this value against its own index; if they do not agree, it just goes back to reading *index* again; however, if they do agree, then it knows that it is the target of the next data storage or retrieval operation.

This selection mechanism is shown in Figure 26. For the actual data operations, each **entry** SI has three ports - an *op* port by which it knows whether a read or write operation will be performed, and *datain* and *dataout* ports. Additionally, it needs an internal variable *store* in which it keeps the value of its data entry (if it has one) stored. These ports also have fan-in (*op* and *datain*) and fan-out (*dataout*) connections to ports with identical names on the **array** assemblage (Figure 26); although, for these connections

we need *OR-semantics*, since *only one* entry *SI will respond to messages on them.*

Pipe/Buffer: A pipe is a process which accepts data from one producer, and passes it on, possibly with a delay, to some consumer. A pipe usually acts as a communication link between two processes. From one point of view, a connection between two ports in $\mathcal{RS}$ is a pipe. However, no data is stored in a $\mathcal{RS}$ connection, while the notion of a pipe usually embodies the notion of buffer, that data can be cached temporarily between the two processes.

In $\mathcal{RS}$ we can implement this as a schema, pipe, which reads values from an input port *a* and writes, subsequently, to its output port *b*. It is necessary that we implement *asynchronous* communications on both the input and output port — otherwise, using pipe would be no different from connecting the ports on the producer and consumer SIs together directly. As an internal data structure, pipe can use an array schema or stack schema such as we have already described.[3]

The time-out schema of Chapter 5 is the appropriate form of asynchronous communication in this case. pipe reads its input port *a*, and, if a non-null value arrives, it stores this; if a null value arrives it assumes that this came from the time-out SI, and this indicates that the producer SI has no data to send to the pipe. It, then, tries to pass on some data to the consumer SI (a write to the *b* port). Note that when this write terminates (and it will, since there is a time-out SI connected to the port), pipe has no way of knowing if the value was read by the time-out process or by the consumer. To rectify this we need to introduce an *acknowledgement* port *ack*, also with a time-out SI connected to it, by which the consumer verifies if it received the data value from pipe. Once the write to *b* has completed, pipe then reads the *ack* port. If the consumer SI fails to send an acknowledgement to this port before the time-out, then pipe assumes that it did not get the data value; otherwise, it recognizes that the data has been passed on and removes it from its internal store.

---

[3]It could also use a vector or matrix data type in the conventional fashion; but this is not the $\mathcal{RS}$ style.

§3. AI Representations

Certain knowledge representations have become common in AI. In this section we shall look at how $\mathcal{RS}$ might implement these representations. As before, the implementations presented here are not the only possible ones, but these were chosen to portray the style of the $\mathcal{RS}$ model. We look at three representations: frames, semantics nets and production systems [12]. We will find that all these function at a level of abstraction higher than we have been used to in $\mathcal{RS}$ so far.

Frames: Knowledge structures based on the idea of 'chunks' of knowledge, which provide context and expectations for dealing with experience, have been used in AI for some time. These have gone under names like frames[60], scripts[77] and schemas[5]. Since $\mathcal{RS}$ is based on Arbib's schema concept, which has many of the connotations of this sort of structure, we would expect $\mathcal{RS}$ to represent it well.

Typically, a frame denotes a skeletal description of some domain of interaction – the standard example is the restaurant frame. This frame has slots for describing the type of restaurant (Chinese, American, etc.), and for action routines which describe how to 'use' a restaurant (e.g., to order a meal first read the menu).

The structure in $\mathcal{RS}$ which determines how a robot interacts with the environment is the task-unit. Internally, this consists of an object model, the perceptual SIs, and the basic actions in the task, the motor SIs. The slots on a frame can be considered equivalent to the ports on a perceptual SI in a task-unit. A 'frame' in $\mathcal{RS}$ would then be a network of perceptual SIs — an assemblage. However, this does not take into account the action routines which can be associated with slots in a frame. The use of a frame to represent some object or location is really only 'half the story'; it must also be possible to gain access to the actions available on this object. The $\mathcal{RS}$ task-unit represents a task, and its perceptual SIs represent a task view of the object; the motor SIs of the task-unit are the actions which it is possible to perform on the object *in the context of this task*.

In summary, the notion of a frame is easily constructed in $\mathcal{RS}$; frames which are just used to represent objects, are essentially represented by a network of SIs, with the ports on the SIs corresponding to the slots in the frame. We make the point that a frame representing an object, or location, or situation, should somehow provide pointers to appropriate actions. In this case, the $\mathcal{RS}$ task-unit best represents a frame – its perceptual SIs being the object model, and its motor SIs being the actions appropriate to this object.

Semantic Networks: Semantic networks are a highly diverse graphic knowledge representation, in which situations, events, objects, relationships, etc., are represented by nodes and arcs in a graph. Typically, the nodes represent objects, concepts or situations, and the arcs represent relationships between them.

For example, if we want to model the fact that elephants are mammals, we build a semantic network which has two nodes, **mammals** and **elephants**, joined by an arc labelled **isa**. If we want to expand this to indicate that elephants have trunks, we add a new node, **trunk**, and join it to **elephant**, with an arc labelled **body-part**.

The main use of a semantic network is to hold knowledge in such a way that it can be easily reasoned about. One of the first reasoning methods used in semantic nets was called *spreading activation*. It worked as follows: To make inferences about a pair of concepts, you start from the node representing each concept and send a sphere of 'spreading activation' through the semantic network. When the wave-fronts from the two sources meet, then a path has been established through the semantic network connecting the two concepts.

A second method of reasoning with semantic nets consists of constructing a 'template' network which corresponds to the information requested; and to try to 'fit' this template to some portion of the semantic network. For example, in our elephant example above, if we wanted to ask the network the question "who has trunks", we would do it as follows: First, we would construct the network fragment with two nodes **who?** and **trunk** connected by an arc labelled **body-part**. We would then try to match this against our network, where nodes and arcs will match if they are equal, and the **who?** node matches any node. Once we have a match, the node to which **who?** is matched is the answer to the question.

We can implement a semantic net in $\mathcal{RS}$ as follows: A schema represents a concept. To put that concept in the network, an instantiation (possibly parameterized) is made. The ports of the schema are the relationships in which an instantiation of that schema can engage. When instantiated these relation-ports are appropriately connected to the ports on other SIs in the network. A message over a port is then equivalent to 'activating' that arc. Reasoning mechanisms are embodied in the schemas.

The spreading activation method is the easiest to implement. The two source nodes begin the 'sphere of activation' by sending out unique messages (markers) on their relation-ports. The first node to get two different markers is that node which represents

a concept common to both source concepts. Note that multiple, simultaneous spreading activations can occur.

We can nest/partition semantic networks by using the assemblage construct. The ports on the resulting assemblage 'node' represent the relationship between this sub-network and the rest of the network. Partitioning the network into assemblages limits (focuses-in) the scope of any reasoning operation. The *RS* implementation of semantic networks is limited by the hierarchical structure of the assemblage - it is difficult to partition a network dynamically (since any assemblage is a fixed group of SIs).

Note that if we combine task-units into the semantic network structure, it is possible to directly associate actions and procedural 'know-how' with concepts. Taken in this vein, the example program we shall see in Chapter 8 is actually a semantic network containing only procedural knowledge.

Production Systems: A production system consists of a set of rules, a current context, and a rule interpreter. A production rule is essentially an IF-THEN statement expressing an action which is appropriate under some conditions, as given in the condition section of the rule. The current context represents the short term knowledge[4] on which the system is working. It is to the contents of this knowledge base that the rule interpreter applies the condition section of each IF-THEN rule. Actions taken by any rule which fires, that is, whose activation section is true in the current context, may add new data to the current context. Production systems in which the rules embody *expert knowledge*, so called Expert Systems, have become very popular in AI.

The precondition/task-unit structure in *RS* is quite similar to an IF-THEN rule. The precondition specifies some condition under which the task-unit is to be instantiated — this can be a function of direct sensory data, or of communications with other SIs. Note, that once created, a task-unit is more than just an action, as with an IF-THEN rule. The task-unit actively filters incoming data through its perceptual SIs. Additionally, the assemblage structure is a tool for grouping 'rules' into contextual chunks. When instaniated, a task-unit may contain a number of other task-units with associated preconditions, representing actions which are appropiate in the context of that task-unit. Some of the preconditions may represent stages of a task to be performed (as in our Chapter 8), some may watch our for error situations.

---

[4]The rules are the long-term memory.

CHAPTER VIII

IMPLEMENTATION OF THE SSG SYSTEM

In order to demonstrate the expressive power of $\mathcal{RS}$ in the robot domain, we shall go through the programming necessary to implement the grasping and manipulation system described in Chapter 2. We start by reviewing the assembly task from Chapter 2; in particular, a high-level assembly program is constructed for this task. Based on this program we can construct the schemas in $\mathcal{RS}$ to carry out the assembly task. Firstly, the general format of a robot program in $\mathcal{RS}$ is discussed. From this description of the overall task, we focus in, in Section 2.1, on the implementation of the *grasping* operations. Schema assertions are used in conjunction with the task-unit notation of Chapter 4 to design the overall structure of the implementation. The full schema descriptions and proof outlines for the *reach* phase of the grasp operation are given in Section 2.2. The full schema descriptions and proof outlines for some of the other principal schemas are presented in Appendix A.

§1. Example Assembly Program

In Chapter 2, Section 2.1, an example assembly task is introduced; on page 11 a list of assembly instructions for the task is presented. This list of human-oriented instructions must be translated into a more mechanical form before we can start to consider its implementation in $\mathcal{RS}$. As an item of simplification we shall omit the operations using the small screws (securing the Transformer and Capacitor). We construct a task-level robot programming language to express the assembly task; the language syntax provides a vehicle for representing the concepts expressed in Chapter 2.

§*1.1* *The Example Task*

Our language (let us call it *RS*-TL1[1] for convenience) is not meant to be a serious attempt at a task-level language, but, rather, a tool to describe the assembly task in the appropriate terms of Chapter 2. *RS*-TL1 has only two sorts of commands: *manipulation* commands and *grasping* commands. The manipulation commands instruct the robot to perform some manipulation with the grasped object. Each manipulation command describes a class of manipulation operations (as discussed in Chapter 2). The grasping commands prepare the robot to manipulate an object in some task, by acquiring the object in some appropriate grip position. The main feature of the grasping command is the `in-order-to` clause, which tells the grasp selection mechanism the main operation (manipulation command) that the grasp will be used to perform. The possible operations (and manipulation commands) are: `move`, a free motion of the hand and object; `place`, a guarded move of the hand and object to some surface; `insert`, the peg-in-hole insertion task; and `screw`, turning the head of a screw or bolt in order to secure it. We construct *object and location modifiers*, such as `in`, `on` and `of` as necessary to describe component and location relationships. Table 3 is the assembly instructions of Chapter 2 reformatted in the outlined language, *RS*-TL1. It is assumed that objects such as BASE-PLATE, TOP-PLATE, etc. are predefined, as are the *relative* positions such as TRANSFORMER-SITE, LEFT-EDGE, HOLE1, etc.

§*1.2* *Sequencing of Robot Programs*

The next step is to consider how this program is represented in *RS*; or if you like, how *RS* provides the semantics for our simple task-level robot programming language. The list of robot commands in Table 3 does *not* have the serial order typically associated with robot programs. The task-unit created by a grasp command may need to persist through several manipulation commands, since it embodies the model of the hand and object. However, manipulations are serially ordered with respect to each other; for a particular grasp, no manipulation command can begin until the grasp is ready, and the previous manipulation command (if any) has finished. The serial nature of manipulations is due to the fact that each manipulation needs sole use of the hand as a resource; hence, no two can use it simultaneously.

---

[1]For Task Language 1.

```
grasp     BASE-PLATE   in-order-to  place-coarsely
place     BASE-PLATE   at           CENTER-OF-WORKSPACE
release
grasp     CAPACITOR    in-order-to  insert-coarsely
move      CAPACITOR    over         ( HOLE in BASE-PLATE )
insert    CAPACITOR    in-to        ( HOLE in BASE-PLATE)
                                    a-distance-of 3cm†
release
grasp     TRANSFORMER  in-order-to  place-precisely
place     TRANSFORMER  at           ( TRANSFORMER-SITE on BASE-PLATE )
release
grasp     SIDE-PLATE1  in-order-to  place-precisely
place     SIDE-PLATE1  at           ( LEFT-EDGE of BASE-PLATE )
release
grasp     SIDE-PLATE2  in-order-to  place-precisely
place     SIDE-PLATE2  at           ( RIGHT-EDGE of BASE-PLATE )
release
grasp     TOP-PLATE    in-order-to  place-precisely
move      TOP-PLATE    over         CENTER-OF-WORKSPACE
place     TOP-PLATE    on           (TOP of SIDE-PLATEs)
release
for       j=1 to 4     do
 grasp    BOLT-j       in-order-to  insert-precisely
 move     BOLT-j       over         ( HOLE-j in TOP-PLATE)
 insert   BOLT-j       in-to        ( HOLE-j in BASE-PLATE )
                                    a-distance-of 10cm†
 screw    BOLT-j       to-torque    0.1Nm†
 release
endfor
```

Table 3: $\mathcal{RS}$-TL1 Assembly Program for the Example Assembly Task.

†These are typical numerical values for these operations.

Both the manipulation commands and the grasp command are represented by task-units in $\mathcal{RS}$. We provide all grasp and manipulation task-units with an input port *begin* and output port *end*, used to implement the ordering discussed above, as follows: Whenever a manipulation or grasp task-unit is instantiated, it first reads its port `begin` before starting any action; it writes a value to its port `end` as its last action before it deinstantiates. Of course, the ports *startup* and *closedown*, discusssed under similar context in Chapters 5 and 6, could also be used here. Additionally, the grasp task-unit has an output port `grasp-ready`, which indicates the object has been successfully acquired, and can now be manipulated. These ports can be used to achieve the desired overlap between the grasp and manipulation task-units, and the sequential order of the manipulation task-units.

Having discussed the format of a robot program in $\mathcal{RS}$, we shall now go into more detail on the implementation of the grasp and manipulation task-units.

§2. The SSG System

In this section, we shall use schema assertion templates and the task-unit shorthand, (developed in Section 1 of Chapter 4), to design the programs for the SSG system. We pursue a top-down design of the implementation, making frequent use of abbreviation predicates. This is followed by a bottom-up construction of the full schema specifications for some of the task-units in the design, demonstrating how the abbreviation predicates can be shown to correspond to schema assertion templates in this detailed construction phase. Appendix A contains full schema descriptions for some of the other principal task-units in the implementation. The shorthand notation allows us to specify as little of the schema description of the task-unit as is necessary to agree with the proposed assertion template. Before proceeding with the design, we summarize the main points of the shorthand notation of Chapter 4, and the assertion methodology of Chapter 6.

Some set of SIs connected together by a connection map c is denoted $\|_c(A_i, B_j, \ldots)$; an assemblage of these SIs is denoted by enclosing the list in square brackets. A schema/SI name can vary from its simplest (and hence, least constrained, in design terms) form, A_i, to its most complex (and most constrained) form:

$$A_i(a1, a2, \ldots)(b1, b2, \ldots)(v1, v2, \ldots)$$

where the a's are input port names, the b's output port names, and the v's are internal variable names. The rule we follow to determine how complex to make the name is: Only ports and variables necessary to describe how the schema/SI fulfills some assertion are included. If these ports and variables occur in the shorthand form of the schema, then they have to occur in the full schema description; however, the full schema description may also have other ports and variables not mentioned in the shorthand.

A task-unit is a special assemblage denoted by: $[(P_1, P_2, \ldots) \overset{c}{\longleftrightarrow} (M_1, M_2, \ldots)]$, where $P_1, P_2, \ldots$ is a list of perceptual SIs, the sensing actions or object models for the robot task which the task-unit implements; $M_1, M_2, \ldots$ are the actions in the task; and c is a connection map which defines the connection between perception and action. We can optionally associate a *precondition schema* with a task-unit, which we denote by: $pre : T$, where pre is the precondition schema, and T is the task-unit with which it is associated. The function of the precondition is to make an instantiation of the task-unit when some condition is satisfied.

With each task-unit, T, we associate an assertion template, $\mathcal{A}$, which describes how that task-unit behaves. We denote this by: $\langle T \rangle\ \mathcal{A}$, where $\mathcal{A}$ is an assertion in the temporal logic system described in Chapter 6. The assertion will usually relate *world-descriptive predicates*, whose values reflect the current state of the external world, to assertions about the *behavior* (as defined in Chapter 6) of the task-unit. We may choose to summarize assertions on schema behavior into *abbreviation predicates*, whose parameters (when specified) are the port names which would be featured in the full assertion. Occasonally, we will combine world-descriptive and abbreviation predicates into an abbreviation predicate which may include an object name, or joint or link number, as a parameter.

It is important to remember that the purpose of the shorthand and assertions are to get to a full schema description for each of the task-units in the implementation, *along* with some formal concept of how the task-units behave. The full schema description, as described in Chapter 4, is what would be fed to an $\mathcal{RS}$ 'compiler'. Also, note that little work can be done on verification using only the shorthand; it is essentially a design tool. For verification, the full schema description is necessary. Hence, this chapter is divided into separate design (2.1) and verification (2.2) sections. *The goal, in the design section, is to come up with an outline of the task-unit or schema in our shorthand, and an assertion which describes its behavior. The goal, in the verification section, is to prove that some schema description obeys a given assertion.*

§2.1 *Design of the Implementation*

The basic goal of grasping is to acquire some particular object, j, in a fashion which allows for expected subsequent manipulation. We can readily sketch the assertion template we should like our basic grasp task-unit, **grasp$object**, to have:

$$\langle grasp\$object\rangle \quad WORLDOBJECT(j) \wedge Suitable(j) \supset \Diamond[(Acquired(j) \wedge ControlPreds(j) \wedge PrecRange \wedge AffixRange) \vee (Error)] \qquad \text{(VIII.1)}$$

where $WORLDOBJECT(j) \wedge Suitable(j)$ describes the condition: If there is an object in the world which meets the description of the target object, then we are interested in trying to grasp it. $WORLDOBJECT(j)$ is a world predicate, and $Suitable(j)$ is an abbreviation predicate of the kind which incorporates, internally, some world-descriptive predicates, hence it has an *object name* (j) as a parameter. It will, necessarily, have to be broken down before it can be implemented.

An error can occur at any stage of the grasp[2], we model this by saying that the result of the task-unit **grasp$object** will be either an error situation, indicated by the abbreviation predicate *Error*, or it will be the acquisition of the object. We shall define the *Error* predicate simply in terms of some *error port*, *er*, on **grasp$object** which produces a diagnostic error message:

$$Error \equiv [Out(er) \wedge \ell(er) = errornumber] \qquad \text{(VIII.2)}$$

We shall use this error port construction for errors all the way through this design and implementation. *Acquired* is an abbreviation predicate indicating the object has been acquired. *ControlPreds* is some abbreviation predicate(s) indicating what control the grasp can exert over the object; in general, we shall refer to such predicates as *control predicates*. *PrecRange* is an assertion describing how *precise* this control is, and *AffixRange* is an assertion saying how securely affixed the object is to the hand; in general, we shall refer to these as *range predicates* (they will be expressed as ranges of some value eventually).

Manipulation: This is then accomplished by some manipulation task-unit, say **manip**, being instantiated and connected *appropriately* to **grasp$object**:

$$\|_c(manip, grasp\$object) \qquad \text{(VIII.3)}$$

[2] Chapter 2 provides an account of the errors which the SSG system recognizes.

The behavior of `manip` is specified by some assertion which details the manipulation requirements of the task `manip` implements. An example of such a manipulation schema is the task-unit `Insert`, which implements the Insert operation, and which has the assertion template:

$$\langle insert \rangle\ PosControl(j,m,\delta) \wedge StiffControl(j,n) \wedge (0 \leq \delta \leq K) \supset \Diamond(OBJPOS(j,d)) \quad \text{(VIII.4)}$$

where $StiffControl$ is a control predicate indicating that port n has stiffness control over object j, and $PosControl$ is another control predicate indicating port m exerts position control on object j. We introduce precision in position control as a range of δ (where K is the most positional imprecision that the insert task can stand); this is an example of the $PrecRange$ range predicate. The position d is the final desired position of the inserted object. Control predicates will translate to port assertions of the form:

$$PosControl(j,m,\delta) \equiv \Box(\ \Diamond In(m) \wedge (OBJPOS(j,m,\delta) \equiv In(m)) \\ \wedge(\mid m \mid \ = 0 \supset (In(m) \wedge OBJPOS(j,p_0,\delta))) \ ) \quad \text{(VIII.5)}$$

where $OBJPOS(j,m,\delta)$ is the world-descriptive predicate which is true when object j is at the position given by $\ell(m)$. $PosControl$ sums up the notion that position control of the object is possible over port m. In summary, the insertion assertion predicate says that if the object can be controlled appropriately and with the correct precision (on the indicated ports) then, eventually, the insertion task will complete. When a manipulation task-unit is connected to a grasp task-unit, the ports must be mapped accordingly (so that the ports in the control predicates 'line up'), and the appropriate control predicates and precision and affixment ranges must be asserted by the grasp.

The Internal Structure of `grasp$object`: Referring to assertion VIII.1, we associate a precondition `locate$object` with the grasping task, whose job is to locate the EOB which represents the desired object and connect the grasping task-unit, **grasp**, to it. The assertion template for this precondition schema is:

$$\langle locate\$object \rangle\ \ WORLDOBJECT(j) \wedge Suitable(j) \supset \\ \Diamond(INST*(grasp) \wedge \neg INST(locate\$object) \\ \wedge OBJPOS(j,g1) \wedge OBJANG(j,g2) \wedge OBJFEA(j,g3)) \quad \text{(VIII.6)}$$

where $g1, g2, g3$ are ports of the $grasp$ task-unit. The *antecedent* here reflects the assertion template for the EOB primitive schema, Assertion VI.8 of Chapter 6, and $Suitable(j)$ describes the concept of testing each object to see if it is the target object. Let us construct `locate$object` with 3 vector input ports, f_1, f_2, f_3, to which are written the target

object's position, orientation and feature values ***in the same order and meaning*** as for the EOB (Definitions 15) or which contain the trivial value, #. For example, if f_1 contains the trivial value, then this denotes that any object, j, with characteristics which agree with f_2 and f_3 and is ***in any position*** satisfies $Suitable(j)$, etc. We can now break down the abbreviation predicate $Suitable(j)$:

$$
\begin{aligned}
Suitable(j) \equiv\ & (OBJPOS(j, f_1) \vee \ell(f_1) = \#) \wedge (OBJANG(j, f_2) \vee \ell(f_2) = \#) \\
& \wedge (OBJFEA(j, f_3) \vee \ell(f_3) = \#)
\end{aligned}
\qquad \text{(VIII.7)}
$$

in which case, instantiating **grasp** and connecting to the proper instantiation of EOB will fulfill the consequent of assertion VIII.6. Now that we have decided on an assertion for the precondition schema which sets up the **grasp** at an appropriate time and connects it appropriately, we can start to structure the **grasp** task-unit internally.

Equation VIII.1 insists that **grasp** must assert the *Acquired* predicate, indicating the object has been acquired; it must also assert the necessary control range predicates to complete the task. The process of providing correct predicates for the task is the *grasp-selection process* in the SSG system; the input being object characteristics and the precision and affixment binary variables. We structure the **grasp$object** task-unit as:

$$
\begin{aligned}
grasp\$object &= locate\$object(f_1, f_2, f_3) : grasp(g1, g2, g3) \\
grasp &= [object()(sm, fl, lo) \overset{c}{\longleftrightarrow} grasp\$select(sm, fl, lo)() : grasp^g] \quad g \in \{\mathcal{E}, \mathcal{L}, \mathcal{P}\}
\end{aligned}
\qquad \text{(VIII.8)}
$$

where we have included here an extended form of the schema names to include the ports f_1, f_2, f_3 on **locate$object**, the ports $g1, g2, g3$ on **grasp**,[3] and the ports sm, lo, fl which are output ports on **object** and input ports on **grasp$select**. In each case, we have included this port information since it is implicit in the assertion for the task-unit; for **locate$object** and **grasp** from Assertion VIII.7 and VIII.6 respectively. We have included the ports sm, fl, lo, since they will carry the essential object feature information by which grasp selection is carried out; VIII.9. We also specify part of the connection map c for the task-unit:

$$
\begin{aligned}
object(lo) &\rightarrow grasp\$select(lo) \\
object(sm) &\rightarrow grasp\$select(sm) \\
object(fl) &\rightarrow grasp\$select(fl)
\end{aligned}
$$

The full schema description of these task-units must have these ports at least.

[3]**grasp** and **grasp**^{g} will be used to denote different schemas.

Grasp Selection: When the EOB representing the target object is located, the task-unit **grasp** is instantiated and connected to it. Internally, **grasp** consists of the object model schema **object**, whose duty, for the moment, will simply be to pass the features of the target object on to **grasp$select**, which is the common precondition schema for all three $\textbf{grasp}^g$, $i \in \mathcal{E}, \mathcal{L}, P$ task-units, each of which implements one of the Encompass, Lateral or Precision grasps of Chapter 2. This precondition schema is structured as a single schema evaluating the selection equations (II.9) of Chapter 2.

In this case, the assertion template is simply the selection equations reformatted. Let P and F be local variables to the grasp selection schema, initialized when the schema is instantiated to hold the precision and affixment requirements of the task. The ports sm, lo and fl are ports connected to the **object** schema through which will be written the characteristics $small, long, flat$ (of Chapter 2) of the object. In order to simplify the assertion for **grasp$select**, we define the following abbreviation predicates on its ports and variables. For notational relief, let us further assume that the port names denote the last value in the local variable sequence, i.e., $\ell(P), etc$, and that the value can be 1 or 0, interpreted as $true, false$ respectively.

$$\begin{aligned} ENC &\equiv P \wedge (sm \vee (\neg F \wedge lo)) \\ LAT &\equiv (\neg sm \wedge fl) \vee \neg((\neg P \oplus \neg F) \wedge lo) \vee (\neg P \wedge F \wedge lo \wedge fl) \\ PRE &\equiv (\neg sm \wedge fl) \vee (\neg P \wedge (sm \vee (F \wedge \neg fl))) \end{aligned} \tag{VIII.9}$$

These are simply the grasp selection equations of Chapter 2 phrased in terms of the ports and variables of **grasp$select**. We can then construct the grasp selection assertion as:

$$\begin{aligned} \langle grasp\$select \rangle \Box [\quad & (ENC \supset \Diamond(INST * (grasp^{\mathcal{E}}) \wedge \neg INST(grasp\$select))) \\ \vee \; & (LAT \supset \Diamond(INST * (grasp^{\mathcal{L}}) \wedge \neg INST(grasp\$select))) \\ \vee \; & (PRE \supset \Diamond(INST * (grasp^{P}) \wedge \neg INST(grasp\$select))) \,] \end{aligned} \tag{VIII.10}$$

There are three OR clauses, each of which are of the precondition form discussed at the end of Chapter 6. Depending on the state of its ports and variables (as given by the appreviation predicates), **grasp$select** will create one of the three grasp task-units.

Control Predicates: Each $\textbf{grasp}^g$ has a unique version of the grasp assertion consequent in assertion VIII.1; that is, some combination of $PrecRange$, $AffixRange$ and $ControlPreds$, which are unique to this grasp[4]. The abbreviation predicate $Acquired(j)$ (from assertion VIII.1) we shall implement as follows: $Acquired(j)$ refers to a particular

[4]If two grasps had the same assertion, then it would be pointless to have two separate grasps.

port on **grasp**^{g} which is written when the object is acquired. Only after the object has been acquired can any manipulation be carried out (in proportion to the grasp being applied). This difference in manipulation ability is quantified by those exact control and range predicates asserted by that grasp.

For example, based on the material in Chapter 2, we can design the combination of control and range predicates which represent each of the Encompass, Lateral and Precision grasps (where n, m are input ports on **grasp**^{g}):

$$\begin{array}{ll} grasp^{\mathcal{E}} & PosControl(j,m,\delta) \wedge (0 \leq \delta \leq \delta_W) \wedge Affixment(Q) \\ grasp^{\mathcal{L}} & PosControl(j,m,\delta) \wedge (0 \leq \delta_\phi \leq \delta_F) \\ & \wedge (0 \leq \delta_\psi \leq \delta_W) \wedge (0 \leq \delta_\theta \leq \delta_W) \wedge Affixment(T) \\ grasp^{\mathcal{P}} & PosControl(j,m,\delta) \wedge (0 \leq \delta \leq \delta_F) \wedge \\ & Stiffness(j,n) \wedge Affixment(W) \end{array} \tag{VIII.11}$$

$$\begin{array}{ll} where & \delta_F = Finger\ precision\ < \ \delta_W = Wrist\ precision \\ and & W < T < Q,\ by\ the\ grasp\ definition. \end{array}$$

δ, again, is a measure of how much position precision is available. δ_F,δ_W are the maximum position errors for the wrist position and finger-tip position respectively. Where necessary, we subscript δ by a degree of freedom, to give precision information on that specific DOF. For example, in the assertion for the Lateral grasp the precision of the ϕ degree of freedom[5] is greatest, since that degree of freedom can be controlled by finger manipulation, while the others must be controlled by the wrist. With the Precision grasp, all degrees of freedom of the object can be controlled with this precision. The affixment ranges are heavily dependent on the hand model; nevertheless, the relationships in VIII.11 are valid for the SSG system, and stem from the grasp descriptions of Chapter 2. An Encompass grasp will always have a greater affixment to the object than either of the other two, and the Precision grasp will always have the least affixment value.

We have already discussed how `object` produces the object characteristic information for grasp selection on the ports sm, fl, lo. Preshaping, acquisition and manipulation are all described relative to the object-frame transformed by the grasp-frame matrix $\mathbf{T}_f$. Once the grasp has been selected, the next function of `object` is to implement this transformation on the position and the orientation of the object (and see Figure 27). `object` should take the position and orientation information which arrives on the ports

[5]Referring to Chapter 2, ϕ is rotation around the $z-axis$ of the grasp coordinate system $\mathbf{T}_{fo}$.

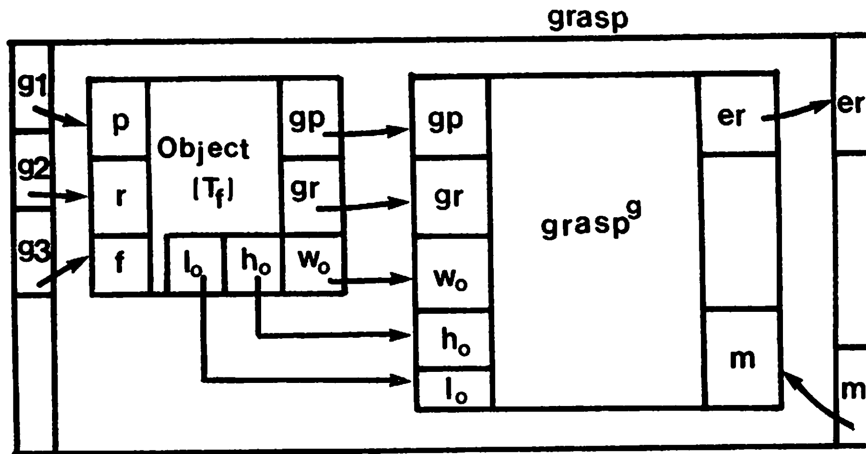


Figure 27: Internal Structure of the grasp Schema.

$g1$ and $g2$ of the **grasp** assemblage into its ports p and r, and transform these by $\mathbf{T}_f$, to give output on gp and gr respectively. We can write the description of the task-unit **grasp** once **grasp$select** has finished as:

$$grasp(g1, g2, g3) \; = \; [object(p,r)(gp,gr) \overset{c}{\longleftrightarrow} grasp^g(gp,gr)()] \quad g \in \{\mathcal{E}, \mathcal{L}, \mathcal{P}\} \qquad \text{(VIII.12)}$$

And the connection mapping is:

$$\begin{array}{lcllcl} g1 & \equiv & object(p), & g2 & \equiv & object(r) \\ object(gp) & \rightarrow & grasp^g(gp), & object(gr) & \rightarrow & grasp^g(gr) \end{array}$$

This description of **grasp** must be taken in conjunction with the one in VIII.8 in order to produce the complete schema description.

The assertion template for **object** (where $\times$ denotes matrix multiplication) then is:

$$\begin{array}{ll} \langle object \rangle & \Box(\; [In(p) \equiv \neg Out(gp) \equiv (\mid p \mid \; = \mid gp \mid)] \wedge \\ & [In(r) \equiv \neg Out(gr) \equiv (\mid r \mid \; = \mid gr \mid)] \wedge \\ & [(\ell(gp) = \ell(p) \times \mathbf{T}_f) \wedge (\ell(gr) = \ell(r) \times \mathbf{T}_f)] \;) \end{array} \qquad \text{(VIII.13)}$$

This specifies **object** as a buffer which transforms its position and pose information by the grasp frame transformation, and then outputs it.

Internal Structure of $grasp^g$: For any grasp $g \in \{\mathcal{E}, \mathcal{L}, \mathcal{P}\}$, **grasp**^{g} has the following structure:

$$grasp^g = [g\$object^g \overset{c}{\longleftrightarrow} (reach^g, preshape^g, acquire^g, manipulate^g)] \qquad \text{(VIII.14)}$$

The object model **g$object**^{g} must embody the preshape transformation for the matrix, $\mathbf{T}_{pg}$. This is implemented, basically, in the same manner as in **object**. **g$object**^{g}

must also present *parameterization* information (as opposed to the selection information represented by **object**) for the phases of the grasp, which are represented by the set of task-units which comprise the motor schemas of the **grasp^g** task-unit. From Chapter 2, this parameterization information is the object position and orientation, and length, width, and height values. Additionally, **g$object^g** determines the ***virtual to physical finger mapping*** for the grasp g. Each grasp will have a different number of virtual fingers; via some number of ports nvf_k (where k is the number of virtual fingers used in the grasp), **g$object^g** informs the motor task-units of the grasp (i.e., **reach^g**, **preshape^g**,etc) of the virtual to physical finger mapping [6]. In summary, **g$object^g** will have the following ports:

$$g\$object^g(gp, gr)(gpp, gpr, len, wid, hgt, nvf_1, \ldots, nvf_k)$$

where the gp, gr values come from those ports with the same names in the **grasp^g** assemblage, and values on these ports are transformed by the (inverse) preshape transformation and output on gpp, gpr. len, wid, hgt are the object's length, width and height measurements respectively. $nvf_1, \ldots, nvf_k$ are the virtual finger ports, where there are k virtual fingers defined for grasp g.

We demand that all **grasp^g** have an error port er, which we shall connect directly to the error port er of **grasp** (see assertion VIII.2). Each motor task-unit of any **grasp^g** will also have an er port, and we connect[7] them *fan-in* to the port of **grasp^g**. In this way, an error message from *any* component of the grasp can be reported. We can now start to separate the components of the grasp assertion, VIII.1, among the components of the grasp task-unit **grasp^g**.

Reaching: The **reach^g** task-unit is charged with bringing the hand to the location of the object. It does so by a two stage process (see Figure 28) [8]: Firstly, the hand is conveyed to some offset δ_z along the $z-axis$ from the object position, at the same time the preshape coordinate frame is oriented by the wrist's degrees of freedom, so that its axes are parallel and in the same sense as those of the grasp frame; secondly, the wrist is moved in along the $z-axis$ until the center of the preshape frame coincides with the center of the grasp frame. The two input ports to the **reach** task-unit, p, r, take the

---

[6] Sections 3.3 to 3.5 of Chapter 2 describe these mappings.

[7] Strictly speaking, we do not connect them; it is an equivalence port mapping, (Chapter 4, Definition 16).

[8] Section 3.2 of Chapter 2.

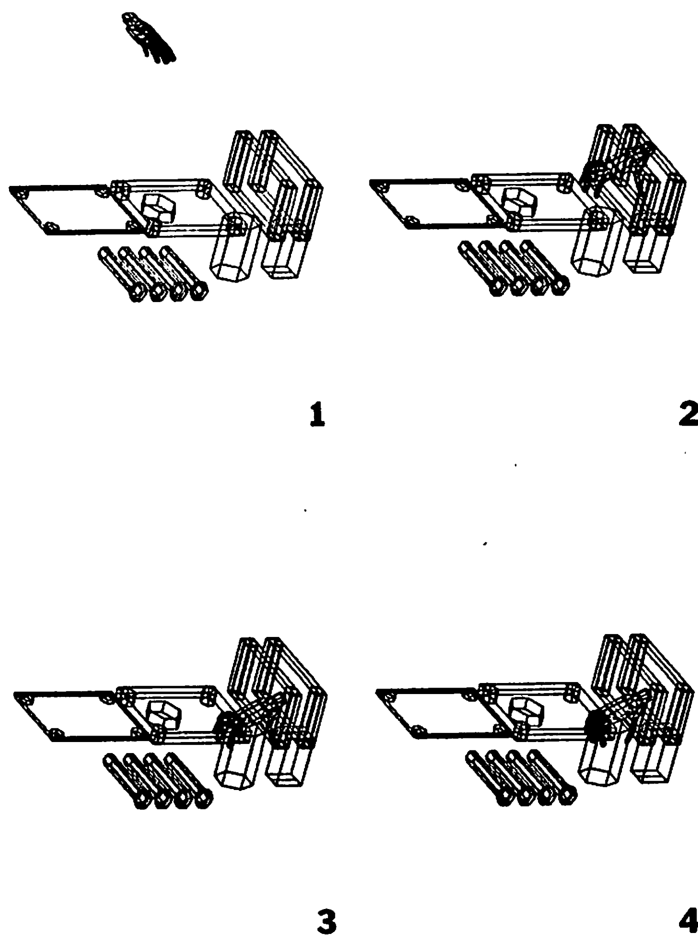


Figure 28: The Reach, Preshape and Acquisition Sequence. From left to right: Initial position; reach to Capacitor minus δ_z; followed by wrist translation of δ_z towards object; and finally acquisition of Capacitor. These graphics are of a simulation of the second grasp instruction in Table 3.

transformed position and orientation information available from **g$object^g**. Note that the reach action is the same for all grasps, so we will drop the g superscript. This prompts us to structure **reach** as:

$$reach(p,r)() = [r\$object(p,r)(pr,rr) \overset{c}{\longleftrightarrow} (move\$wrist(pr)(), orient\$wrist(rr)())] \quad \text{(VIII.15)}$$

and to connect it up as:

$$\begin{array}{lll} p & \equiv & r\$object(p), \quad r \equiv r\$object(r) \\ r\$object(pr) & \rightarrow & move\$wrist(pr), \quad r\$object(rr) \rightarrow orient\$wrist(rr) \end{array}$$

We can now write down the assertion which **reach** has to fulfill:

$$\langle reach \rangle \;\; \Diamond \;\; ([(WRISTPOS(\ell(p) - \delta_z) \wedge WRISTANG(r)) \wedge \Diamond WRISTPOS(p)] \vee Rerror) \quad \text{(VIII.16)}$$

where the predicates $WRISTPOS$, $WRISTANG$ are true, if the robot wrist has a position and orientation, as given by the last value of their argument. We have omitted the precision range for these predicates (Section 3, Chapter 6), to make them easier to read. We use the notation $\ell(p) - \delta_z$ to indicate the number obtained by subtracting the z offset, δ_z, from the last value on the local variable p. The abbreviation predicate $Rerror$ represents an error condition in the reach; we shall assume it has the same structure as all our error predicates, the writing of some error value to a special error port, er.

The **reach** task-unit motor schemas are two basic schemas similar to the detailed example in Section 4.2 of Chapter 6; given a position on their input ports, they cause the movement of, in this case, the wrist's position and rotation degrees of freedom, to some specified values. Those values are (eventually) the object position and location, transformed through **object** ($\mathbf{T}_j$) and through **g$object^g** (inverse $\mathbf{T}_{pg}$). When we take all these connections into account, we can say that **reach** asserts $\Diamond AtObject \vee Rerror$[9], where:

$$\begin{array}{ll} AtObject(j,p,r) \equiv & OBJPOS(j, \ell(p) \times \mathbf{T}_{pg} \times \mathbf{T}_j^{-1}) \wedge OBJANG(j, \ell(r) \times \mathbf{T}_{pg} \times \mathbf{T}_j^{-1}) \\ & \wedge WRISTPOS(p) \wedge WRISTANG(r) \end{array} \quad \text{(VIII.17)}$$

and where $\times$ denotes matrix multiplication, and we use the notation $\ell(p) \times \mathbf{T}$ to indicate the value obtained by multiplying the last value of the local variable p by the matrix $\mathbf{T}$.

[9]Recall that we cannot verify any assertion using the shorthand; for verification, we must use the full schema description. The shorthand is a design tool.

To detect the **approach failure** case, we need to *time-out* on reaching the object, or we need to check for unexpected tactile contacts which may prevent the hand from moving. We shall not go into detail on this, as it is getting away from the grasp domain, and into collision detection and avoidance. However, we leave *Rerror* in place to represent the output of such an analysis: $Rerror \equiv [Out(er) \wedge \ell(er) = approachfailure]$.

Virtual Fingers: The **preshape**^{g}, **acquired**^{g} and **manipulate**^{g} task-units are all structured in terms of actions of virtual fingers. Use of virtual fingers requires two operations: Firstly, some number of physical fingers must be mapped to each virtual finger. Secondly, the mapping between the degree(s) of freedom of the virtual finger and those of the physical finger must be constructed. If $VFset_k$ is the set of physical finger indices considered in VF_k, (remember, this data is available from **g$object**^{g}), and ℓ is the physical finger index, and k is the virtual finger number: We introduce the family of preconditions VFk, which create one instance of their task-unit, T, per physical finger in virtual finger k:

$$\langle VFk \rangle \quad \forall \ell \in VFset_k \supset \Diamond(INST * (T)) \qquad \text{(VIII.18)}$$

The initialization section of each of the **preshape**^{g},**acquired**^{g} and **manipulate**^{g} task-units will embody these VFk. The description of the assemblage from Chapter 5, constrains all SIs created by the initialization section of the assemblage to *start simultaneously*. Hence, VFk will ensure that, eventually, the appropriate number of T SIs are created, and that the assemblage will not start until all of them have been created. Placing the VFk preconditions in the initialization part of the assemblage strengthens the assertion VIII.18 sufficiently to guarantee that all the necessary T SIs exist when the assemblage begins.

The second virtual finger mapping mechanism, going from the degrees of freedom of a virtual finger to those of a particular physical finger in its $VFset$, is accomplished uniquely in each component of the grasp with the goal of maintaining a particular relationship between the virtual and physical finger characteristics.

Preshaping: The **preshape**^{g} task-unit schema must prepare the hand to acquire the object. We overview, first, the abbreviation predicates we shall use in the preshape: An important part of the preshape is to ensure that the object can fit into the hand; for this we use the abbreviation predicate $Canfit(j)$. The preshape must also ensure that the object can be acquired in an appropriate way for that grasp; $TipAcquire(j)$ is used for

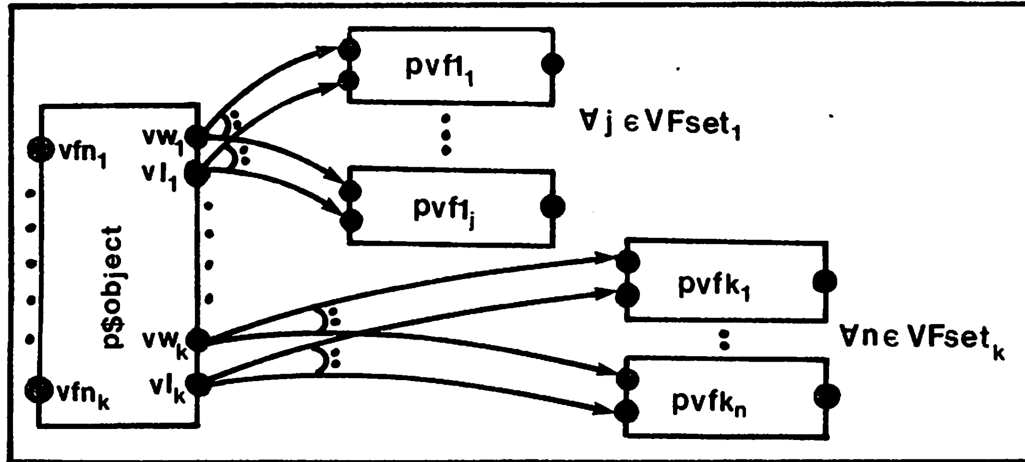


Figure 29: The preshape^g Assemblage.

the Precision grasp's finger-tip acquisition strategy, and *PhalAcquire(j)*, for the Lateral and Encompass grasps' finger phalange acquisition strategy. Finally, the preshape will set up finger positions to ensure stable grasping; *Acquired(j)* $\supset$ *Stable(j)*.

The preshape^g task-unit, for each of the three grasps, has the following generic structure:

$$preshape^g(vfn_1, \ldots, vfn_k) = \\ [p\$object^g(vfn_1, \ldots, vfn_k)(vw_1, \ldots, vw_k, vl_1, \ldots, vl_k) \overset{c}{\longleftrightarrow} VFk : PVFk^g(dist, wid)] \quad \text{(VIII.19)}$$

where k is the number of virtual fingers in grasp g, and these ports are connected as follows (and see Figure 29):

$$\begin{array}{llll} For\ i = 1 \ldots k: & vfn_i & \equiv & p\$object(vfn_i) \\ For\ i = 1 \ldots k, \forall j \in VFset_i: & p\$object(vw_i) & \rightarrow & PVFi_j(wid) \\ & p\$object(vl_i) & \rightarrow & PVFi_j(dist) \end{array}$$

p$object^g** breaks the parameterized object data coming in from **g$object^g into preshape parameters for each virtual finger. **PVFk^g** is the schema which maps from virtual finger actions to physical finger actions. **VFk**, in this case, creates one instantiation of **PVFk**^g_j per physical finger j in $Vfset_k$. **VFk** is a general name, allowing us to describe the preshape for any of the grasps. In reality, we would have **VF1,VF2** for the Encompass and Lateral grasps, and **VF1,VF2,VF3** for the Precision grasp.

The way we represent the fact that different fingers may have different roles in the preshape, is by providing different ports for each virtual finger, i, on **p$object^g** (vw_i, vl_i).

The information which gives the unique role of virtual finger i comes through these ports; vw_i gives the virtual finger width, computed from the object length as described in Chapter 2, and vl_i gives the finger-tip/thumb-tip separation, calculated from the object width, again, as described in Chapter 2. The precondition **VFi** will have created one instantiation of **PVFi** for each physical finger in $VFset_i$; the connection map links all instantiations of $PVFi$ to the same virtual finger ports, vw_i, vl_i, on **p$object**^{g}.

The virtual finger highlights some particular features of the task which are important, and which the fingers must 'obey'. In the case of the preshape, the important information is given by our preshape assertion; the object must be able to fit into the hand, it must be acquirable in the manner of the grasp, and it must be stable (in some sense) when grasped. The virtual finger information in preshape is the finger-tip/thumb-tip separation (so the object can fit in) and the separation of the fingers (spreading the fingers over the object circumference facilitates a final stable grasp in the absence of detailed object knowledge). We have already discussed the use of the ports vw_i, vl_i to give this information for each virtual finger i.

We define the abbreviation predicate $CANFITV(j,k)$ to be true if virtual finger k is open wide enough to fit the target object, j. In that case, the abbreviation predicate $CANFIT(j)$, which we have already described, is:

$$CANFIT(j) \equiv \bigwedge_k CANFITV(j,k) \tag{VIII.20}$$

We define the abbreviation predicate $CANFITF(d,i)$ to be true if physical finger i and the thumb have a separation d. If we can show d is the width of object j, then:

$$CANFITV(j,k) \equiv \bigwedge_{i \in VFset_k} CANFITF(d,i) \tag{VIII.21}$$

We associate this assertion with PVFk:

$$\langle PVFk_i^g \rangle \ \Diamond(CANFITF(d,i)) \tag{VIII.22}$$

Thus, from a predicate which describes whether one finger and thumb are separated widely enough to permit the object passage, we can determine if the object can fit into the whole hand.

<u>Preshape Virtual Finger</u>: For the preshape phase, the mapping from virtual finger to physical finger is accomplished by using the relationship between finger-tip/thumb-

tip separation and finger angles developed in Chapter 2. In all three grasps, interfinger spacing is done using the forward kinematics, again as described in Chapter 2.

$$PVFk_i^g(dis, wid) = [(angsep_i(dis)(f_2,\ldots,f_n), space(wid)(f_1)) \stackrel{c}{\longleftrightarrow} move\$finger_j(f_1,\ldots,f_n)] \quad \text{(VIII.23)}$$

where there are n degrees of freedom in the physical finger ($n = 3$ for the Salisbury hand, $n = 4$ for the human hand). And the connection map is:

$$\begin{array}{llll} dis & \equiv angsep_i(dis), & wid & \equiv space(wid) \\ for\ l = 2,\ldots,n & & & \\ angsep_i(f_l) & \rightarrow move\$finger_i(f_l), & space(f_1) & \rightarrow move\$finger_i(f_1) \end{array}$$

Each PVFk_i^g must assert $CANFITF(d,i)$; from this, we get $CANFITV(i,k)$ as indicated above. $\mathbf{angsep}_i$ is responsible for estimating the finger angles from the linear separation equation for finger i, to ensure the finger-thumb separation is enough to permit the object to pass. At the lowest level, therefore, all these $CANFIT$ predicates must be described in terms of the *endpoint separations*. The information about the object width coming in to PVFk_j^g through dis, allows the 'connection' to be made between endpoint separations and the object j (as discussed above) to derive $CANFITV$ from $CANFITF$. `move$finger` is a schema which moves the physical finger to some defined joint angles (similar to `move$wrist, orient$wrist`).

The SSG system takes a coarse view of stability in grasping. Static stability is set up by ensuring that all contacts exert a small positive force on the object, directed at its center of mass. Dynamic stability is approached by spreading the fingers around the perimeter of the object. This approach is a *heuristic* stability rule which is used when detailed object knowledge is not available. Each PVFk_i^g is sent a virtual finger width (finger spacing within the virtual finger), *or* a virtual finger spacing (finger separation between adjacent fingers in separate virtual fingers). The initialization section of $\mathbf{preshape}^g$ in creating the PVFk^g SIs, decides, based on the algorithm on Page 28 of Chapter 2, Section 3.2. Chapter 2 also details the algorithm for selecting finger separations based on object size in order to spread the fingers around the object; this computation is carried out (as discussed) by $\texttt{p\$object}^g$.

The **space** schema evaluates the finger base angle value in order to set some specified interfinger separation for finger i and its adjacent finger $i+1$. We introduce the abbreviation predicate $SPREAD(s,i)$ which is true if the distance between the endpoints of

finger i and $i+1$ is s. In general, we shall want to use $SPREAD$ to ensure that the fingers are distributed around the object circumference of object j. If we can show this, then our heuristic approach to stable grasping allows us to assert $Acquired(j) \supset Stable(j)$.

Acquisition: The **acquire**^{g} task-unit implements the acquisition phase of the grasp, and is triggered by the precondition `close`. The acquisition phase should begin when the wrist is within some *small* distance of the object j, and when the preshape has finished. We define the `acquire` task-unit to have a structure similar to the **preshape** task-unit:

$$acquire^g() = close : [a\$object^g()(vw_1, \ldots, vw_n) \overset{c}{\longleftrightarrow} VFk : AVFk^g(d)] \qquad \text{(VIII.24)}$$

where there are n virtual fingers in grasp g. The object model `a$object`^{g} provides the feedforward estimation of the object surface for each virtual finger through the port d; this information is passed to the AVFk^g SI via the port vw_k. The angular separation equations of Chapter 2 can be used in conjunction with knowledge of the object width, to estimate at what finger angles contact should occur. This information is necessary to tell when the acquisition has *failed* (more on this later). The assertion which describes the behavior of **acquire**^{g} is:

$$\begin{aligned}(acquire^g) \quad WRISTPOS(p) \wedge \quad & OBJPOS(j,p) \wedge CanFit(j) \wedge AcquPred \\ & \supset \Diamond((Acquired(j) \wedge AffixRange) \vee Aerror)\end{aligned} \qquad \text{(VIII.25)}$$

where the first part of this assertion is the condition in which it is appropriate for the acquisition to begin, i.e., the precondition for the grasp. Let δ_W represent the positioning error of the wrist. Let us assume a schema implementation of *close*, the precondition schema `close`, which has an input port p:

$$\begin{aligned}(close^g) \quad & WRISTPOS(p, \delta_W) \wedge CANFIT(j) \wedge (Acquired(j) \supset Stable(j)) \wedge AcquPred \\ & \supset \Diamond(INST * (acquire^g) \wedge \neg INST(close^g))\end{aligned}$$

(VIII.26)

Remember, abbreviation predicates eventually have to transform to port assertions; hence, `close` will likely have to be connected to **preshape**^{g} (to determine when $CANFIT$ has been asserted) and `reach` (to determine when $WRISTPOS$ is ready) in order to evaluate its antecedent.

Referring back again to the assertion for the `acquire` task-unit: *AcquPred* is the *TipAcquire* or *PhalAcquire*, whichever is appropriate for this grasp. *AffixRange* is an affixment range predicate. We could consider this (as we do the *Acqupreds*) a fixed quantity per grasp type (as outlined in assertion VIII.10). Alternatively, we could calculate

the number of hand-object contacts (from tactile sensors), and by knowing the type of grasp, could use this number to give a specific affixment value for the grasp.

We shall investigate the error component of the acquire phase closely. Again, we define the abbreviation predicate:

$$Aerror \equiv [Out(er) \wedge (\ell(er) = CaptureFail \vee \ell(er) = ContactFail)] \qquad \text{(VIII.27)}$$

VFk makes one instance of $\mathbf{AVFk}^g$ per physical finger defined in virtual finger k; again, the initialization section of the task-unit $\mathtt{acquire}^g$ will set up the virtual fingers, as in preshape. $\mathbf{AFVk}_i^g$ is then responsible for mapping virtual finger parameters onto physical finger i for object acquisition; it implements the contact strategies per grasp described in Chapter 2.

$$AVFk_i^g(d) = [(touch_i()(c), angsep_i(dis)(f_2 \ldots, f_n)) \overset{c}{\longleftrightarrow} (move\$finger_i(f_1 \ldots, f_n), iterate(c, d)(dis, t))] \qquad \text{(VIII.28)}$$

Let us consider the precision acquisition strategy for a moment. Each finger is moved in towards the object until finger-tip contact is made (if joint stiffness is controllable, then they are moved in at low stiffness): **iterate** is responsible for this progression. The angular separation equations can be used to select successive finger-tip postions, in such a manner that the finger-tips follow a trajectory towards *the object's center of mass*: $\mathbf{angsep}_i$ translates a finger-tip/ thumb-tip separation (dis) to joint angles (f_i), given to **move$finger**. We can detect when the finger-tips achieve contact, using port c of **touch**. When the finger-tips achieve contact, which we detect with $\mathbf{touch}_i$,[10] we can determine from the value of the separation at contact if the object has been acquired or not.

Let $Contact(i)$ be some grasp-dependent contact termination condition for finger i; for the Precision grasp this is equivalent to contact on the last link of the finger. $\mathbf{AVFk}_i^g$ has the assertion:

$$\langle AVFk_i^g \rangle \quad \forall i.\Diamond(Objcontact(j) \vee Aerror) \qquad \text{(VIII.29)}$$

which we explain as follows: If contact occurs and the finger-tip separation is greater than the object width, then we have **contact failure**; if it is less that the object width,

[10]And they will eventually touch with something, since at zero finger-tip/thumb-tip separation the finger and thumb touch.

then we have **capture failure**. If *dis* is a port denoting the current finger-tip/thumb-tip separation, and d is the input port connected to `a$object`^{g} which carries the object width value, then:

$$\begin{array}{ll} CaptureFailure \equiv & \bigvee_{j \in VFset_k} Contact(j) \wedge (\ell(d) - \ell(dis) > \varepsilon) \\ ContactFailure \equiv & \bigvee_{j \in VFset_k} Contact(j) \wedge (\ell(d) - \ell(dis) < \varepsilon) \\ Objcontact(i) \equiv & \bigwedge_{j \in VFset_k} Contact(j) \wedge (\ell(dis) = \ell(d) \pm \varepsilon) \end{array} \qquad \text{(VIII.30)}$$

where ε represents the precision with which the object width is known.

Once all contacts have been made, the next step is to set up the contact forces to keep the object gripped. At all times, we shall demand that there is a small positive force on each contact point. Of course, this may change as the object is manipulated, but the forces must remain positive at all times; meaning that the finger-object contact is maintained. We summarize this constraint with the important assertion *Gripped*(i). As we shall see, this assertion plays a vital role in setting up the manipulation control predicates.

Manipulation: The `manipulate`^{g} task-unit, in general terms, is:

$$manipulate^g = acquired : [m\$object^g \overset{c}{\longleftrightarrow} VFk : MVFk^g] \qquad \text{(VIII.31)}$$

The manipulation of the object cannot begin until the object has been acquired. The precondition *Acquired* (from data supplied by `acquire`) determines the correct time to create the manipulate task-unit. The assertion for this precondition is:

$$\langle acquired^g \rangle \quad \square(Acquired(j) \supset \diamond(INST*(manipulate^g) \wedge \neg INST(acquired))) \qquad \text{(VIII.32)}$$

Depending on the grasp, the contents of `manipulate`^{g} will vary. Again `MVFk` provides the mapping from virtual finger to a specific physical finger. For the Encompass grasp, the `manipulate`^{g} task-unit has a much simpler form, consisting of just the `move$wrist` and `orient$wrist` schemas (since `object` and `g$object`^{g} will do the correct mapping from object coordinate frame, through the grasp frame transformation, through the (inverse) preshape frame transformation to the wrist frame). For the Precision grasp and Lateral grasp, however, the mapping from virtual finger to physical finger is done by endpoint calculation[11] [12]; this takes advantage of the fact that the virtual to physical finger mapping is an integral part of the grasp component.

[11]Precision manipulation component, see Chapter 4, Example 24.

[12]Inverse finger kinematics, Chapter 2, Section 3.2.

In all manipulation cases, the assumption which allows us to go from a specification of finger contact points (even when we move this wrist, it is really the finger contacts which move the object) is that the object is considered 'glued' to the hand at the contact points. In general, this fact will hold true if the finger contacts exert a small positive force towards the center of mass of the object. We shall assume that *Gripped*(j) embodies this. Thus, if *Gripped*(j) $\wedge$ *Acquired*(j), then, if either the wrist moves the contact points, or the fingers move (and maintain *Gripped*(j)), then $OBJPOS$ moves. In this way, control predicates, such as *Poscontrol* in VIII.4, can be defined.

We have now completed the general outline of the implemention of the SSG system in $\mathcal{RS}$. In the next section, we present the detailed schema specifications for two of schemas; the `locate$object` precondition, and the `reach` task-unit.

§2.2 SSG Schema Specifications

In this section, we will present the `reach` task-unit, developed in the previous section, in more detail. The trend will now be bottom-up; we will start to construct the detailed schema specifications and their assertions for the abbreviation predicates of the previous section. For each schema specification, we shall include a short informal proof outline to verify the schema assertion template. For each assemblage, we shall use the detailed initialization specification to set up the assemblage assertions and the connection map. A proof outline is included to verify the production of the assemblage assertion from the component SI assertions. For this, we shall rely heavily upon the ***network proof rules*** and the ***network liveness axiom*** developed in Chapter 6.

<u>`move$wrist,orient$wrist`</u>: We shall assume these schemas have assertion templates of the form:

$$
\begin{array}{lll}
\langle move\$wrist \rangle & \Box(& \Diamond In(x) \wedge \\
& & (In(x) \equiv Out(t) \equiv WRISTPOS(x)) \\
& & \wedge In(di) \wedge \\
& & (INST(move\$wrist_i) \equiv (|\ di\ | \ = 0))\ ) \\
\langle orient\$wrist \rangle & \Box(& \Diamond In(x) \wedge \\
& & (In(x) \equiv Out(t) \equiv WRISTANG(x)) \\
& & \wedge In(di) \wedge \\
& & (INST(orient\$wrist_i) \equiv (|\ di\ | \ = 0))\ )
\end{array}
\qquad \text{(VIII.33)}
$$

An output on t will occur for each of these schemas, when the wrist has been placed

as directed by input port x. Note the introduction of the *di* port, a write to which will cause the SI to deinstantiate itself.

reach: The reach task-unit moves the wrist to the position on its input port p, minus some offset along the $z-axis$, and orients the hand to the grasp-frame. It then moves it in along the $z-axis$, so that the preshape frame origin coincides with the grasp frame origin. When both of these have completed, it writes its result port t, to indicate that the reach has completed, and terminates. It is structured as an assemblage of `move$wrist`, `orient$wrist` as motor schemas, and `r$object` as a perceptual schema. We shall omit `orient$wrist` for simplicity, its actions are a subset of those we need from `move$wrist`.

The schema `r$object` implements the synchronization of the two phases of the reach, and the initial offset value.

```
[  r$object
   Input-Port-List:     ( p:Vector t1:Real )
   Output-Port-List:    ( po:Vector fini:Real)
   Variable-List:       ( temp:Real )
   Behavior:            (po:=p-(0,0,δ,1);
                        temp:=t1;
                        po:=p;
                        temp:=t1;
                        fini:=1;
                        Stop   )               ]
```

$$\langle object \rangle \quad \Box(\quad [(\mid t1 \mid\ < 2 \supset (In(t1) \equiv \neg Out(po) \equiv In(p) \equiv (\mid p \mid\ = \mid po \mid))) \wedge (\ell(po) = \ell(p) - \delta * (1 - \mid t1 \mid))] \wedge [(\mid t1 \mid\ > 1) \equiv Out(fini)] \quad) \qquad \text{(VIII.34)}$$

Proof Outline: We reason informally that the above assertion describes the schema `r$object`. The first line of this assertion describes an invariant relationship between the input and output at the ports p, po and $t1$. The relationships can be verified by inspection. For example, p must be read before po is written; then $t1$ is read. Initally, $\mid t \mid\ = 0$, hence, the value written to po is $\ell(p) - \delta$, that is, the position of the object minus some offset on the $z-axis$, δ. Subsequently, the length of $t1$ can be 1, meaning that $\ell(po)$ is written. However, when the length of $t1$ gets to be 2, then the above input-output relationship is voided, and $out(fini)$ is asserted. Note that the actual value of $\ell(t1)$ is irrelevant; it is simply used as a *synchronization* between the two parts of the reach.

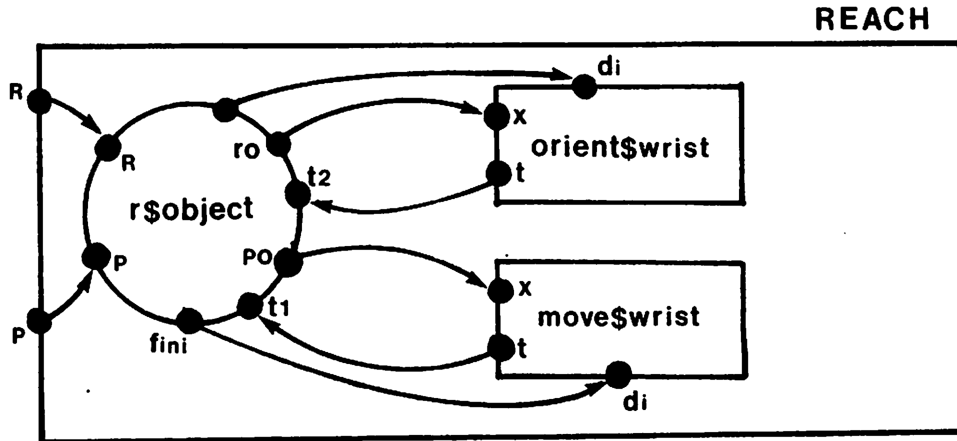
We now define **reach** task-unit:

```
[  reach                  Task-Unit
   Input-Port-List:       ( p:Vector r:Vector h:Integer )
   Output-Port-List:      ( t:Integer )
   Sensory-Schemas:       (r$object)
   Motor-Schemas:         (move$wrist,orient$wrist,add)
   Initialization:        (r$object(≡p) () ();
                           move$wrist(r$object(po))
                             (r$object(t1) r$object(fini)) ();      )  ]
```

We can characterize this task-unit initialization as:

$$\Diamond(INST*(r\$object) \wedge C1*) \wedge \Diamond(INST*(move\$wrist) \wedge C2*) \qquad \text{(VIII.35)}$$

$$\begin{array}{llcl} C1*: & reach(p) & \equiv & r\$object(p) \\ C2*: & move\$wrist(x) & \leftarrow & r\$object(po) \\ & r\$object(t1) & \leftarrow & move\$wrist(t) \\ & move\$wrist(di) & \leftarrow & r\$object(fini) \end{array} \qquad \text{(VIII.36)}$$



We want to show that **reach** can be described by the assertion:

$$\langle reach^{g} \rangle \quad \Diamond \quad ([(WRISTPOS(p - \delta_z) \wedge WRISTANG(r)) \wedge \Diamond WRISTPOS(p)] \vee Rerror) \qquad \text{(VIII.37)}$$

<u>Proof Outline</u>: By application of the assemblage rule, we get the assertion characterizing **reach** as the conjunction of the two assertions, VIII.33, and VIII.34. We then use the connection maps $C1*, C2*$ to show that VIII.37 can be deduced from this conjunction.

Initially, we have all traces of length 0, and, therefore, $In(p)$ from VIII.34. For convenience, we shall assume that whatever is passing information to **reach** is always

transmitting. In that case, we shall get $\Diamond \mid p \mid = 1$ from liveness, which implies $\Diamond(Out(po))$ and $\ell(po) = \ell(p) - \delta_z$. From liveness and VIII.33, we can deduce that, eventually, the port connected to po, x, will experience an event. From VIII.33, we have then $\Diamond(WRISTPOS(x))$ and $\ell(x) = \ell(p) - \delta_z$. This is the first part of the assertion proved. We now show that getting here implies $\Diamond(WRISTPOS(p))$. From liveness and $C2*$, we can get $\Diamond(\mid t1 \mid > 0)$; therefore, from VIII.34, and repeating the above steps, we get $\Diamond(WRISTPOS(p))$.

That concludes all the full schema descriptions we shall verify in this chapter. Appendix A contains descriptions and proof outlines for some of the other task-units in the SSG implementation.

CHAPTER IX

COMPARISONS

In constructing $\mathcal{RS}$, we referenced other related, or motivating work. To give a better view of how $\mathcal{RS}$ fits in to the literature, we shall now compare and contrast it in some detail with other work. Few models in the literature have similar goals and structure to $\mathcal{RS}$, but some are close enough that similarities and differences can be meaningfully noted. In the following sections, we shall compare $\mathcal{RS}$ with:

1. Robot and AI models
   (a) Previous Schema Work.
   (b) Robot Computation Models.
   (c) The NBS Robot System.
   (d) The Actor Model of Computation.
2. General Models
   (a) Hoare's Communicating Sequential Processes.
   (b) The Programming Language OCCAM.

§1. Robot and AI Models

§1.1 Previous Schema Work

The concept of schemas was created by Michael Arbib, and his has been the dominant voice in this area. However, Arbib's ideas have been applied to tactile sensing by Ken Overton [69], and to the vision domain by Hanson and Riseman, and Weymouth

[86]. Both of these have taken the basic schema structure of [5,6] and constructed a programming system from it.

We discussed Overton's schemas in Chapter 3. In summary, a robot task is decribed by a number of schemas; each of which is a unit of action for the robot, an abstract data-type which becomes active when the environment matches some expected state. Overton's schema has the following components: an *activation section*, in which the sensory and goal state of the robot is analyzed to determine whether the action represented by this schema is appropriate in the current state of the world; an *event section*, in which the action described by this schema is programmed, including the necessary sensing to correctly parameterize this action; and a *memory section*, in which prior executions of this schema can be used to modulate the next execution.

Overton does not produce an extended justification for his choice of structure. Additionally, his schemas have no formal semantics, and have no concept of verifiable behavior. His system is constructed at a higher level of computing 'granularity' than $\mathcal{RS}$. Overton has no concept of an assemblage, and communication between schemas is implemented by competing *activation levels*, which are calculated based on sensory input and the contents of a *goal blackboard*. His schemas have special commands for setting goals on this blackboard. Schema instantiation can only occur through an activation section (not directly by a command from another schema instantiation); data passage from one schema instantiation to another can *only* occur via the medium of the goal blackboard.

A task-unit precondition in $\mathcal{RS}$ is similar to his *activation section*; a task-unit is itself equivalent to his *event section*, and its perceptual schemas embody the sensing necessary for the task. Overton associated an *activation level* with each schema instantiation, calculated by his activation section – the activation level embodies how *well* the activation section condition was matched. The precondition in $\mathcal{RS}$ can emulate this by passing a number to the created task-unit; however, there is no explicit *activation level* in $\mathcal{RS}$. His *memory section* (probably coming from the definition of a motor schema in the psychological literature) is harder to map into $\mathcal{RS}$, involving, as it does, a lot of hidden mechanisms. The memory section adapts 'tunable' variables within a schema, by determining if they are frequently set to particular values — these values then become embedded into a new version of that schema. The only way to implement this in $\mathcal{RS}$ is by programming it explicitly.

In summary, Overton's schemas are constructed at a much higher level of computing granularity than $\mathcal{RS}$, and are essentially an informal construction. Some of his concepts

map directly to $\mathcal{RS}$, and his schemas could be constructed in the $\mathcal{RS}$ model. A basic disadvantage of his system is the inability to pass data and control (synchronization) between computing agents, except through a goal blackboard structure. The concept of an activation level, however, seems to be a useful tool with which to structure robot programs.

The schema system developed by Weymouth [86] for visual processing has more structure than Overton's system. However, it does not have any formal semantics, and again, there is no concept of verification of behavior. The result of this (as we have mentioned in the design of $\mathcal{RS}$) is that it is difficult to assess how the model suits some domain (other than simply programming many examples), and exactly how powerful (expressive, computational), the model is. Note that in $\mathcal{RS}$ we can demonstrate a list of model properties with well-defined semantics, which are targeted at specific robot domain characteristics (Chapters 3, 4 and 10).

Weymouth's schemas are sequential processes which communicate by asynchronous message passing — this we can duplicate in any number of ways in $\mathcal{RS}$. There is also an assemblage-type mechanism; each schema can be structured as a number of concurrent *strategies*, forming an internal network. This structure is non-hierarchical, and the schema instantiation can exist even if all the internal strategies cease to exist (although the schema instantiation will not then play any role in processing).

The use of separating the identity of the assemblage from the identity of the network of which it is composed is doubtful. However, the non-hierarchical structuring can sometimes be useful. For example, to create an assemblage of a number of existing SIs, optionally including some newly instantiated schemas. This could be used to incorporate some existing visual feature (the existing SIs) into some higher level visual model (the newly created SIs).

There is no concept of operations across all instances of a schema in the vision schema system — despite the obvious use we made of this for locating visual representations of objects. Although such an operation might be expensive in pixel-level processing, at the level of features, or aggregates of features, it provides an excellent parallel search facility.

In summary, the Weymouth schema system, like the Overton system, has no formal structure. Communication occurs by asynchronous message passing, which can be modeled directly in $\mathcal{RS}$. The assemblage-style construct is non-hierarchical, which seems useful. However, there is no notion of operations across all instances of a schema, which

also seems like a useful tool in visual analysis.

§1.2 *Robot Computation Models*

There has been little work in this area, as we mentioned in Chapters 1 and 3. However, one system which was instrumental in the construction of $\mathcal{RS}$, was Geschke's RSS. We compare $\mathcal{RS}$ to RSS briefly.

RSS [26] is a concurrent model; the computing agents are called *servo processes*, and they do not communicate with each other. Each servo process implements some connection between sensing and action. A servo process in RSS is equivalent then to a task-unit in $\mathcal{RS}$. Task-units are more versatile than RSS servo processes in that they can be of arbitrary complexity, whereas, in RSS there are *fixed classes* of servo processes. This would be like limiting $\mathcal{RS}$ task-units to include just the primitive schemas.

RSS is not a formal model of computation, nevertheless, it does have a notion of verification. RSS is constructed so that the program descriptions are similar to the form of control theoretic equations. In this way, if the control theory analysis is correct, then the RSS servo processes should control the robot correctly. This trend in language design has been taken a step further by Ish-Shalom [38].

§1.3 *The NBS Robot System*

Albus and his group at the National Bureau of Standards (NBS) have constructed an interesting robot control model. Their model is composed of general statements describing robot computation, as well as the details of a *particular* robot control system. The former we can compare directly with $\mathcal{RS}$. The latter, on the other hand, the details of a particular robot control system, are not addressed by $\mathcal{RS}$ (they would be some program in $\mathcal{RS}$). We overview the NBS system briefly, then compare it with $\mathcal{RS}$. Finally, we consider how the two might complement each other.

Overview of the NBS System

The NBS group has constructed a hierarchical robot control system consisting of separate, but coupled, *sensory, modeling* and *control* hierarchies (see Figure 30)[2, 11,13]. In the *control hierarchy (H)*, the commands and actions that are input into each level are decomposed into sequences of more primitive actions which are then fed to the next

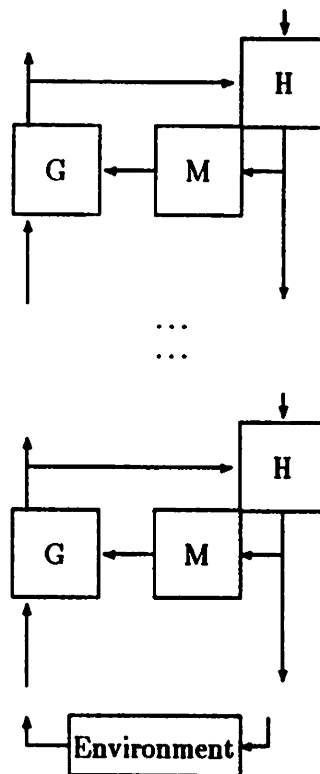


Figure 30: NBS Hierarchical Robot Controller.

lowest level. The time scale decreases as actions descend the hierarchy. Each level of the control hierarchy receives sensory input from an appropriate level of the *sensing hierarchy (G)*. This sensing hierarchy works bottom-up (the control hierarchy, by decomposing actions, works top-down). At the lowest level of the sensory hierarchy, position, force and torque data are measured; at the next highest level tactile, proximity and simple visual data are measured. At higher levels, objects are constructed from these basic sensory measurements. After each level in the hierarchy, the output actions are fed to the appropriate level of the *modeling hierarchy (M)* to generate expected sensory conditions. High correlation between expected sensations, as generated by the modeling hierarchy, and sensory data, as generated by the sensory hierarchy, indicates the task is progressing well. The modeling hierarchy is essentially a knowledge base embodying information known about objects and tasks.

The NBS hierarchical control concept is applied from the level of the factory to the level of the robot [13]. A typical list of the levels of the hierarchy for a factory might be: factory, shop, cell, workstation, equipment. For a single robot the levels might be [11]: task, subtask, primitive-action, trajectory, coordinated joint motion, servo. A particular set of levels constitutes a *particular* robot control system.

At all levels, the H, G and M modules are *concurrent* processes. Communication is achieved with a blackboard-like common memory. Each module is both considered to be, *and* programmed as, a finite state automaton. Barbera [1982] argues eloquently for state tables, and the IF-THEN of expert system fame, as a superior programming style in the context of the NBS system, rather than the procedural programming style. The argument is that the state-table approach shows more clearly the input and output relationships of a task.

Recently, some papers have described the sensory and modeling hierarchies as separate from the control hierarchy [40, 78]. Except at the lowest levels, the main task of this combined hierarchy is to construct a description of the workspace and objects in it, which is continually updated in real-time. In general, the amount of interaction between the control and sensory-modeling hierarchies is small, thus maximizing utilization of the inherent parallelism between them. Nevertheless, the control hierarchy can pass commands to focus attention, and will, of course, query the sensory-modeling hierarchy for object descriptive data.

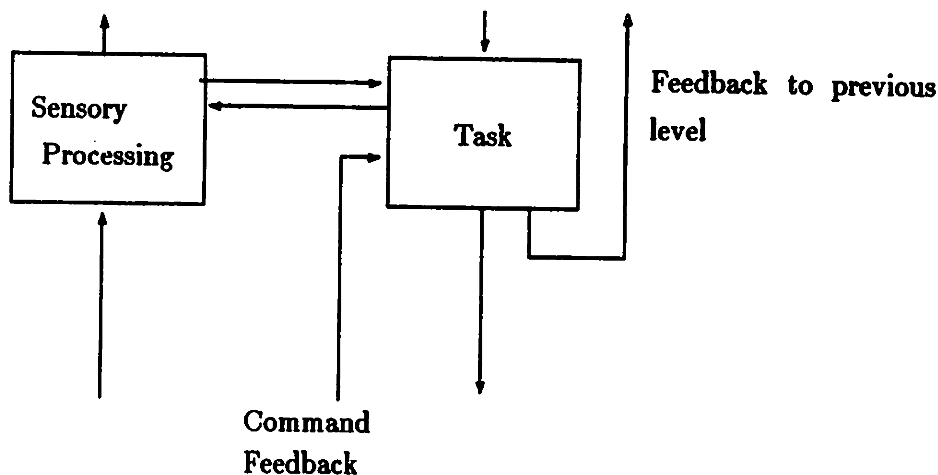


Figure 31: A Level of the NBS System.

Comparison with $\mathcal{RS}$

Some major similarities between the NBS system and $\mathcal{RS}$ are:

- The recognition of the hierarchical structure of robot tasks.
- The role of sensation and action in a particular task-unit (or level in the NBS system, see Figure 31).

However, the way each system chooses to use these structuring concepts is quite different. The NBS hierarchy is fixed and explicit; in $\mathcal{RS}$, the sensori-motor hierarchy is implicit, and could have recursive subtrees (but not a tangled hierarchy, the recursion must have a basis case). Because it is a formal system, assertions can be meaningfully associated with the 'levels' in a $\mathcal{RS}$ hierarchy (these are the assertion templates). In addition, the hierarchy in $\mathcal{RS}$ is *dynamically* constructed, in a fashion appropriate to the environment and to the task. Levels can be added or removed, as necessary.

$\mathcal{RS}$ tends to emphasize the structure of action and perception at a particular level, i.e., the task-unit, as opposed to the global sensori-motor hierarchy in the NBS. The NBS emphasis has led to the preferential development of the sensory and modeling hierarchies over the control hierarchy. $\mathcal{RS}$ allows for a more task-directed object modeling system; the task being done can assist in the construction of appropriate object models. Although the NBS sensing and modeling hierarchy does accept input from the control

hierarchy for focusing purposes, it must still maintain its own complete object models of the environment.

This leads back to the NBS view of the hierarchy as fixed and explicit; thus the levels of the sensing hierarchy never change, and specific built-in models can be utilized. The *RS* view is that the object models in the environment are closely tied (beyond some fundamental level; we shall return to this) to the tasks being done. The primitive schemas and the pre-supplied assemblage construct the lower levels of the implicit sensori-motor hierarchy. But the user program, by specifying object model information as part of the task-unit, also instructs the modeling part of the sensory system, i.e., what features to aggregate into an 'object' for this task (and, hence, what features to pay attention to). Thus the programmer constructs the dynamic sensori-motor hierarchy in programming the task.

In summary, *RS* concentrates more on the task side of robot programming than the NBS system does, and *RS* allows for task input into the process of building useful object models. There are stages of sensation which are independent of the task being done; and the NBS system expresses these well. *RS*, on the other hand, deals only superficially with these areas in terms of the primitive sensory schemas.

The NBS system considers all levels in all hierarchies as concurrent processes interacting through a shared memory. *RS*, on the other hand, is a true distributed model and could be implemented on a multiprocessor or distributed computer system with message passing. The NBS system could be rephrased as a message passing system, the common memory does not seem fundamental to its structure. *RS* has already made this transition.

The NBS system programs all its levels using state tables. Essentially, Barbera [11] claims that error and exception cases are more easily 'added on', with minimal interactions with existing lines in the state table. In the light of bad experience with the expandability of expert system rule bases, this argument may not hold up. This is especially true since the internal state of a module can be used as part of the state table, hence, giving rise to meta-rules, which complicate tremendously the introduction of arbitrary new rules (or lines in the state table). *RS* uses an AI style frame-like concept, the schema, for describing computing agents. The schema behavior component is described in a standard ALGOL-style syntax; thus one could program IF-THEN rules in it. However, one could also program in any other style.

It is clear that portions of the NBS system and the $\mathcal{RS}$ model complement each other. Specifically in terms of sensing, in which $\mathcal{RS}$ is less precise than the NBS system, and in terms of task description which is more developed in $\mathcal{RS}$ than in the NBS system.

§1.4 *The Actor Model of Computation*

The Actor model of computation [30] is a general object-oriented model of computation, derived from languages such as Smalltalk and Planner and from the λ-calculus. A computing agent in the model is called ***an actor***. An ***event*** in the model is the receipt of a message by an actor. All communication is asynchronous and unbuffered. An actor is defined by how it behaves when it receives a message; this is described by its ***script***.

Each actor has an *acquaintance* list, which describes the 'context' for that actor, that is, the names of other actors that it knows. An actor can create other actors, and can send messages to any actor explicitly named in its script, or that is in its acquaintance list, or whose name is sent to it in a message.

An actor is described by a sequential process in the same manner that a schema is defined in $\mathcal{RS}$. Actor creation and schema instantiation are similar in a number of ways; the new computing agent is 'connected up' in a local context when created, and there may be several actors working from the same script. Unlike $\mathcal{RS}$, actors have no explicit concept of instantiation number (and, hence, a `forall` instances operation is impossible).

The following is an example recursive **factorial** actor from Clinger [1981]:

```
(factorial=  accept              [           continuation n ]
             if or    [ (lessp   [ n 1])     (equal [ n 1 ]) ]
                      then       send 1      to continuation
                      else (     create (    (multiply-by-n
                                             = accept [ x ]
                                              send times [n x] to continuation))
                                 send [      multiply-by-n (minus [ n 1 ])]
                                  to         factorial))
```

This can be compared with the `factorial` schema example in Chapter 4, Example 11. The factorial actor creates three actors, recursively, all of which share the script:

```
accept [ x ]
send times [ n x ] to continuation
```

Any message to **factorial** needs to specify where the result is to be sent (the **continuation** variable). **factorial** should be sent a message like `[ user 4 ]`, meaning evaluate the factorial of 4 and send the result to the actor called `user`. Each recursive instance of

the **multiply-by-n** code would then have the **continuation** variable then bound from **factorial**'s acquaintance list to the previous instance. Eventually, a copy would be made which can send 1 back along **continuation**, which is connected to the instance which occurs previous to it, etc.

Clinger [1981] constructs a formal semantics for actors, based on power-domains. The port automaton model, on which $\mathcal{RS}$ semantics is based, was developed in response to the complexity of power-domains: Port automata offer a more intuitive and less complicated means of formally expressing the behavior of a process. A basic structure on which $\mathcal{RS}$ semantics is built, is the port connection of two port automata. This provides us with the basis for viewing a network of computing agents as a single computing agent; that is, it provides us with the semantics of the assemblage. There exists no such formal structure for actors.

§2. General Purpose Models

§2.1 Communicating Sequential Processes

In Hoare [1978], a concurrent programming language called *CSP* (Communicating Sequential Processes) was introduced. It was novel in that it used input and output as its fundamental operations. A newer version of CSP is introduced in [16,32], together with a formal semantics for concurrent programming. This new version is more a computational model than a programming language. CSP is oriented towards general-purpose concurrent programming; $\mathcal{RS}$, on the other hand, is designed for a particular kind of programming, the specification of robot behavior. It is useful, nevertheless, to compare CSP and $\mathcal{RS}$ since they share some design aims (concurrent processing) and some structural features in common.

The Theory of CSP

CSP has as its basis the concept of instantaneous *events* and processes, which engage in events. The behavior of a process is recorded as a sequence of events up to some moment in time. This is called the *trace* of the process, and it is finite in length. Hoare [1985] introduces a notation for specifying the behavior of a process P in the following manner: *P* engages in some event or set of events *e*, and from then on behaves identically

to process Q. If e is an event and P and Q processes, then in CSP notation: $P = (e \rightarrow Q)$ or where $e1 \dots en$ are events: $P = (e1 \rightarrow (e2 \rightarrow \cdots (en \rightarrow Q) \cdots))$.

In [32] and [16], a comprehensive and elegant set of operators for combining processes and events is described and their nature investigated. The basic policy of CSP is not to minimize on the number of operators, but instead, to define as many operators as seems appropriate to investigate a range of distinct concepts. For example, there are five distinct process composition operators:

1. Parallel Composition, $R = P \| Q$: An event can only occur in the trace of R, if both P and Q have simultaneously executed that event.

2. Non-deterministic Composition, $R = P \sqcap Q$: R behaves exactly like P or Q, but it is not possible to determine in advance which one.

3. Conditional Composition, $R = P [] Q$: R will behave exactly like P or Q, but the choice will depend on the first event offered to R.

4. Interleaving Composition, $R = P ||| Q$: An event in the trace of R requires only that either of P or Q participates in that event.

5. Sequential Composition, $R = Q; P$: R behaves like Q until Q terminates, then it behaves like P.

A *communication* event in CSP is described by a pair $c.v$, where c is a *channel* and v is a data message on that channel: $c!v$ denotes writing the value v to c, and $c?x$ denotes reading a value from c into x. A channel is used for *synchronous, unidirectional* communication between *exactly two* processes. Thus in $R = P \| Q$, the communication event $c.v$ in the trace of R results when P engages in the communication event $c!v$ and Q engages in the complementary event $c?x$.

Brooks et al. [1985] want to be able to distinguish between processes by observing their traces. They introduce $initials(P)$, the set of events which the process P can engage in on its first step, and $refusals(P)$ the set of events which, if offered to P by its environment, will cause P to deadlock on its first step. A *deterministic* process can then be defined as a process which can never refuse any event it can engage in. A non-deterministic process, for example, might, at any stage, be able to engage in some event which would not result in deadlock; however, it might non-deterministically "choose" not to.

The concept of verification in CSP consists of ensuring that a process P meets its specification S: P **sat** S. S will be a trace specification; and P **sat** S is considered true if all traces of P imply that S is true. Although Hoare develops a wide range of laws about operations on traces, his proof method is (he admits) informal.

Comparison of $\mathcal{RS}$ and CSP

The styles of computation described by $\mathcal{RS}$ and CSP are quite different: CSP offers a wide range of operators and has a wide range of application; $\mathcal{RS}$ is targeted at a particular type of computation with specific characteristics. Despite this, there are common threads within the two.

Communication in CSP can occur between exactly two processes, by the act of both processes referencing the same channel in complementary communication events. Communication between two computing agents in $\mathcal{RS}$ is explicitly specified when an entry in the network connection map maps the input port of one computing agent to the output port of another. The specification of the connection is quite *separate* from any communication which might occur on that connection. Communication then occurs by the simple act of computing agents synchronously reading input ports, or writing output ports.

$\mathcal{RS}$ has only one process composition operator, the Assemblage operation, the semantics of which is spelled out operationally via the port automaton model. However, since the network connection map is not constrained to be a bijection, the semantics of fan-in and fan-out can be used to express the multiple composition operators in CSP (see Figures 32,33, 34).

With $\mathcal{RS}$, we have attempted to provide a more formal verification and design methodology (in temporal logic) than that of CSP. The $\mathcal{RS}$ trace definition and the CSP trace definition are identical when CSP events are restricted to communication events. The $\mathcal{RS}$ definition of behavior is, unfortunately, much more complex than the CSP definition, but it is necessary for the development of the temporal logic methodology. Verification is more structured in $\mathcal{RS}$ than CSP, consisting of the 2 step method discussed in Chapter 6 — deriving, or verifying, an assertion on SI behavior first, then applying the assemblage rule to derive or verify the behavior of a network of SIs, an assemblage.

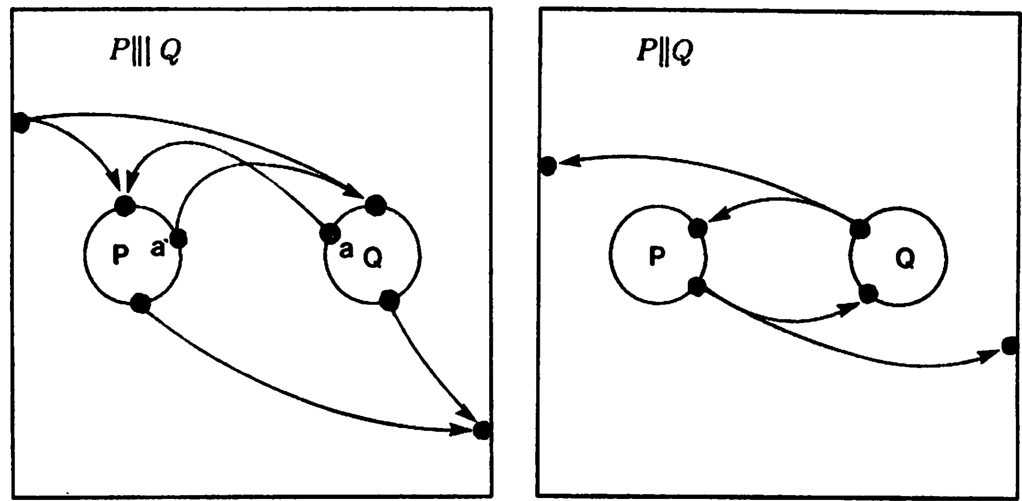


Figure 32: $\mathcal{RS}$ and CSP Composition Operations: Part I.
On the left: Interleaving composition; Non-determinism (as we have defined it in Chapter 5) in the order of OR-semantics fan-out means one of P or Q will receive data written to the input port before the other. The ports a,a' are used to inhibit a duplicate response. If P receives the message first from the fan-out, then it sends a unique failure value over a' to inhibit Q from responding. On the right: Parallel composition; R only engages in any event that P and Q simultaneously engage in.

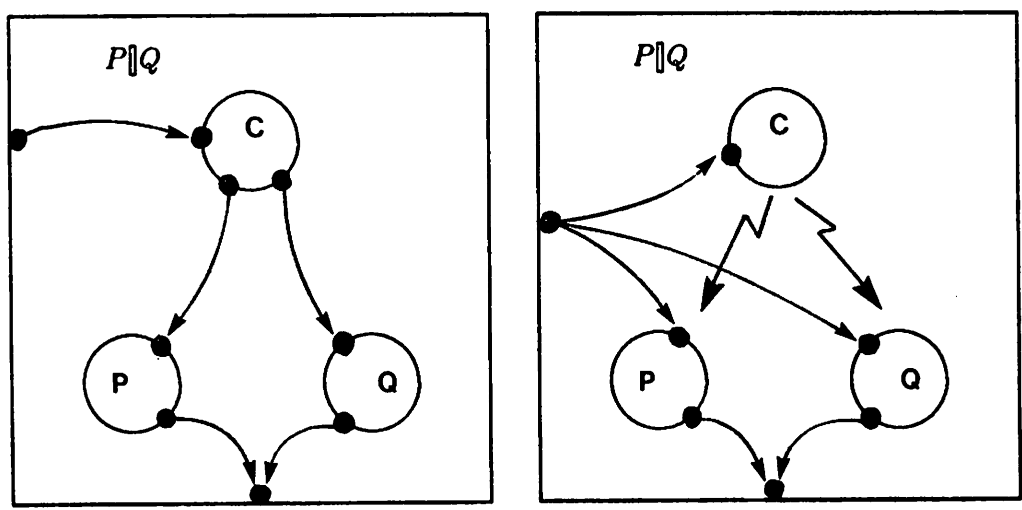


Figure 33: $\mathcal{RS}$ and CSP Composition Operations: Part II.
Two possible implementations of conditional composition. On the left: C choses to forward messages to P or Q. On the right: C chooses to create P or Q to handle all future messages.

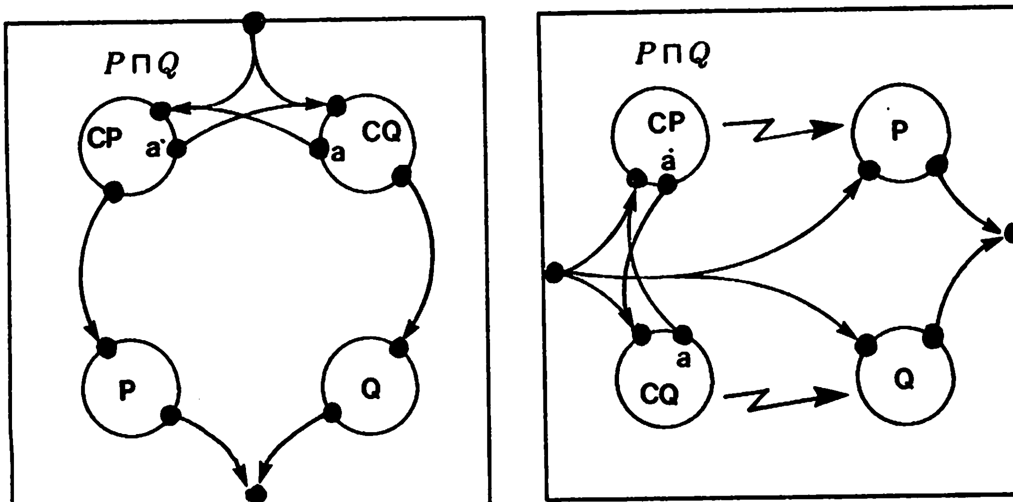


Figure 34: $\mathcal{RS}$ and CSP Composition Operations: Part III.
Two possible implementations of non-deterministic composition. On the left: The non-determinism in the order of OR-semantics fan-out means CP or CQ will receive the message first and inhibit the other via the ports a,a'. If CQ receives the message first, it inhibits CP via a, and then routes messages through to Q. On the right: Again, non-determinism in the order of fan-out is used to make a non-deterministic choice of whether CQ or CP receives the message first. If CQ receives the message first, it inhibits CP via a and then creates Q to handle all further messages.

§2.2 OCCAM

OCCAM[1] [59] is a concurrent programming language whose design philosophy is "to be as simple as possible". It also closely follows CSP [32,p240]. The three primitive processes in OCCAM are: assignment, input from a channel and output to a channel. The process composition operators are: SEQ, for CSPs sequential composition; PAR, for CSPs parallel composition; and ALT, for CSPs conditional composition. Again, process communication is by channels, and is restricted to occur synchronously between exactly two processes. OCCAM supports a procedure operation, PROC, similar to procedures in conventional languages, except that a procedure call in OCCAM will result in the creation of a process. Procedures can take variables and channels as parameters.

INMOS have constructed a processor chip, the transputer [37], designed to be an element in a multiprocessor system, and to run OCCAM as its assembly language. The chip has fast and efficient process switch capabilities (essential for a concurrent programming basis), and also has an on-chip process scheduler. Each chip has four links for connection

[1] After 'Occam's Razor', attributed to the 14th Century William of Ockham.

to other transputers. OCCAM programs run transparently on transputer networks, the transputers handling the inter-transputer channel communication. Thus, the actual size and configuration of the network is completely transparent to the programmer.

RS and OCCAM

OCCAM is an excellent vehicle to implement *RS*. Its structures and that of *RS* are similar enough that a basic set of *RS* could be implemented quickly. An (cursory) example of this mapping is:

A port in *RS* can map directly onto a channel in OCCAM. Port connections can be modeled by PIPE[2] processes in OCCAM: processes which take input on some channel and simply write it to another. A schema can be considered equivalent to a `PROC` declaration in OCCAM (the body of which is `WHILE TRUE SEQ` ···), and the set of ports of a schema can be channel parameters to the `PROC`. The instantiation operation consists of invoking a parallel instance of the `PROC`, with PIPE processes modeling how the channels of the `PROC` are connected to channels on other `PROC` instances. Fan-in and Fan-out on connections can be treated exactly as their semantics are defined in the port automaton model, since, the port automaton model also demands one-to-one connections.

The ability to use, and ease of use, of a transputer network greatly simplifies the testing of *RS* in a true distributed environment. A basic problem may arise in that our work is linked with other work in developing appropriate scheduling [74] algorithms for robot control systems. The fact that the process scheduler in a transputer is on-chip may hinder this research. On the other hand, further investigation may show that the transputer scheduler can be used as a unit in the distributed scheduling algorithm.

---

[2]CSP PIPES, 32, p151.

CHAPTER X

CONCLUSION

§1. Summary

This dissertation began by considering the fact that the robot systems were becoming too complex to continue being programmed as simply the peripherals of a general-purpose computer. As a solution to this, we advocated analyzing the computational characteristics of the robot domain for the purpose of designing a model of computation which was specifically aimed at robot control.

This analysis was begun by investigating what was involved in programming a particular complex robot system, a dextrous hand. In Chapter 2, we developed an example grasping and manipulation framework for a dextrous robot hand. This framework was novel in that it accepted task requirement input to the grasping selection process, and in that it was independent of the hand model used. This example, the SSG system, was then carried all the way through the dissertation.

Chapter 3 developed a set of four observations about the computational characteristics of the robot domain which were used as the fundamental principles upon which the computational model was constructed. These four observations were:

1. The inherent parallelism in robots and robot programming.
2. The special role of perception and action in robotics, and the connection between the two.
3. The usefulness of prototypical action plans and object models.
4. The need for formal semantics and verification methods.

With these basic specifications, Chapter 4 constructed the $\mathcal{RS}$ model of computation as a series of informal examples. $\mathcal{RS}$ is a model of distributed computation to suit the

parallelism inherent in the robot domain (Observation One). The computing agents in the model are created by making parameterized copies (closures in the programming terminology) of a generic description called a schema. This mechanism was chosen to allow for prototypical action plans and object models (Observation Three). The assemblage construct was introduced to structure both object models and plans (Observation Three), to allow local groupings of control SIs (Observation One), and also to form the basis of the task-unit (Observation Two). The task-unit is a special assemblage with designated sensory and motor components. The task-unit represents a task; its perceptual SIs are the object model(s) for the task, and the motor SIs the basic motor actions of the task. This reflects the parallelism between perception and action, and also the special connection between the two.

Chapter 5 developed the formal semantics for the $\mathcal{RS}$ model based on the port automaton model of Steenstrup, Arbib and Manes [1983]. Our first concern was to built a version of a port automaton which suited our schema definition. Having satisfied ourselves that we had not lost any of the power or versatility of the original port automaton model, we began to construct the semantics of $\mathcal{RS}$. The important steps here were to be able to characterize well the behavior of SIs, the nature of communication between them, and the operations for creating, deleting and aggregating SIs.

Chapter 4 introduced a programming notation, and Chapter 5 gave the formal semantics for this notation in terms of the construction of a port automaton. These semantics were used again later in Chapter 6, in the definition of a verification methodology for $\mathcal{RS}$. Chapter 5 described how fan-in and fan-out in the connections between ports yielded interesting behaviors; in particular, it demonstrated how asynchronous communication could be accomplished in a synchronous communication paradigm. The port connection automaton of Steenstrup, Arbib and Manes was used as the semantics of the instantiation, deinstantiation and assemblage (aggregation) operations.

In Chapter 6, we developed a temporal logic methodology to write schema specifications, and to verify if a schema, or a network of SIs, satisfied that assertion. The trace model of Nguyen et al. [1984] provided the basic structure of the temporal logic model. The definition of the *input-output trace* of a port automaton was then used to adapt this model to $\mathcal{RS}$. Extensions to the basic model included dynamic process creation and deletion, world-descriptive predicates, and abbreviation predicates. In Chapter 8, we used this methodology to develop an implementation of the SSG system in the $\mathcal{RS}$ model.

Chapter 7 described an application of the $\mathcal{RS}$ model to some common Computer Science and AI structures. Chapter 9 compared $\mathcal{RS}$ to a number of other models in the literature; mainly, the Actor model, the NBS robot system, and CSP/OCCAM.

§2. Implementation

A *distributed robot control environment* was implemented, based on $\mathcal{RS}$. The environment was constructed in Pascal under VMS^{TM} on a VAX^{TM}11/780. The environment consisted of a *compiler* and *emulator* for $\mathcal{RS}$, and a robot simulation package. The implementation of $\mathcal{RS}$ in this environment differs slightly from the model described in this dissertation. The implementation is a subset of $\mathcal{RS}$, and the notation in which programs are written has been geared to a terminal keyboard rather than a laser printer.

Most of the example $\mathcal{RS}$ programs were tested, in one form or another, in this programming environment. The graphic output which appears in figures, in various chapters of this dissertation, was generated on this simulation. The graphics were produced by the robot simulation component of the environment. This simulation could be used with a number of object and robot hand models. The input to the simulation from the $\mathcal{RS}$ emulator was a vector of joint angles, and these were the input to whatever primitive motor schemas had been instantiated. The output of the simulation to the emulator was a vector of joint positions, Cartesian link endpoints, link intersection points with object models (if any; used to emulate tactile sensing), and high-level visual data. The $\mathcal{RS}$ emulator took this information and formatted it as the output of whatever primitive schemas had been instantiated.

§3. Future Research

This dissertation has opened up a number of areas for further work. It is one of the only attempts to say 'what is special' about robotics in computational terms. The goal of this dissertation was to come up with a model of computation for robots — a way of describing and analyzing computation from a robotics point of view. What we have accomplished is simply the start. As well as refining the basic ideas postulated in the dissertation, implementing this model will require looking at computer architectures, looking at distributed scheduling, looking more closely at the details of sensory

preprocessing , etc.

The emphasis in Chapter 5 was to construct a semantics for $\mathcal{RS}$ by which we could understand some of its power. Thus, we chose to construct a very basic version of the `forall` statement, a distinct `time$out` schema, explicit fan-in and fan-out specification, etc. Now that we understand that these extensions can be carried out, a sensible move, from the point of view of synthesis of programs, is to subsume them into a *super-schema* definition, which allows, among other possible extensions, the following:

1. Statements other than just an instantiation in the body of a `forall` loop; the semantics will be understood to be that the body of the loop is an 'implicit' schema definition.

2. Time-out values can be associated directly with ports in the body of the port read or write. Again, the semantics will be understood to be the creation of an explicit `time$out` SI. Alternatively, Observation 30 of Chapter 5 could be used as the semantics of an asynchronous communication mechanism.

One of the weaker parts of $\mathcal{RS}$ is the verification and design methodology. In concept, it is correct; however, the actual mechanisms for expressing schema assertions and for verifying them are too complex. Part of this complexity is inherited from the behavior definition of the Nguyen et al. trace model. A likely approach here would be to redefine behavior more in terms of the *input-output trace* of the port automaton. As well as simplifying the definition of behavior, this would have the added benefit of a more uniform approach to both schema verification and assemblage verification.

The use of world-descriptive predicates to axiomatize the behavior of the real world is a very useful tool. The primitive schemas of $\mathcal{RS}$ then provide an excellent way to 'involve' these predicates into the description of process behavior. Another profitable avenue of research is to determine a useful set for world-predicates for dealing with real equipment; and to elaborate on concepts such as the *total perception axiom*. In this dissertation, we have adopted a heavily motor-oriented view of complex robot systems; we have neglected many sensory processing issues. This ia one major area in which $\mathcal{RS}$ needs development.

APPENDIX A

ADDITIONAL $\mathcal{RS}$ PROGRAMS

The **reach** task-unit was presented in detail in Chapter 8. This appendix contains the **preshape**, **p$object**, **locate$object**, **test**, and **PVFk** schemas developed in the design section, Section 2.1 of Chapter 6. A brief proof outline is presented with each task-unit.

move$finger: The exact form of this will depend on the physical hand model: For the Salisbury hand model, it will have three input ports, one for each DOF; for the human hand model, it will need four ports, again one for each DOF. For the human model, **move$finger** has an assertion of the form:

$$\begin{aligned} \langle move\$finger\rangle \quad \forall i.\Box(\quad & [\Diamond \bigwedge_{j=1\ldots4} In(f_j)] \wedge \\ & [\bigwedge_{j=1\ldots4} In(f_j) \equiv Out(t) \equiv \bigwedge_{j=1\ldots4} POSITION(FM(i,j), f_j)] \wedge \\ & [In(di)] \wedge \\ & [INST(move\$finger_i) \equiv (|\, di \,| = 0)]\ ) \end{aligned} \tag{A.1}$$

where i is the instantiation number, and where $FM(i,j)$ is a *global function* for the particular physical hand model. It maps each of the the input ports (f_j) of an instantiation i of $move\$finger$ to particular degrees of freedom on a finger. For example, from Chapter 2, Figure 5, for both hand models model $FM(i,j) = 3*i+j-1$; i.e., f_1 of $move\$finger$ controls the base joint of finger i, which is joint $3*i$; f_2 controls the second joint, which is joint $3*i+1$, etc. The port t is written when the finger is actually at the positions demanded. And again, we have included a deinstantiation port.

pos$finger: reports on the joints positions on a finger. Again, this version is for the human hand model ($j = 1, \ldots, 4$):

$$\forall i.\Box(\ [\bigwedge_{j=1\ldots4} Out(f_j)] \wedge [\bigwedge_{j=1\ldots4} Out(f_j) \equiv POSITION(FM(i,j), f_j)]\ ) \tag{A.2}$$

The locate$object Precondition Schema

<u>locate$object</u>: This precondition schema takes, as input parameter, a description of an object. It searches all instantiations of EOB to find one whose port values fit the object description, by creating and connecting one instance of a *test* schema per EOB instance. If the *test* SIs report back an appropriate EOB, locate$object then creates an instance of its task-unit ($grasp_i$) connected to this EOB_i. This is an extension of Example 22 in Chapter 4. The object description is a vector of values corresponding, in order and meaning, to the ports of the EOB schema.

The test schema definition is:

```
[ test
  Input-Port-List:     ( h1,h2,h3:Vector )
  Output-Port-List:    ( judgment:Integer )
  Variable-List:       ( df1,df2,df3:Vector )
  Behavior:            ( If (h1 =df1)&(h2 =df2)&(h3 =df3)
                           Then judgment:=#I; Stop;
                           Else Stop;
                         Endif; )                        ]
```

$$(test) \quad \forall i.\Box\Big([\Diamond \bigwedge_{j=1...3} In(h_j)] \wedge [\Diamond\neg INST(test)] \wedge \\ [(\bigwedge_{j=1...3} |h_j| > 0) \supset \\ ((\bigwedge_{j=1...3} \ell(h_j) = \ell(df_j)) \equiv \\ (\Diamond Out(judgment) \wedge (\ell(judgment) = i)))] \Big)$$

<u>Proof Outline:</u> We reason informally that the schema description obeys the assertion. The two 'eventually' clauses in the 'always' clause are the easiest places to start. The schema body is a single if statement, which is in an implicit while true do loop (by the defintion of a schema description); hence, its condition will be evaluated infinitely often, causing the h_j ports to be read infinitely often. In each execution of the schema body, either the else or the then part of the if *must* be executed. But notice that Stop occurs in both instances, hence, it must eventually be executed in one of them. Once the read completes (and here we invoke the schema body liveness axiom to say that it will) the traces of all h_j are now of length 1. The antecedent of the implication is now true. If all

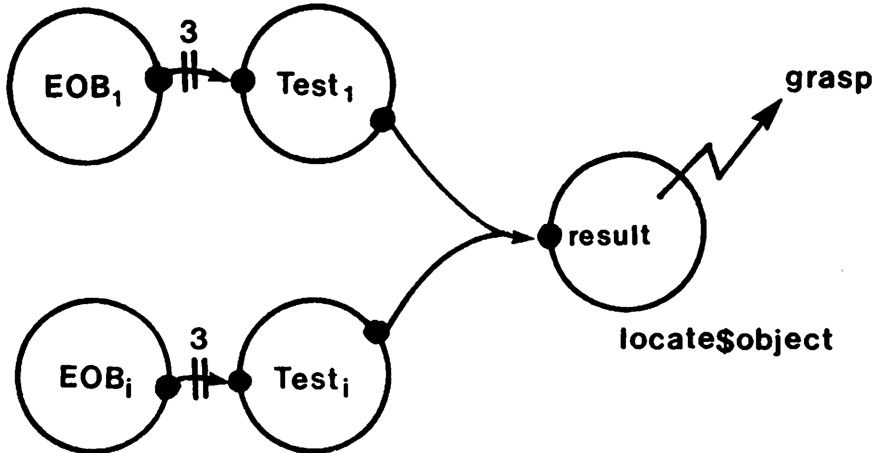
$\ell(h_j) = \ell(df_j)$, then we get $\Diamond out(judgment)$ and the value outputted is the instantiation number i; if at least one $\ell(h_j) \neq \ell(df_j)$, then no value is output.

The `locate$object` schema definition is:

```
[  locate$object
   Input-Port-List:      ( result:Integer )
   Output-Port-List:     ()
   Variable-List:        ( f1...f3:Vector i:Integer)
   Behavior:             ( Forall EOBi:
                            Testi (EOB(f1)...EOB(f3))(result)  (f1...f3);
                           Endforall;
                           i:=result;
                           GRASP(EOBi (f1)...EOBi (f3)) () ()
                           Stop;    )                                  ]
```

$$\Box(\ |\forall EOB_i.INST(EOB_i) \supset INST*(test_i) \wedge \Phi * \wedge C1*| \wedge \ || result | > 1 \supset \Diamond(INST*(grasp) \wedge C2*) \wedge \Diamond \neg INST(locate\$object) | \ ) \quad \text{(A.3)}$$

$$\begin{array}{llcl} C1*: \forall i & test_i(h1) & \leftarrow & EOB_i(f_1) \\ & test_i(h2) & \leftarrow & EOB_i(f_2) \\ & test_i(h3) & \leftarrow & EOB_i(f_3) \\ & locate\$object(result) & \leftarrow & test_i(judgment) \\ C2*: & grasp(g1) & \leftarrow & EOB_{\ell(result)}(f_1) \\ & grasp(g2) & \leftarrow & EOB_{\ell(result)}(f_2) \\ & grasp(g3) & \leftarrow & EOB_{\ell(result)}(f_3) \\ \Phi*: & df1 & \leftarrow & f1 \\ & df1 & \leftarrow & f1 \\ & df1 & \leftarrow & f1 \end{array} \quad \text{(A.4)}$$



<u>Proof Outline</u>: We reason informally that the schema description obeys the assertion. Under the schema body liveness axiom, the schema must reach the last statement in its

body infinitely often; however, this is the deinstantiation command, hence we can say that eventually we shall negate $INST(locate\$object)$. The first clause in the assertion is the direct effect of the `forall` statement from its transition axiom. Note that $INST*$, once asserted, *remains always true*. However, the value of $INST$ can, and usually does, change. The instantiation command for grasp cannot be executed until a value is received over *result*; this is modeled by having the increase in length of the variable, *result* (that is, a communication on the port *result*), imply the creation of the grasp SI. $C1*, C2*$ and $\Phi*$ can be verified by inspection of the instantiation command.

The primitive sensory schema EOB has the assertion:

$$\langle EOB \rangle \forall i. \Box (\ (\bigwedge_{j=1,\ldots,3} Out(f_j)) \wedge WORLDOBJECT(i) \wedge \\ OBJPOS(i, f_1) \wedge OBJANG(i, f_2) \wedge OBJFEA(i, f_3) \ ) \tag{A.5}$$

We want to prove that the network of `locate$object` and $\texttt{test}_i$ and $\texttt{EOB}_i$ SIs has the following assertion:

$$\langle locate\$object \rangle \ WORLDOBJECT(j) \wedge Suitable(j) \supset \\ \Diamond(INST*(grasp) \wedge \neg INST(locate\$object) \\ \wedge OBJPOS(j, g1) \wedge OBJANG(j, g2) \wedge OBJFEA(j, g3)) \tag{A.6}$$

<u>Proof Outline</u>: We use the assemblage/network rule to form the assertion which is the behavior of the network of SIs, from the individual assertions for the `EOB,test` and `locate$object` assertions. Using the renaming rule on these assertions plus the standard rules and theorems for temporal logic, we deduce that conjunction of component assertions implies A.6 above. From A.3, we know that there is one $\texttt{test}_i$ for each $\texttt{EOB}_i$; our perception axiom states that every object in the world is represented by an EOB instance. Hence, if an object in the world exists which satisfies the initialized variables of test df_1, df_2, df_3, as set up by $\Phi*$, then there will be an instance of $\texttt{test}_j$ which will be connected to $\texttt{EOB}_j$, as indicated by $C1*$.

Note that for an EOB we have $\Box \bigwedge Out(f_k)$, and from `test` we have $\Diamond \bigwedge In(h_j)$; hence, our network liveness axiom implies that communication occurs eventually: $\Diamond \mid h_j \mid > 0$. From `test`, once the EOB has transmitted its data, this results in the tranmission of the instantiation number of the successful `test` instance back to `locate$object`. Again, we invoke the liveness axiom to have $\Diamond \mid result \mid > 1$, from which we use A.3 to infer $\Diamond INST*(grasp)$.

In summary, we have shown that $WORLDOBJECT(j) \wedge Suitable(j)$, implies $\Diamond Inst*(grasp)$. Additionally, the value $\ell(result)$ is the instantiation number of the **test** SI for which we had $\bigwedge_{j=1...3} \ell(h_j) = \ell(df_j)$, so by using the **EOB** assertion and the map $C2*$, we can easily show that the ports connected to the **grasp** SI satisfy $OBJPOS(g1, i)$,etc.

The preshape$^{\mathcal{E}}$ Task-Unit

PVF$k^{\mathcal{E}}$: This task-unit translates the actions of the kth virtual finger for the Encompass preshape into the actions of a physical finger. **VFk**$_i$ exists if finger i is in $VFset_k$ for this grasp. Internally, it consists of **angsep**, **space** and **pos$finger** as sensory schemas and **move$finger** as motor schema.

angsep$^{\mathcal{E}}$ is a simple buffer style schema, which just continually evaluates the separation linear equation for this finger (instantiation i) for this grasp g. It accepts input on port d then produces an appropriate finger angle on each of its ports (b_1, b_2, b_3), using the μ_i value from the separation equation for this finger (i). We shall present the assertion directly:

$$
\begin{array}{ll}
\langle angsep \rangle \forall i. \Box (& [In(d) \equiv (\bigwedge_{j=1...3} Out(b_j)) \equiv (\bigwedge_{j=1...3} In(a_j))] \wedge \\
& [\bigwedge_{j=1...3} \ell(b_j) = \ell(d) * \mu_i + \ell(a)] \wedge \\
& [\Diamond In(d)] \quad)
\end{array}
\tag{A.7}
$$

In similar fashion, we shall assume $space_i$ reads an interfinger spacing value on its port d, and produces a base joint angle (ψ_1) to effect this finger separation on port a (details from Chapter 2):

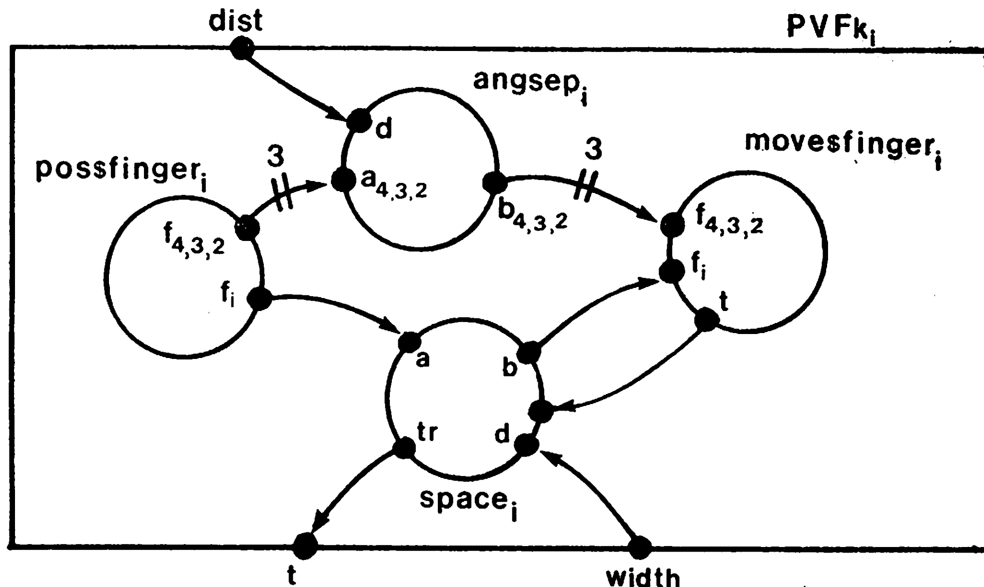
$$
\begin{array}{ll}
\langle space \rangle \ \forall i. \Box (& [In(d) \equiv (| \, d \, | \ = | \, t \, | \,)] \wedge \\
& [Out(b) \equiv In(a) \equiv \neg In(d)] \wedge \\
& [Out(tr) \equiv SPREAD(i, d)] \quad)
\end{array}
\tag{A.8}
$$

The schema definition of the schemas **PVFk**$^{\mathcal{E}}$ is:

```
[  PVFk                 TASK-UNIT
   Input-Port-List:     ( dist:Real width:Real )
   Output-Port-List:    ( t:Integer )
    Sensory-Schemas:     (angsep,space)
    Motor-Schemas:       (move$finger)
   Initialization:      ( angsep(≡dist)(≡t)();
                          space(≡width)()();
                         pos$finger()(angsep(a1)...(a3),space(a))();
                         move$finger  (angsep(b1)...(b3),space(b))
                                                        (space(t))();   )   ]
```

For simplicity, let us dispense with listing the INST* predicates; let us assume them, and just present one network map:

$$\begin{array}{llcl} \forall i & angsep(a_i) & \leftarrow & pos\$finger(f_j) \\ \forall i. & move\$finger(f_i) & \leftarrow & angsep(b_i) \\ & space(a) & \leftarrow & pos\$finger(f_4) \\ & move\$finger(f_4) & \leftarrow & space(b) \\ & space(d) & \equiv & PVFk(width) \\ & space(tr) & \equiv & PVFk(t) \\ & angsep(d) & \equiv & PVFk(dist) \end{array} \qquad (A.9)$$



$$(PVFk) \ \ \forall i.\Diamond(CANFITF(dist,i) \wedge SPREAD(width,i)) \qquad (A.10)$$

Proof Outline: Using the assemblage rule, the assertion for $PVFk_i$ is the conjunction of assertions A.1, A.7 ,A.8 and A.2. From the connection mapping, liveness, and the definition of the angular separation equations, we get $\Diamond(CANFITF(i,width))$ di-

rectly. From our discussion of the interfinger separation algorithm in Chapter 2, we have $\Diamond(out(tr))$. Note that since angsep can assert $CANFITF$ with the first set of joint values to `move$finger`, and that space will assert $SPREAD$ with the first *or some subsequent* joint angles, then $\Diamond(out(tr)) \equiv CANFIT(i, dist) \wedge SPREAD(i, width)$.

Preshape$^{\mathcal{E}}$: The preshape for the Encompass grasp is typical of all the preshapes. The perceptual schema *p$object* analyzes the object characteristics coming into the preshape task-unit through its connection with *g$object*, (port $fea : Vector$). It decides, first of all, how to assign physical fingers to virtual fingers. This is a 2-VF grasp, VF1 is the thumb, and VF2 is the rest of the fingers; we shall have a hand model dependent function $\nu(l)$, which, based upon the length of the object to be grasped, determines how many real fingers are involved in VF2 (as we discussed in Chapter 2). The VFk precondition is then used to set up appropriate instances of $PVF1$ and $PVF2$ from the ν value. All $PVFk_i$ are fan-out connected to two ports VkL and VkW.

`p$object` has the responsibility of calculating the values to send to the virtual fingers, to accomplish appropriate spreading and separation. This is a relatively simple schema, and we will assume the assertion:

$$\begin{aligned}&\Box(Out(V1W) \wedge OBJWIDTH(i, V1W) \wedge Out(V1L) \wedge OBJLENGTH(i, x)\\ &\quad Out(V2W) \wedge OBJWIDTH(i, V2W) \wedge Out(V2L) \wedge OBJLENGTH(i, V2L * 4)\\ &\quad \wedge(x = sqrt(\ell(V2W)^2 - \ell(V1L)^2)))\end{aligned} \quad (A.11)$$

This assertion states that the values sent to the fingers will distribute the fingers around the object's circumference in such a fashion as described in Chapter 2, Section 3.2, to facilitate stable grasping. The definition of x in the last line is to simplify the arguments of the world predicates in previous lines.

```
[  preshape^E          TASK-UNIT
   Input-Port-List:    ( fea:vector )
   Output-Port-List:   ( t:integer )
    Sensory-Schemas:   ( p$object )
    Motor-Schemas:     ( PVF1,PVF2 )
   Initialization:     (For i := 1...ν(fea(2))
                         PVF2( p$object(V2W),p$object(V2L))  (||+≡t))();
                        Endfor;
                        PVF1(p$object(V1W),p$object(V1L))  (||+≡t))();
                                )                                 ]
```

where we use $\|^+$ here to denote that the fan-in to the port t has AND-semantics. Since we have not used this before, its purpose is worth repeating: Any read on the port

t of **preshape** will not terminate until all the PVFk t ports have been written. The actual value which will be read from t is the last one written to it by a PVFk; however, we are not interested in the value written. We use AND-semantics here to ensure that a read to t will not terminate until *all* the PVFk SIs have separated and spread their respective fingers, and therefore, written their respective t ports.

Again, we shall skip listing the instantiation predicates and maps, and go right to the assertion we want to associate with **preshape**:

$$\langle preshape^{\mathcal{E}} \rangle \; \Diamond(CANFIT(i) \wedge (Acquired(i) \supset Stable(i))) \qquad \text{(A.12)}$$

Proof Outline: From the assemblage rule, we conjunct all the PVFk_j instantiation assertions, and that of `p$object`. From each PVFk_j, we know that when we read its tr port we can assume $CANFITF(i, dist) \wedge SPREAD(i, width)$. The AND-semantics fan-in at t computes the $CANFIT$ predicate by ANDing all t ports for all virtual fingers, for all fingers, as described by VIII.21. Since we have established $\Diamond Out(tr)$ for each port, we can say $\Diamond Out(sum)$ and thus $\Diamond CANFIT(i)$. However, we can also establish that each finger has achieved the $SPREAD$ predicate, and by our stable grasping heuristic, we can assert, therefore, $\Diamond(Acquired(i) \supset Stable(i))$.

APPENDIX B

GLOSSARY

The glossary is divided into three sections. In the first section, the notation and terminology used in describing the grasping and manipulation framework is presented. In the second section, the notation and terminology used to describe the model and its formal semantics is presented. In the final section, the notation and terminology of the temporal logic verification method is presented. Each section is organized as follows: Firstly, the symbols are defined in order of explanatory convenience. Secondly, the key phrases and definitions are summarized in alphabetical order.

§1. The Grasping and Manipulation Terminology

$\mathbf{F}$: A coordinate system.

$\mathbf{T}$: A homogeneous transformation matrix.

$\mathbf{F}_b$: The world coordinate system.

$\mathbf{F}_o$: The coordinate system based in the object model.

$\mathbf{T}_o$: The homogeneous transformation matrix relating the object coordinate system to the world coordinate system.

$\mathbf{F}_{iw}$: The coordinate system based in the wrist of hand model i.

$\mathbf{T}_{bw}$: The homogeneous transformation matrix relating the wrist coordinate system to the world coordinate system.

$\mathbf{F}_{ipj}$: The preshape coordinate system for grasp j on hand model i.

$\mathbf{T}_{ipj}$: The transformation matrix relating $\mathbf{F}_{ipj}$ to the wrist coordinate system.

$\mathbf{F}_{of}$: The grasp-site coordinate system, the coordinate system based in the object, which specifies which part of the object to be grasped and in what orientation the object is to be grasped.

$\mathbf{T}_f$: The transformation matrix describing the grasp-site relative to the object-centered coordinate frame (high-level input to the grasping process).

$\mathbf{VF}_i$: Virtual Finger i.

$\mathbf{VFset}_i$: The set of physical finger indices of the fingers in virtual finger i.

l_o, h_o, w_o : Object length, width and height measurements in $\mathbf{F}_o$.

Acquisition Component, A_g : That part of the grasp which describes strategies for acquiring the object, once the preshaped hand has been positioned in the target object vicinity.

Angular Separation Equations : The equations embodying a least-square linear fit between finger-tip/thumb-tip separation and joint angles, for a given finger on a given hand model.

Approach Axis : The trajectory with which the hand must approach the object in order to be able to acquire the object.

Dextrous Hand : A robot hand which can manipulate a grasped object.

Grasp (P_g, A_g, M_g) : The description of a domain of interaction between the hand and a grasped object. It is completely described by its preshape, P_g, acquisition, A_g, and manipulation, M_g, components.

Grasp Selection : The process of choosing a grasp based on the task requirements and target object description.

Preshape Component, P_g : That part of the grasp which configures the hand in preparation for object acquisition.

Reach : That phase of the grasp where the preshaping hand is brought to the vicinity of the object.

Manipulation Component, M_g : That part of the grasp which describes how a gripped object can be manipulated.

Set of Simple Grasps, SSG : The Encompass, Lateral and Precision grasps, and their associated selection mechanisms.

Stability : Static: The sum of the forces and moments on the grasped object are zero. Dynamic: When displaced a small amount, the grasped object will experience forces tending to restore it to its initial position.

Target Object : The object to be grasped.

Task Requirements : The constraints placed on the choice of grasp by the operations to be accomplished with the target object, once it has been acquired.

Virtual Fingers : Logical units of grasp description.

§2. The $\mathcal{RS}$ model Terminology

$\|_c(A_i, B_j, \ldots)$ **:** The SIs A_i, B_j, etc., are all concurrently active and connected together by the network map C. When enclosed in square brackets, i.e., $[\|_c(A_i, B_j, \ldots)]$, this denotes that the listed SIs are an assemblage.

#I : A schema operation which returns the instantiation number of the SI which executes it.

≡ : When placed before port names in a schema instantiation instruction, this means that the indicated (by position) port on the new SI is to be considered equivalent to the assemblage port whose name follows this symbol.

$\|, \|^+$ **:** When placed between port names in an instantiation instruction, this means that there is fan-in/out (depending on whether it is an input or an output port) at the indicated (by position) port. The fan-in/out has OR-semantics if $\|$ is used, and AND-semantics if $\|^+$ is used.

Assemblage : An assemblage is a computing agent whose behavior is defined in terms of the interaction of a network of SIs.

Fan-in : Fan-in occurs when more than one output port is connected to the same input port. Unless otherwise stated, the Fan-in has OR-semantics.

Fan-out : Fan-out occurs when more than one input port is connected to the same output port. Unless otherwise stated, the Fan-out has OR-semantics.

Forall : A schema operation which causes some instantiation instruction to be executed once for each instantiation of some specified index schema which exists at the time of execution.

Instantiation Number : A number associated with a computing agent upon its creation. These numbers are scoped by assemblages, and within the assemblage the schema name and instantiation number are a unique name for the computing agent.

Motor Schema/SI : A schema/SI, or set of schemas/SIs, which describe the basic motor actions available in some task-unit.

Network (Connection) Map : A relation between ports on a set of SIs. defined in terms of the Steentrup et al.'s bijective port connection map.

Port Array : A port array is an indexed set of ports. Port arrays can be dynamic in size.

Port Automaton : A non-deterministic sequential machine equipped with a set of ports. A version of this machine with unidirection ports (uPA) is used to construct the semantics of $\mathcal{RS}$.

Ports : A set of named and typed communication objects associated with each schema. Communication occurs by synchronous message passing over connected ports.

Perceptual Schema/SI : A schema/SI, or set of schemas/SIs, which is (are) the object model for some task-unit.

Precondition Schema/SI : A schema/SI, associated with a task-unit, which is responsible for instantiating the task-unit under some condition.

Primitive Schema : The set of primitive schemas are predefined schemas which interface $\mathcal{RS}$ to the robot and to the world. The internals and general description of these schemas are implementation dependent.

Schema : A generic description of a computing agent. Schema names are written in `teletype` font.

Schema Description : The specification of a schema according to the syntax described in Chapters 4 and 5.

Schema Instantiation, SI : A computing agent, constructed from a schema by using the instantiation operation. An SI name is written as a schema name subscripted with an instantiation number.

Shorthand Notation : A notation developed to allow the top-down development of robot programs in $\mathcal{RS}$. It is not a programming notation, and it can only be used to assist in the development of full schema descriptions.

Task-Unit : A task-unit is a structured assemblage, designed to be used as the unit of robot programming. It is written in $\mathcal{RS}$ shorthand notation as $pre : [P \overset{c}{\longleftrightarrow} M]$; where, pre is called the precondition SI, P is a set of perceptual SIs, and M is a set of motor SIs.

§3. The Temporal Logic Terminology

δ, δ_i **:** An error/precision range.

ς **:** An assertion for a schema or schema instantiation.

s_i **:** State i in a temporal logic universe.

$\Box, \Diamond, \bigcirc, \mathcal{U}, \mathcal{N}$ **:** The temporal logic modal operators. They are called: ***always, eventually, next, until*** and ***unless*** respectively.

$\langle P \rangle \;\; \varsigma$ **:** The specification operator, meaning that ς is an assertion for schema/SI P.

$x = a_0, a_1, \ldots, a_n$ **:** A variable x which is a sequence, also denoted $x = [a_0, a_1, \ldots, a - n]$.

$\ell(x)$ **:** The last value in the sequence x, also denoted $x(n)$.

$\mid x \mid$ **:** The length of the sequence x.

$F_v(d, x)$ **:** A function designed to emulate a critically damped position servo system with setpoint d and current position x.

Abbreviation Predicate : A short 'name' for an assertion not yet completely developed; a top-down design tool.

Assertion Template : An assertion in temporal logic associated with a schema S. All behaviors of an instantiation of S must satisfy this assertion.

$INST(S_i)$ **:** The special instantiation predicate, true iff S_i has been instantiated and not yet deinstantiated.

$INST*(S_i)$ **:** A version of the instantiation predicate augmented to include the assertion of any assertion template for S with the global variables bound (in particular with the instantiation number bound to i).

In, Out **:** The port communication functions.

Liveness Axiom : An axiom which allows us to conclude that computing agents will actually make progress in their computations.

Local/Global Symbols : Local symbols can change their meaning or value in each state; global symbols have a uniform interpretation in all states.

Perception Axiom : An axiom which allows us to conclude that the robot has a satisfactory model of the environment, thus circumventing the necessity for constructing complex sensory search programs.

Transition Axiom : An axiom which embodies the effect of executing a particular program statement.

World-Descriptive Predicates : Predicates which reflect the state of the robot's environment.

BIBLIOGRAPHY

[1] Alagić, S., and Arbib, M.A., *The Design of Well-Structured and Correct Programs,* Springer-Verlag, 1978.

[2] Albus, J., MacLean, C., Barbera, A., and Fitzgerald, M., "Hierarchical Control for Robots in an Automated Factory," *Proceedings, 13th ISIR*, Chicago, Ill., Apr., 1983, pp.13.29–13.43.

[3] Andrews, G.R., "Synchronizing Resources," *ACM Trans. Prog. Lang. and Sys.*, Vol.3, No.4, Oct., 1981, pp.405–430.

[4] Andrews, G.R., and Schneider, F.B., "Concepts and Notations for Concurrent Programming," *Computing Surveys*, Vol.15, No.1, Mar., 1983, pp.3–43.

[5] Arbib, M.A., "Parallelism, Slides, Schemas, and Frames," in (W.E. Hartnett, Ed.) *Systems, Approaches, Theories, Applications*, D.Reidel Publishing Co., 1977, pp.27-43.

[6] Arbib, M.A., "Perceptual Structures and Distributed Motor Control," in (V.B.Brooks, Ed.) *Handbook of Physiology : The Nervous System, II. Motor Control*, American Physiological Society: Bethesda, MD, 1981, pp.1449-1480.

[7] Arbib, M.A., Overton, K.J., and Lawton, D.T., "Perceptual Systems for Robots," *Interdisciplinary Science Reviews*, 9, 1984, pp.31–46.

[8] Arbib, M.A., Iberall, A., and Lyons, D., "Coordinated Control Programs for Movements of the Hand," *Exp. Brain Res. Suppl.*, 10, 1985, pp.111-129.

[9] Asada, T., "Object Handling System for Manual Industry," *IEEE Trans. SMC*, Vol.SMC-9, No.2, Feb., 1979, pp.79–89.

[10] Baker, B.S., Fortune, S.J., and Grosse, E.H., "Stable Prehension with a Multi-Fingered Hand," *Proceedings, IEEE International Conference on Robotics and Automation*, St. Louis, MO, Mar. 25-28, 1985, pp.570–575.

[11] Barbera, A., "Microcomputer Networks and Robot Interfaces," in (J.Albus, Ed.) *Discussions on Robotics*, Unpublished Lecture Notes, 1982.

[12] Barr, A., and Feigenbaum, E., *The Handbook of Artificial Intelligence*, HeurisTech Press, 1981.

[13] Bloom, H., Furlanz, C., and McLean, C., "Emulation as a Tool in the Design of Factory Automation Systems," *Proceedings, IEEE Workshop on Languages for Automation*, Nov., 1984, pp.60–66.

[14] Blume, C., and Jakob, W., "Design of the Structured Robot Language," in (A. Danthine and M. Geradin, Eds.) *Advanced Software in Robotics*, Elsevier Science Publishers B.V. (North-Holland), 1984, pp.127–143.

[15] Boissonnat, J., "Stable Matching Between a Hand Structure and an Object Silhouette," *IEEE Trans.PAMI*, Vol.PAMI-4, No.6, Nov., 1982, pp.603–612.

[16] Brooks, S.D., Hoare, C.A.R., and Roscoe, A.W., "A Theory of Communicating Sequential Processes," *JACM*, 31,3, Jul., 1985, pp.560–599.

[17] Cheriton, D.R., "Thoth, a Portable Real-Time Operating System," *CACM*, Vol.22, No.2, Feb., 1979, pp.105–115.

[18] Clinger, W.D., "Foundations of Actor Semantics," *Technical Report* 633, MIT AI Lab, Cambridge, MA, May, 1981.

[19] Corkill, D.D., and Lesser V.R., "A Goal Directed Hearsay-II Architecture: Unifying Data-Directed and Goal-Directed Control," *Technical Report* 81-15, Department of Computer and Information Science, University of Massachusetts, Amherst, MA, Jun., 1981.

[20] Crossley, F.R., and Umholtz, F.G., "Design for a Three Fingered Hand," *Mechanisms and Machine Theory*, Vol.12, No.1, 1977, pp.85–93.

[21] Cutkosky, M., "Grasping and Fine Manipulation for Automated Manufacturing," *Ph.D. Dissertation*, Dept. of Mech. Eng., Carnegie-Mellon University, Pittsburg, PA, Sept., 1985.

[22] Davis, R., "Report on the Workshop on Distributed AI," *SIGART*, No. 73, Oct., 1980, pp.42 52.

[23] Dijkstra, E.W., *A Discipline of Programming*, Prentice-Hall: New Jersey, 1976.

[24] Fearing, R.S., "Simplified Grasping and Manipulation with Dextrous Robot Hands," *AI Lab Memo 809*, MIT, Cambridge, MA, Nov., 1984.

[25] Fikes, R.E., Hart, P.E., and Nilsson, N.J., "Some New Directions in Robot Problem Solving," *Machine Intelligence*, Vol.3, 1972, pp.405–430.

[26] Geschke, C., "A System for Programming and Controlling Sensor-Based Robot Manipulators," *IEEE Trans. on PAMI*, Vol. PAMI-5, No. 1, Jan., 1983, pp.1–7.

[27] Gruver, W.A., Soroka, B.I., Craig, J.J., and Turner, T.L., "Industrial Robot Programming Languages: A Comparative Evaluation," *IEEE Trans. SMC*, Vol. SMC-14, No.4, Jul./Aug., 1984, pp.565–571.

[28] Hanafusa, H., and Asada, H., "Stable Prehension by a Robot Hand with Elastic Fingers," *Proceedings, 7th ISIR*, Tokyo, Japan, Oct., 1977, pp.361–368.

[29] Henderson, T.C., Wu, S.F., and Hansen, C., "MKS: A Multisensor Kernel System," *IEEE Trans.SMC*, Vol.SMC-14, No.5, Sep./Oct., 1984, pp.784–791.

[30] Hewitt, C., "Viewing Control Structures as Patterns of Passing Messages," *Artificial Intelligence*, 8, 1977, pp.323-364.

[31] Hillis, W.D., "The Connection Machine," AI Lab Memo 646, MIT, Cambridge, MA, Sept., 1981.

[32] Hoare, C.A.R., *Communicating Sequential Processes*, Prentice-Hall International Series in Computer Science, 1985.

[33] Hollerbach, J.M., and Sahar, G., "Wrist-Partitioned Inverse Kinematic Accelerations and Manipulator Dynamics," *AI Lab Memo* 717, MIT, Cambridge, MA, Apr., 1983.

[34] Iberall, T., and Lyons, D.M., "Towards Perceptual Robotics," *IEEE Conference on Systems, Man and Cybernetics*, Nova Scotia, Oct. 9-12, 1984.

[35] Iberall, T., Bingham, G., and Arbib, M.A., "Opposition Space as a Structuring Concept in the Analysis of Skilled Hand Movements," *Technical Report* 85-19, Department of Computer and Information Science, University of Massachusetts, Amherst, MA, 1985.

[36] Ingalls, D.H.H., "The Smalltalk-76 Programming System: Design and Implementation," *5th Annual ACM Symp. on Principles of Programming Languages*, Jan., 1978, pp.9-15.

[37] INMOS, *IMS T424 Transputer*, Preliminary Data, 1985.

[38] Ish-Shalom, J., "The CS Language Concept: A New Approach to Robot Motion Design," *International Journal of Robotics and Automation*, Vol.4, No.1, Spring, 1985, pp.42–58.

[39] Jacobson, S.C., Wood, J.E., Knuti, D.F., Biggers, K.B., and Iverson, E.K., "The Version I Utah/MIT Dextrous Hand," *Proceedings, 5th CISM-IFToMM*, Jun., 1984, pp.1–8.

[40] Kent, E., Shneier, M., Hong, T., "Building Representations From Fusions of Multiple Views," *Proceedings, IEEE Conference on Robotics and Automation*, 1986, pp.1634–1639.

[41] Kerr, J., and Roth, B., "Analysis of Multifingered Hands," *IJRA.*, Vol.4, No.4, Winter, 1986, pp.3–17.

[42] Kerridge, J.M., and Simpson, D., "Three Solutions for a Robot Arm Controller Using Pascal-Plus, OCCAM and Edison," *Software – Practice and Experience*, Vol.14, 1984, pp.3–15.

[43] Kfoury, A.J., Moll, R.N., and Arbib, M.A., *A Programming Approach to Computability*, Texts and Monographs in Computer Science, Springer-Verlag: New York, Heidelberg, Berlin, 1982.

[44] Kornfeld, W.A., and Hewitt, C., "The Scientific Community Metaphor," *IEEE Trans. SMC*, Vol. SMC-11, No. 1, Jan., 1981, pp.24–33.

[45] Laugier, C., and Pertin, J., "Automatic Grasping: A Case Study in Accessibility Analysis," in (A. Denthine and M.Geradin, Eds.) *Advanced Software in Robotics*, Elsevier Science Publishers, B.V. (North-Holland), 1984, pp.201–214.

[46] LeBlanc, T.J., and Friedberg, S.A., "HPC: A Model of Structure and Change in Distributed Systems," *IEEE Trans.Computer*, Vol.C-34, No.12, Dec., 1985, pp.1124–1129.

[47] Li, Z., and Shastry, S., "Task Oriented Optimal Grasping by Multifingered Robot Hands," *Memo UCB/ERL M86/43*, Electronics Research Laboratory, University of California at Berkeley, May, 1986.

[48] Lieberman, L.I., and Wesley, M.A., "AUTOPASS: An Automated Programming System for Computer Controlled Mechanical Assembly," *IBM J.Res.Dev*, Vol.21, No.4, 1977, pp.321–333.

[49] Lozano-Perez, T., "Task Planning," in (Brady, M., Hollerbach, J., Johnson, T., Lozano-Perez, T., Mason, M., Eds.) *Robot Motion Planning and Control*, MIT Press, Cambridge, MA, and London, England, 1983.

[50] Lozano-Perez, T., "Robot Programming," *AI Lab Memo 698*, MIT, Cambridge, MA, Dec., 1982.

[51] Lozano-Perez, T., and Brooks, R., "An Approach to Automatic Robot Programming," *AI Memo 842*, MIT, Cambridge, MA, Apr., 1985.

[52] Luh, J.Y.S., "An Anatomy of Industrial Robots and Their Control," *IEEE Trans. Automatic Control*, Vol.AC-28, No.2, Feb., 1978., pp.133–153.

[53] Lumnia, R., Shneier, M., and Kent, E., "Real-Time Iconic Image Processing," *Proceedings, IEEE Conference on Robotics and Automation*, St. Louis, MO, Mar. 25–28, 1985, pp.873–878.

[54] Lyons, D.M., "A Simple Set of Grasps for a Dextrous Hand," *Proceedings of the 1985 International Conference on Robotics and Automation*, St. Louis, MO, Mar. 25–28, 1985, pp.588–593.

[55] MacQueen, D.B., "Models for Distributed Computing," *Rapport de Recherche*, 351 RIA Laboria, Rocquencourt, France, Apr., 1979.

[56] Manna, Z., "Verification of Sequential Programs: Temporal Axiomitization," *Technical Report* STAN-CS-81-877, Department of Computer Science, Stanford University, Stanford, CA, 1981.

[57] Manna, Z., and Pnueli, A., "Verification of Concurrent Programs: A Temporal Proof System," *Technical Report* STAN-CS-83-967, Department of Computer Science, Stanford University, Stanford, CA, Jun., 1983.

[58] Mason, M.T., "Compliance and Force Control for Computer Controlled Manipulators," *IEEE Trans. SMC*, Vol.SMC-11, No.6, Jun., 1981, pp.418–432.

[59] May, D., "OCCAM," *SIGPLAN Notices*, 18,14, Apr., 1983, pp.69–80.

[60] Minsky, M., "A Framework for Representing Knowledge," in (P.Winston, Ed.) *The Psychology of Computer Vision*, McGraw-Hill, 1975, pp.211–277.

[61] Misunas, D.P., "A Computer Architecture for Data-Flow Computation," *Technical Report* MIT/LCS/TM-100, Department of Computer Science and Electrical Engineering, MIT, Cambridge, MA, 1978.

[62] Mujtaba, S., and Goldman,R., "AL Users' Manual," *Technical Report* STAN-CS-81-889, Department of Computer Science, Stanford University, Standford, CA, Dec., 1981.

[63] Neisser, U., *Cognition and Reality: Principles and Implication of Cognitive Psychology*, San Francisco: Freeman, 1976.

[64] Nguyen,V., "Constructing Force Closure Grasps," *Proceedings, IEEE Conference on Robotics and Automation*, San Francisco, CA, 1986, pp.1368–1373.

[65] Nguyen V., Gries, D., and Owicki, S., "A Model and Temporal Proof System for Networks of Processes," *Technical Report* TR 84-651, Department of Computer Science, Cornell University, Nov., 1984.

[66] Nguyen V., Gries, D., and Owicki, S., "A Model and Temporal Proof System for Networks of Processes," *Distributed Computing*, 1, 1986, pp.7–25.

[67] Noyes, T.A., "Robot Programming Languages: A Study and Design," *Technical Report 83-22*, Department of Computer and Information Science, University of Massachusetts, Amherst, MA, Oct., 1983.

[68] Ousterhout, J.K., Scelza, D.A., and Sindhu, P.S., "Medusa: An Experiment in Distributed Operating System Structure," *CACM*, Vol.23, Feb., 1980, pp.92-105.

[69] Overton, K.J., "The Acquisition, Processing, and Use of Tactile Sensor Data in Robot Control," *Ph.D. Dissertation*, Department of Computer and Information Science, University of Massachusetts, Amherst, MA, May, 1984.

[70] Paul, R.P., *Robot Manipulators: Mathematics, Programming and Control*, MIT Press: Cambridge, MA, and London, England, 1982.

[71] Piaget, J., *The Psychology of Intelligence*, Littlefield and Adams & Co.: Totowa, NJ, 1981.

[72] Popplestone, R.J., Ambler, A.P., and Bellos, I., "A Language for Describing Assemblies," University of Edinburgh, Department of Artificial Intelligence, *Research Paper 79*, Sept., 1978.

[73] Raibert, M., "Legged Locomotion — the Robotics of Running," Invited Presentation, *SIAM Conference on Geometric Modelling and Simulation*, Albany, NY, 1985.

[74] Ramamritham, K., Pocock, G., Lyons, D.M., and Arbib, M.A., "Towards Distributed Robot Control Systems," *IFAC Symposium on Robot Control*, Barcelona, Spain, Nov. 6-8th, 1985, pp.209-213.

[75] Rescher, N., and Urquhart, A., *Temporal Logic*, Springer-Verlag: New York, Wien, 1971.

[76] Salisbury, J.K., "Kinematic and Force Analysis of Articulated Hands," *Ph.D. Dissertation*, Department of Mech. Eng., Stanford University, Stanford, CA, 1982.

[77] Schank, R.C., Abelson, R.P., *Scripts, Plans, Goals and Understanding*, Lawrence Earlbaum Assoc.: Hillsdale, NJ, 1977.

[78] Shneier, M., Kent, E., Albus, J., Mansbach, P., Nashman, M., Palomba, L., Rutkowski, W., and Wheatley, T., "Robot Sensing for a Hierarchical Control System," *Proceedings, 13th ISIR*, 1983, pp.14.50–14.66.

[79] Smith, R.G., "The Contract Net Protocol: High-Level Communication and Control in a Distributed Probem Solver," *IEEE Trans. on Computers*, Vol. C-29, No. 12, Dec., 1980, pp.1104–1113.

[80] Steenstrup, M., Arbib, M.A., and Manes, E.G., "Port Automata and the Algebra of Concurrent Processes," *Journal of Computer and System Sciences*, Vol. 27, No. 1, Aug., 1983, pp.29–50.

[81] Taylor, R., "A Synthesis of Manipulator Control Programs from Task-Level Specifications," *Technical Report* STAN-CS-76-560, Department of Computer Science, Stanford University, Stanford, CA, Jul., 1976.

[82] *The VAL Reference Manual.* Unimation, 1978.

[83] Theriault, D., "A Primer for the ACT-1 Language," *AI Lab Memo* 672, MIT, Cambridge, MA, May, 1981.

[84] Theriault, D., "Issues in the Design and Implementation of ACT-2," *Technical Report* 788, AI Lab, MIT, Cambridge, MA, Jun., 1983.

[85] Tsukune, H., "A Formalization of the Cooperative Activity of the Sensor Motor System," *Bul. Electrotech. Lab.*, Vol.43, Nos. 7&8, 1979, pp.34–53.

[86] Weymouth, T., "Schema-Guided Visual Interpretation," *Ph.D. Dissertation*, Department of Computer and Information Science, University of Massachusetts, Amherst, MA, Jun., 1985.

[87] Will, P.M., and Grossman, D.D., "An Experimental System for Computer Controlled Manipulators," *IEEE Trans. Computing*, Vol.C-24, No.9, 1975, pp.879–888.

[88] Wolter, J., Volz, R., and Woo, A.C., "Automatic Generation of Gripping Positions," *IEEE Trans. SMC*, Vol. SMC-15, No.2, Mar./Apr., 1985, pp.204–213.

[89] Vijaykumar, R., and Arbib, M.A., "Dynamic Planning for Sensor-Based Robots," *IFAC Symposium on Robot Control*, Barcelona, Spain, Nov. 6-8th, 1985, pp.401–406.

[90] Zakarov, V., "Parallelism and Array Processing," *IEEE Trans. Computer*, Vol.C-33, No.1, Jan., 1984, pp.43-77.