EXPERIMENTS IN THE INTEGRATION AND CONTROL OF AN INTELLIGENT MANUFACTURING WORKCELL

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Gerardo Pardo-Castellote
June 1995

Copyright © 1995 by Gerardo Pardo-Castellote
All Rights Reserved.

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Robert H. Cannon, Jr.
Department of Aeronautics and Astronautics
(Principal Adviser)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Gene F. Franklin
Department of Electrical Engineering

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Jean-Claude Latombe
Department of Computer Science

Approved for the University Committee on Graduate Studies:

Abstract

This thesis comprises the experimental development of an intelligent, dual-arm robotic workcell. The system combines a high-level graphical user interface, an on-line motion planner, real-time vision, and an on-line simulator to control a dual-arm robotic system from the task level.

The graphical user interface provides for high-level user direction of the task to be done. The motion planner generates complete on-line plans to carry out these directives, specifying both single and dual-armed motion and manipulation. Combined with the robot control and real-time vision, the system is capable of performing object acquisition from a moving conveyor belt as well as reacting to environmental changes on-line.

The thesis covers in detail four main topics:

1. System design and interfaces. The system is based on a novel "interface-first" design technique. This technique structures the complex command and data flow as combinations of three fundamental robotic interface components: the world-state interface, the robot-command interface, and the task-level-direction interface.

2. Network communication architecture. Complex distributed robotic systems require very complex data flow. A powerful new subscription-based, network-data-sharing system, was developed (and is being commercialized) that enables transparent connectivity.

3. Control system. The architecture and design of the hierarchical control system for the experimental dual-arm assembly workcell is described.

4. Path time-parameterization. A fast (linear-time, proximate-optimal) solution to the fundamental problem of time-parameterization of robot paths is presented.

The design was verified experimentally in a dual-arm robotic workcell. Experimental results are presented showing the system performing complex, multi-step tasks autonomously, including dual-arm object acquisitions from a moving conveyor, object motion among obstacles, re-grasps, and hand-overs. All these tasks occur under task-level human supervision.

To my parents

Acknowledgements

I wish to thank my principal advisor, Professor Robert H. Cannon Jr., for his guidance, encouragement, and support during the course of the research. His exemplary personality and technical insight have made graduate research fruitful and enjoyable. I was extremely fortunate to have him as an advisor.

I thank my associate advisor, Professor Gene Franklin for his perceptive comments and accommodating my last-minute needs.

I am indebted to my associate advisor, Professor Jean-Claude Latombe for his professional and personal advice from the moment I arrived to Stanford. He first guided me during my Master in Computer Science and has since influenced and supported my career in uncountable occasions.

I am further indebted Stan Schneider for convincing me to join the Aerospace Robotics Laboratory and being a constant source of friendship, advice (technical and otherwise), and encouragement. I have greatly benefited from the legacy of his work.

The Aerospace Robotics Laboratory has always been a friendly and stimulating research environment. I thank Professors Robert Cannon and Steve Rock for creating and directing it, Jane Lintott for administering it with such skill, and all my fellow students for their good-natured attitude and help. Among them I would like to specially thank Larry Pfeffer for developing the experimental hardware and his extensive advice on the design of the control system; Denny Morse for his selfless and timely assistance on computer subjects; Marc Ullman for his lessons on real-time computers, and general advice in all subjects; Dave Meer and Steve Ims for their dedication to administering the lab computer system; Stef Ssonck, Kurt Zimmermann, Howard Wang, David Miles, and Jeff Russakow for their patient review of parts of this thesis and constructive comments.

I extend my thanks to the ARL technical staff. Gad Shelef for his help in the design of the modifications to the manipulator hardware, and Godwin Zhang for his efforts and able assistance in maintaining the electronics of the workcell.

Financial support for this research was provided by a Fellowship form the Spanish Ministry of Education and ARPA contract N00014-92-J-1809.

Finally, I would like to thank my parents for implanting in me the desire to learn, stressing and contributing to my education, giving me freedom to conduct my life from and early age, and always being respectful, understanding, and supportive. I really could not have asked for anything more. It is to them that I dedicate this thesis.

Contents

Abstract iv

Acknowledgements vi

List of Tables xii

List of Figures xiii

1 Introduction 1

1.1 Motivation 1
1.2 Research Objectives 3
1.3 Research Issues 4
1.4 Summary of Results 7
1.4.1 System Capabilities and Demonstrations 7
1.4.2 Contributions 9
1.5 Related Research Activities 12
1.5.1 Supporting Research Activities 13
1.6 Reader's Guide 15

2 Experimental Dual-Arm Robotic Workcell 16

2.1 Introduction 16
2.2 Manipulator Mechanism and Sensors 16
2.2.1 Kinematics 17
2.2.2 Joint-Torque Sensors 20
2.2.3 Joint Encoders 20
2.3 Workspace Objects 22

2.4 Conveyor 23
2.5 Vision System 24
2.6 Computer System 25

3 Architecture, Interfaces and System Operation 28
3.1 Literature Review 29
3.2 Approaches to System Design 34
3.3 Modular Interfaces 37
3.4 Information Interfaces for the Manufacturing Workcell 38
3.4.1 The World-State Interface 38
3.4.2 The System-Command Interface 39
3.4.3 The Task-Specification Interface 41
3.4.4 Custom Interfaces for the Manufacturing Workcell 42
3.5 System Architecture 43
3.5.1 The Graphical User Interface 44
3.5.2 The Simulator 45
3.5.3 The World Modeler 48
3.5.4 The Hierarchical Control System 49
3.5.5 The Planning Subsystem 49
3.6 Experimental System Operation 51
3.6.1 Operational Description 51
3.6.2 Experimental Tasks 52
3.7 Summary and Conclusions 77

4 Communications in Distributed Robotic Systems 78
4.1 The Role of Communications Within the System 80
4.2 Literature Review: Communications in Distributed Systems 82
4.2.1 The applications' view of the information exchange 83
4.2.2 Implementation aspects 85
4.2.3 Review of Related Approaches 86
4.3 The NDDS Communications Model 90
4.3.1 Producer Characteristics 92
4.3.2 Consumer Characteristics 93
4.3.3 One-time Queries 94

4.3.4 Reliable Updates 95
4.4 Implementation 96
4.4.1 Architectural Overview 96
4.4.2 Data Management Overview 98
4.5 Experimental Results: NDDS 98
4.5.1 Experimental Context 100
4.5.2 Latency of the Updates 101
4.5.3 Maximum update rates 105
4.6 Summary 111

5 World Modeling **113**
5.1 World Modeling for the Manufacturing Workcell 114
5.2 Literature Review: World Modeling for Robotics 116
5.3 Architecture of the World Modeler 119
5.3.1 The Object-Based Layer 121
5.3.2 The Sensor Processing/Integration Layer 122
5.3.3 Characteristics of the Architecture 122
5.4 Sensor Integration 124
5.4.1 Integration of Kinematics and Vision for Robot-Arm Position 125
5.4.2 Robot State and Vision for Grasped-Object Position 131
5.4.3 Conveyor Displacement and Vision for Object Position 132
5.5 Summary 136

6 Hierarchical Control System **137**
6.1 Control System for the Manufacturing Workcell 137
6.2 Literature Review: Hierarchical Robotic Control Systems 138
6.3 Control System Hierarchy 139
6.4 Joint-Control Layer 141
6.5 Arm-Control Layer 147
6.5.1 Arm Controller 147
6.5.2 Arm Controller Performance 151
6.6 Object-Control Layer 151
6.7 Strategic-Control Layer 157
6.7.1 The Finite-State Machine Programming Model 159

6.7.2 Strategic Control of the Workcell Using FSMs 160
6.7.3 Picking an Object from the Conveyor Belt 168
6.7.4 Experimental Pick Operations 170
6.8 Summary 178

7 On-Line Trajectory Generation 179
7.1 The Role of Trajectory Generation within the Workcell 179
7.2 Literature Review: Trajectory Generation 182
7.3 Problem Formulation and Known Results 185
7.3.1 Minimum travel time with limits on actuator torque, acceleration and velocity 186
7.3.2 Case of velocity-independent torque limits 190
7.4 Formulation of the Proximate-Optimal Problem 190
7.5 The Proximate-Optimal Time-Parameterization Algorithm 194
7.5.1 Continuous-Domain version 194
7.5.2 Algorithm correctness 195
7.5.3 Discrete Implementation 197
7.5.4 Complexity of the algorithm 199
7.5.5 Complexity of other approaches 200
7.5.6 Configuration-independent limits on velocities and accelerations 207
7.5.7 Modification of Ongoing Trajectories 208
7.6 Experimental Results: Time Parameterization 212
7.6.1 Straight-line trajectories for a point mass 213
7.6.2 Trajectories for the workcell manipulators 215
7.6.3 Modifications to on-going trajectories 218
7.7 Summary and Conclusions 225

8 Conclusions 226
8.1 Summary and Conclusions 226
8.2 Suggestions for Future Research 232
8.2.1 Extensions to the Architecture and Interfaces 232
8.2.2 Enhancements to the Subsystems 233
8.2.3 Higher Degree of Planning Integration 235
8.2.4 Enhancements to the Communications Layer. 236
8.2.5 Possible Follow-on Experiments. 237

8.3 Concluding Remarks 237

A Calibration **240**

A.1 Vision System Calibration 240

A.2 Arm Base Location Calibration 240

A.3 Inertial Properties Measurement 242

B Joint Control Parameters **244**

C Arm Control Parameters **246**

D Manipulator Equations of Motion **248**

E Canonical Paths for Time-Parameterization **251**

F Worst-Case Paths for Time-Parameterization **253**

G Matrix Plotting Utilities **258**

G.1 Overview 258

G.2 Application Interface 260

G.3 Selected Interface Documentation 261

H Selected Manual Pages **263**

List of Tables

2.1 Arm kinematic parameters 21
2.2 Actuator characteristics 21
2.3 Encoder types and resolution 22
2.4 Characteristics of the workspace objects 24
2.5 Real-Time Computer Components 26
3.1 Characteristics of the information flow 38
3.2 The world state interface 39
3.3 System command interface 40
3.4 Task-specification interface 42
4.1 Functional interface to produce, consume and query data. 91
5.1 World-Model private interface 129
6.1 Summary of control layers and their interfaces. 142
6.2 Notation for arm-impedance controller derivation. 148
6.3 Modes of Operation 164
7.1 Related approaches to the Decoupled-Optimal-Time-Parameterization Problem . . 186
A.1 Calibrated arm parameters 242
B.1 Estimator parameters for the right shoulder motor plant. 244
B.2 Controller and parameters for the right shoulder motor plant. 245
C.1 Parameters for the arm-level controllers. 246
D.1 Meaning of symbols in dynamic equations for the arms. 250

List of Figures

1.1 Research Objective . . . 4
1.2 Experimental System . . . 7
2.1 Experimental Dual-Arm Robotic Workcell . . . 17
2.2 Z and Yaw degrees of freedom . . . 18
2.3 Arm Schematic . . . 18
2.4 Schematic of last Z and Yaw degrees of freedom and actuation . . . 19
2.5 The different objects the robot interacts with . . . 23
2.6 Hardware Architecture . . . 27
3.1 Contrasted Design Techniques . . . 35
3.2 Modular Interface Design . . . 37
3.3 System Architecture . . . 43
3.4 Commanding and Monitoring the Workcell with the GUI (1-6) . . . 46
3.5 Commanding and Monitoring the Workcell with the GUI (7-12) . . . 47
3.6 Commanding and Monitoring the Workcell with the GUI (13-18). . . . 55
3.7 Commanding and Monitoring the Workcell with the several User Interfaces . . . 56
3.8 Concept of Safe-Under-Time-Delay (SUTD) planning . . . 57
3.9 Use of configuration-time space for safe-under-time-delay (SUTD) planning . . . 58
3.10 System Configurations . . . 59
3.11 Typical system operation . . . 60
3.12 Animation of capture and delivery . . . 61
3.13 Animation of hand-over operation . . . 62
3.14 Photographs during hand-over operation . . . 63
3.15 Animation of multi-object-placing sequence with static objects (stages 1-8) . . . 64
3.16 Animation of multi-object-placing sequence with static objects (stages 9-14) . . . 65
3.17 Photographs during multi-object-placing sequence with static objects (stages 1-6) 66

3.18 Photographs during static assembly sequence (stages 7-10) 67
3.19 Photographs during static assembly sequence (stages 11-14) 68
3.20 Animation of multi-object-placing sequence with objects delivered on the conveyor 70
3.21 Photographs during multi-object-placing sequence with with objects delivered on conveyor 71
3.22 Animation of multi-object-placing sequence with different conveyor position (stages 1-6) 72
3.23 Animation of assembly sequence with conveyor across the workspace (stages 7-14) 73
3.24 Detailed strobe images of an multi-part-placement operation (1-15) 74
3.25 Detailed strobe images of an multi-part-placement operation (16-30) 75
3.26 Detailed strobe images of an multi-part-placement operation (31-45 76
4.1 Architecture of the Two-Armed robotic workcell. 80
4.2 Multiple producer conflict resolution. 93
4.3 Consumer notification rate. 94
4.4 One-Time Query Parameters. 95
4.5 Communication between NDDS nodes. 97
4.6 Latency overhead of NDDS as a function of item size. 101
4.7 Latency overhead as a function of the number of items 103
4.8 Latency overhead as a function of the number of items for two consumers 104
4.9 Latency overhead as a function of the number of items for four consumers 105
4.10 Maximum update rate as a function of the number of subscribers. 106
4.11 Influence of number of items requested on the maximum update rate. 108
4.12 Influence of computer performance and update size on the maximum update rate. . 109
4.13 Maximum update rate for two clients as a function of the item sizes. 110
4.14 Maximum update rate for four clients as a function of the item sizes.. 111
5.1 Roles of the World Modeling Subsystem 115
5.2 Architecture of the World Modeling Subsystem 119
5.3 Arms and rigid-objects are represented using software objects. 121
5.4 Comparison of WM architecture with purely hierarchical WM 123
5.5 On-Line estimator of initial angular offsets 127
5.6 Performance of On-Line angular-offset estimator 128
5.7 Adaptation to power-up configuration 130
5.8 Sensor Fusion for the position of grasped objects 131

5.9 Sensor integration scheme for objects on a conveyor 133
5.10 Tracking performance of objects on the conveyor at 5 cm/sec 134
5.11 Tracking performance of objects on the conveyor at 10 and 20 cm/s 135
6.1 Hierarchical Control Subsystem 139
6.2 Hierarchical Control Dataflow 140
6.3 Experimental frequency response for the motor-drive-train-link plant of the right arm. 144
6.4 Closed-loop roots for the motor-drive-train-link plant and controller of the right shoulder. 145
6.5 Joint-control layer dataflow. 145
6.6 Joint-torque step response of left shoulder subsystem. 146
6.7 Block diagram for the computed-torque impedance controller. 149
6.8 Merging a cartesian-space computed-torque controller with a joint-space PID controller. 149
6.9 Arm-control layer dataflow. 152
6.10 Experimental tracking performance: straight-line path 153
6.11 Experimental tracking performance: path through kinematic singularity. 154
6.12 Animation of trajectory through kinematic singularity. 155
6.13 Comparison of cooperative object control with coordinated object control 156
6.14 Dataflow of the object-impedance controller 157
6.15 Experimental tracking performance of the object-impedance controller 158
6.16 Animation of trajectory followed by the object under cooperative control 159
6.17 ControlShell.4.x's Finite-State Machine model 160
6.18 Architecture of the strategic-control layer 161
6.19 Sources of arm and object trajectories 163
6.20 Transition Graph for the Arm Finite-State Machines 166
6.21 Transition Graph for Capture Subchain 167
6.22 Transition Graph for Release Subchain 168
6.23 Capture Strategy 169
6.24 Experimental single-arm capture of moving object 171
6.25 Experimental dual-arm capture of moving object 172
6.26 Experimental simultaneous multiple-object capture 173
6.27 Animation of experimental data of simultaneous multiple-object capture 174

6.28 Photographic sequence of system picking two objects from the conveyor simultaneously . . . 175
6.29 Photographic sequence of system picking two objects from the conveyor . . . 176
6.30 Photographic sequence of system using two arms to pick the same object from the conveyor . . . 177
7.1 The Time-Parameterization Process . . . 180
7.2 Optimal time-parameterization constraints in phase-space . . . 189
7.3 Illustration of conditions of Theorem 1 . . . 192
7.4 Integration trajectory in phase space using proximate-optimal constraints . . . 193
7.5 Steps of proximate-optimal algorithm . . . 196
7.6 Running time versus number of via points for different number of degrees of freedom 200
7.7 Sketch of Shin's direct-integration algorithm (from [164], Figure 6) . . . 202
7.8 Worst-case scenario for direct-integration algorithms. . . . 203
7.9 Modified direct-integration algorithm that avoids worst-case complexity problem . 205
7.10 Comparison of approaches to time-parameterization . . . 206
7.11 Patching of an ongoing trajectory . . . 209
7.12 Trajectory Modification Algorithm . . . 211
7.13 Example of Trajectory Modification Algorithm . . . 212
7.14 Time-parameterization of straight-line path with acceleration limits . . . 213
7.15 Time-parameterization of straight-line path with velocity and acceleration limits . 215
7.16 Time-parameterization of a joint-space straight-line path . . . 216
7.17 Time-parameterization of a cartesian straight-line path . . . 217
7.18 Time-parameterization of a dual-arm path . . . 220
7.19 Initial path and two subsequent modifications . . . 221
7.20 Time-parameterization of initial path . . . 222
7.21 Results after first modification to on-going trajectory . . . 223
7.22 Results after second modification to on-going trajectory . . . 224
A.1 Vision system calibration . . . 241
A.2 Kinematic calibration . . . 242
C.1 Step Response of Inverse-Dynamic Arm Controller . . . 247
D.1 Arm Schematic . . . 248
D.2 Manipulator Coordinates . . . 249
E.1 Filtered random walks of different length for 2 degrees of freedom . . . 252

F.1 Two worst-case paths of different lengths. Each paths is composed of $2N$ circular arcs. 254
F.2 Phase space direct integration and detail. 256
G.1 A typical session using the `matrixPlotServer` 259

Chapter 1

Introduction

This dissertation describes the Integrated Dual-Arm Manufacturing Workcell project conducted at the Stanford Aerospace Robotics Laboratory (ARL) during 1991-1994. This project encompasses the experimental development of an intelligent, dual-arm robotic workcell that includes on-line planning as an integral part of the architecture. The system combines a high-level graphical user interface, a hierarchical control system, an on-line motion planner, a real-time vision system, and an on-line simulator to control and monitor the workcell.

1.1 Motivation

Technical progress has an ever increased need for automation whereby human labor is promoted to a higher, creative level of non-repetitive tasks performed in a human-friendly environment. In this context, robots provide two important capabilities: (1) they can move and manipulate the environment (manufacturing, materials handling) and (2) they can gather information passively (inspection) or actively (sample collection). In this manner, robots can extend the frontier of tasks humans can comfortably achieve.

The current range of tasks subject to robotic automation ranges from manufacturing (assembly, packaging, welding, spray-painting), operations in hazardous environments (waste-remediation, radioactive or toxic environments), and activities in human-unfriendly environments (underwater sample collection, space exploration, mining).

The long-term goal is the development of systems capable of operating autonomously (or semi-autonomously) in unstructured environments. These systems would accept task descriptions from human supervisors and perform these tasks on their own, while handling all the specific details

themselves. These systems should adapt to the surrounding circumstances to accommodate changes in both external and internal conditions, partial failures, and other (unexpected) events.

To achieve these ambitious goals, robotic systems must incorporate sophisticated control, sensing, and planning capabilities. On-line planning is required to avoid the need to pre-program in excruciating detail all the required tasks. Tediousness aside, important details may be simply unknown in advance and therefore cannot be preprogrammed. Sophisticated sensing is essential if the system is to exhibit some degree of autonomy in a non-perfectly-structured environment. Control is fundamental for the robust operation of any mechanical system interacting with its environment.

Unfortunately, systems of such sophistication will be complex. Equally important for their success is the requirement for these systems to be developed economically and operated with ease. Given the difficulty of systems integration, its economical development requires a solid architecture and methodology, powerful interfaces, and flexible communication[1]. Ease of operation requires a high-level human interface for both task specification/command and system monitoring.

In the context of manufacturing workcells, the natural evolution will take robotics beyond the current sensor-poor robotic systems operating in perfectly structured environments (painting, spot-welding, pick-and-place from feeders), where robots perform repetitive tasks, towards a new generation of sensor-rich robots capable of achieving a variety of tasks (contact assemblies, surface finishing etc.) in a flexible manner, i.e., without the need for complicated fixturing and scheduling. Within the manufacturing context, "reacting to the environment" will extend beyond avoidance of unexpected objects (e.g. humans or other robots) towards adaptation to changing manufacturing needs (e.g. be able to change the mixture/rate of production and even to reconfigure to build new parts).

The economic benefits of an intelligent workcell would be significant. Currently all tasks in the workcell must be carefully preprogrammed. This requires a perfectly-structured environment where everything is known in advance. Structured environments depend on a variety of fixturing devices and must be carefully scheduled. All this adds significant costs for the additional equipment and time required to arrange the operation. For products with a short shelf life, time-to-market and time-to-volume are becoming more critical than steady-state production rate. Complete preprogramming of the workcell causes design and manufacturing to proceed serially for a given product. Once the design is complete, careful workcell pre-programming, fixture selection and fabrication setup are required before production can begin. Only at this point may some flaws become apparent, which

[1]Many of these systems are distributed due to size/weight/power requirements, or the need to control/monitor them from remote (safe) locations. Hence the need to communicate among the different subsystems.

may require redesign. A workcell incorporating sophisticated planning capabilities can operate without much of the fixture and scheduling burden, so production can begin as soon as the design is complete, and perhaps more importantly, design improvements can be incorporated as they occur, without the need for reprogramming.

To help advance the underlying theory and technologies necessary to develop such systems, the Stanford Aerospace Robotics Laboratory (ARL) has undertaken focused experimental study of the problems associated with developing intelligent manufacturing workcells.

1.2 Research Objectives

The objective of this research is to study the architecture and interfaces of distributed robotic systems that incorporate planning as a fundamental, integral part, and to demonstrate these ideas *experimentally* in a system capable of semi-autonomous capture and (cooperative) placement of multiple (moving) objects under high-level user direction, even in the unforeseen presence of workspace objects. Moreover, the system must function without any á priori fixturing or scheduling. The experimental concept is depicted in Figure 1.1.

Specifically, the goal of this project is to develop a complete *experimental system*–integrating a two-armed robot, a conveyor, vision sensing, planning and human interaction–that meets the following requirements:

- High-Level human interface. Complete tasks are specified only in terms of the desired assemblies and their locations (not of the steps, sequences, paths, via points etc. required to achieve them).

- Semi-autonomous operation in a dynamic environment. The system must be capable of autonomous tracking and acquisition of parts from a moving conveyor.

- Two-handed manipulation. The system must be able to use both manipulators cooperatively whenever appropriate.

Moreover, the system must:

- Operate in a distributed environment where the command, control, and planning stations are distributed among computers in a network; and it must do so *reliably*, allowing for alternative subsystems to replace/interact with the original ones at any time.

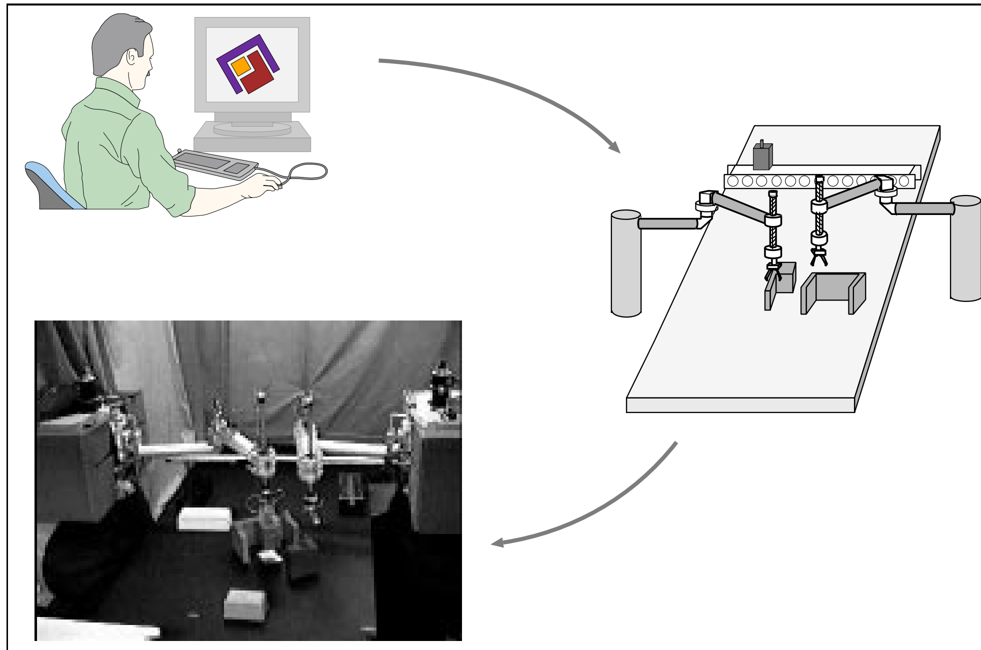


Figure 1.1: **Research Objective**

A manufacturing workcell is directed by a user using very-high-level task commands to make an assembly. The parts required for the assembly will appear on a conveyor. The system automatically acquires the parts and performs the assembly. All intermediate steps: required paths, trajectories, tracking and acquisition of parts, manipulation, etc., are performed by the system autonomously under high-level human supervision.

- Allow multiple concurrent operating modes in a transparent fashion. For instance, have multiple users interact/monitor the operation of the system.
- Operate in a semi-structured environment with no fixtures nor á priori scheduling.
- Ensure safety by combining on-line planning and real-time collision checking.
- Incorporate and define "open" interfaces between the major subsystems in such a manner that the system can easily be expanded, and new capabilities can be added. In particular it must interact with on-line planners in an expandable manner, allowing for the incorporation of diverse planning methodologies.

1.3 Research Issues

The above objectives require advancing the current research in several areas:

System architecture, design, interfaces and integration. System integration is a difficult problem. Often the integration effort surpasses that of developing the intermediate subsystems. While the number of subsystems grows linearly with system size, the number of possible interfaces increases quadratically (every subsystem can potentially interface with each of the remaining ones). To alleviate this problem, it is important to carefully select both architecture and interfaces. While substantial research has addressed the architectures required for specific robotic applications, similar research in the robotic subsystem interfaces has been limited.

Integration of planning into robotic systems. The need for planning in many robotic systems is well established. Planning gives these systems higher degrees of autonomy and increased robustness as they can react to unexpected events. On-board planning permits utilization of high-level task commands instead of high-bandwidth teleoperation signals. Autonomy allows systems to operate in the presence of large time delays (such as in space exploration), low communication bandwidth (such as undersea vehicles), and allows the human to be more productive as a supervisor (as opposed to operator or low-level programmer). However, integrating planning into a control system with hard real-time deadlines remains a great challenge. Proof of this has been the emergence of more "reflective" approaches that trade off the ability to pursue and reason towards long-term goals with the capability to react quickly to environmental changes. The balancing between these approaches, and the specific interfaces between planning and robotic subsystems, is an active area of research.

Automatic trajectory generation. Most industrial robots still perform repetitive tasks (pick-and-place, inspection, welding etc.). In most of these cases the spatial motion of the robots is determined by the application, while the trajectory itself (the motion as a function of time) is painfully hand-tuned by system integrators so that minimum cycle times can be achieved without exceeding the capabilities of the manipulators (e.g. motor torque limits). Substantial research has been performed, and many algorithms have been proposed to automatically generate trajectories from the spatial description of the path that minimize fairly general performance indices (e.g. minimum time or energy consumption), subject to the limits of the controlled system. Only recently has attention been focused on "on-line" algorithms that trade off overall optimality for algorithmic speed. Still, for a real-time system, a critical property of any algorithm is the ability to predict "á priori " its running time. This aspect had not yet been addressed.

Control hierarchies for multiple-arm robotic workcells. The control of multiple-arm robotic workcells is a complex software problem. Hierarchies are one methodology to tackle these complex systems. A hierarchical approach allows the problem to be simplified and redefined in a layered manner where each layer performs a well-defined function and transforms (simplifies or abstracts) the problem for the layers above. The benefits of this abstraction process have to be balanced with the constraints imposed by the hierarchy; that is, higher levels are now limited in their actions by the layers below. While many solutions have been proposed, designing a hierarchical control system that simplifies the control problem without hampering or unnecessarily restricting the flexibility remains an unresolved problem.

Information sharing (communication) in distributed robotic systems. The need to share data and event information is universal to all distributed[2] systems. Distribution may be dictated by physical requirements such as weight/size/power budgets that prevent all the sensing/computation from being on-board. Distribution may also be desirable whenever the system functions as an extension/replacement for humans in hostile environments, and requires remote human direction and monitoring. Perhaps as fundamentally, many of these systems are distributed for "logical" reasons: As the sophistication of the system and the need to incorporate more complex sensing, reasoning and interfaces grows, so does the tendency to design such systems as a network of subsystem modules where each subsystem performs a well-defined function. This has resulted in the proliferation of custom "communication" schemes for specific applications (e.g telerobotics) and in ongoing research in distributed blackboard and other information-sharing architectures. Distributed robotic systems in general and robotic systems in particular have specific needs that are not addressed by traditional approaches (such as distributed databases, transaction systems, etc.). Well-established semantics, such as client/server, shared memory and message passing, do not exploit or support the specific characteristics of repetitive sensor-type information. More recently, publish/subscribe (dissemination-oriented) communication models have been researched. Prior to the work presented in this thesis, there was no communication protocol that exploited the nature of the information flow in distributed control applications while enabling transparent communication and complex, dynamically-reconfigurable, data flows.

[2]The word distributed is used to mean that the different subsystems are logically or physically separate, containing their own computational resources and communicating over some link.

1.4 Summary of Results

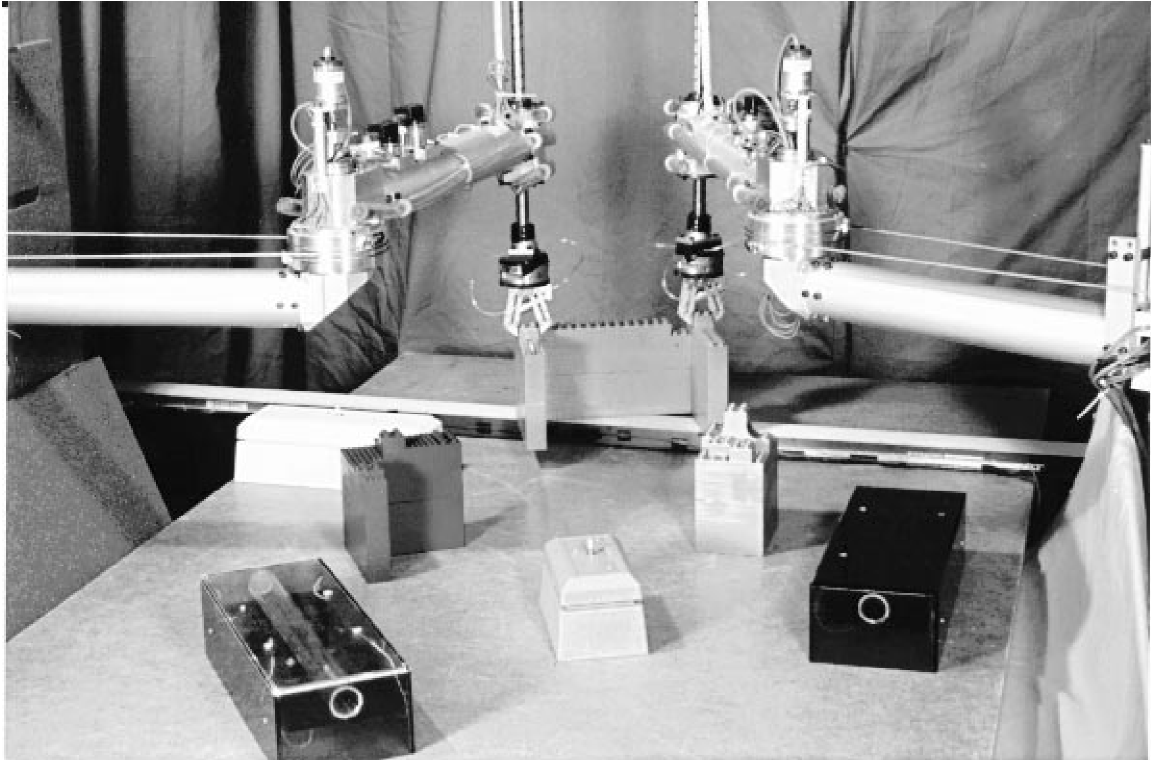


Figure 1.2: **Experimental System**

The experimental setup is composed of two SCARA-type manipulators, a conveyor, a set of manipulable objects, a number of non-graspable obstacles, and a vision system (not shown in the picture).

1.4.1 System Capabilities and Demonstrations

The ability to direct a dual-arm workcell (Figure 1.2) with very-high-level assembly-type commands has been experimentally demonstrated. Using a graphical interface, the operator can directly specify desired locations for several objects in the workspace without concern for: (1) current object positions, (2) how objects arrive to the workcell, (3) which arm or how many arms should grasp the objects, and (4) what intermediate motions are required. The system is essentially driven by the availability of parts (as they arrive on a conveyor or otherwise appear in the workspace of the arms). The user can supervise the system's operation by changing its directives or issuing new commands.

The ability to plan collision-free trajectories for either (or both) arms to capture objects from a moving conveyor and deliver them among workspace objects has been demonstrated. These plans are computed on-line using the sensed position of all objects and arms in the workspace.

The power of anonymous communication and interfaces has been illustrated in several experiments where multiple human interfaces operate concurrently and are used to command the system. The rest of the system did not have to be programmed specially, and was not even aware of these changes. The interfaces also allowed the author to develop a simulator that could be used to masquerade as the real hardware for illustration and debugging purposes.

Transparent subscription-based network communication has also been demonstrated. Aside from allowing each subsystem module to tailor the nature and frequency of the information updates to its specific needs, it has made it possible to move (and replicate) the different subsystem modules among different computer systems[3] while the system is operating. Transparent network communication allowed us to take advantage of the best computer hardware available for each function. Specifically, the planning subsystem was moved to take advantage of higher-performance computers, while the graphical interface was replicated to allow operators in different rooms to interact and take advantage of specific graphics hardware. The advantages are increased flexibility and robustness.

[3]With the exception of the subsystem that controls the actual hardware which is tied to the real-time cage containing drivers for the motors and sensors.

To the author's knowledge, this research has accomplished the first:

- Experimental demonstration of dual-arm capture and multi-step delivery of moving objects among obstacles in the workspace using on-line planning.
- Fully distributed, subscription-based data-sharing system that supports multiple producers and consumers, and incorporates a realistic model of time.
- Separable time-parameterization algorithm with linear run-time complexity and predictable execution time (for a given manipulator as a function of the path length).
- Experimental demonstration of anonymous, stateless robot interfaces, allowing simulator masquerading and collaborative multiple-user interaction.
- Demonstration of modular, parametric interface design applied to robotic systems.
- Robotic-system design using the "interfaces-first" technique.

1.4.2 Contributions

This research makes the following contributions to the fields of robotics, distributed control and software systems:

1. A complete system architecture integrating on-line planning into a distributed, dual-arm robotic workcell has been developed. A careful analysis of the interfaces and world modelling issues has been performed, specifically concerning the interaction of planning with a hard real-time control system in the presence of unpredictable planning and communication delays. The final system incorporated a variety of subsystems:

   - Human Interface. A graphical user interface containing a 3-dimensional rendering of the workspace allows the human to issue high-level task commands.
   - Planner. A combined task-planner and path-planner[4] subsystem decomposes task commands into a sequence of safe, primitive "robot commands."
   - Robot Workcell Controller and World Model. A hierarchical controller for the dual-arm workcell and the supporting world model. The world model integrates data from different sensors, taking advantage of domain-specific knowledge given the status of the workcell.

[4]These planners were developed at the Stanford Computer Science Robotics Laboratory.

- Robot Workcell Simulator. A simulator capable of masquerading as the workcell while providing graphical simulation and world model functionality.
- Software Communication Bus. A Software package that allows transparent distributed network connectivity with unlimited fan-in and fan-out, thus providing the illusion of a software bus.

The success of this design should provide a useful model for the development of similar systems.

2. A philosophy for designing robotic systems beginning with the interfaces (*interfaces-first design*) has been proposed, developed, and demonstrated in a complete operational system. Furthermore, the concept of *anonymous* interfaces has been introduced and successfully demonstrated.

3. A modular approach to developing system interfaces has been proposed and demonstrated. This technique structures the complex command and data flow as combinations of several *primitive* interface modules. These primitive interface modules have parameters that allow customization for use within a specific, larger interface.

4. Three fundamental robotic-interface components: world-model, robot-command, and task-level-direction have been developed. These interfaces can be combined to generate custom interfaces between all the subsystems. A detailed study of the nature of the dataflow in robotic systems has led to the selection of these interfaces to encapsulate three fundamentally different classes of information:

   - World-State Interface. Encapsulates periodic, sensor-type information, where only the most current data is required. Different clients need this information at different rates to accommodate their specific needs.
   - Robot-Command Interface. Command-type information that must be delivered reliably and in order. This information is non-periodic and indicates an action that must be taken immediately or discarded.
   - Task-Interface. Goal-type information. Also requires reliable ordered delivery, and is generated asynchronously. However, it does not indicate an explicit action but rather a "success condition," and it persists until the goal is achieved or explicitly cancelled.

These interfaces alone are capable of describing the system geometry and kinematics necessary for the planner and human-interface subsystems, providing a step toward the development of generic ("common") interfaces for robotic systems. They should serve as useful examples for similar systems.

5. A new subscription-based, network-data-sharing system (NDDS[5]) has been developed. This system allows data to be transparently accessed anywhere on a computer network. It introduces several new concepts not present in existing distributed communication packages:

   - Multiple anonymous producers and consumers. An unlimited number of producers can dynamically override each other. Any consumer can access data produced anywhere.
   - Realistic model of time. Allows explicit specification of custom update rates, deadlines and what-to-do if a deadline is missed. Data is time tagged and decisions can be made based on the time when data was produced/consumed.
   - Fully symmetric, stateless implementation. The implementation contains no central servers or privileged nodes. All communication nodes are symmetric and contain no state (other than time-decaying caches). This implementation is very robust to partial failures.

   Complex distributed robotic systems require very complex data flow. NDDS orchestrates this data flow and provides a model of the communication that matches well the requirements of distributed control systems. NDDS has been demonstrated in the experimental system, and is now being utilised by most current projects at the Stanford Aerospace Robotics Laboratory [102, 195, 206]. This system has also become a commercial product [145].

6. A fast (linear-time, proximate-optimal) solution to the problem of time-parameterization of robot paths has been obtained. This algorithm has the fundamental property that for a given manipulator dynamics, its running time is predictable, depending only on the length of the path. This allows planners to anticipate the minimum time required for a motion and thereby adjust the intercept position when a rendezvous with a moving object is required. Moreover, the algorithm provides for modification of on-going trajectories. This algorithm has been packaged in a general tool that allows specification of arbitrary robot dynamics. It is now being used in a variety of experiments [112, 104].

[5]NDDS stands for "Network Data-Delivery Service".

7. Strategic algorithms have been developed to allow tracking and capture of moving objects by one or two arms, as well as simultaneous capture of multiple objects (one per arm). These algorithms tie planned trajectories with local intercept trajectories and merge direct sensor-driven feedback (tracking) in some degrees of freedom with task-driven trajectories on other degrees of freedom. In addition, a scalable approach to the strategic control of a multi-arm workcell has been developed. This approach uses multiple instances of otherwise identical finite state machines to control each arm in the workcell.

8. A graphical human interface with full 3-dimensional perspective has been developed allowing facile specification and construction of task commands. The GUI is updated with direct sensor data (from a world model) and displays the status of both the workcell and the planning subsystem.

9. The concept of task-level control [189] has been extended to tasks where goal-driven planning is required and where multiple concurrent goals can be present at any time.

10. A complete control hierarchy ranging from joint-level local torque control to high-level arm and object control has been demonstrated. Specifically, a set of mixed control modes for intercepting, tracking and picking moving objects smoothly from a conveyor has been constructed.

1.5 Related Research Activities

This section lists some of the efforts closely related to this research as well as supporting activities by other researchers from the Stanford Aerospace Robotics Laboratory. This is not intended as a literature review. The appropriate chapters will include detailed discussions and reviews of the related research.

Previous research at the Stanford Aerospace Robotics Laboratory [155, 189] has demonstrated the ability to track and capture objects with multiple manipulators using vision feedback. Schneider [157] also demonstrated simple cooperative-arm vision-guided assembly. None of the previous work at ARL incorporated on-line planning.

In the field of on-line planning for robotic workcells, Hormann, Rembold and Lueth [1, 67, 95] describe KAMRO: a workcell containing a pair of 6-DOF Puma 260 manipulators mounted on the ceiling of a 3 DOF omni-directional platform. Their system shares similar goals with the

author's and incorporates on-line planning to achieve semi-autonomous operation. One of the major differences with the author's scenario is that there were no moving objects in the workspace.

Fu and Hsu [51] have developed a similar scenario dealing with two robots and multiple moving objects on a conveyor belt. In this work, they assume that only the last links of the arms may collide with each other and that the objects are fed slowly enough that the robots can always pick up objects from locations that do not deviate significantly from their nominal positions. The author's approach is substantially less restrictive.

Other authors have presented results on either tracking [12] and/or capturing objects using vision. In some of these cases [8] vision was used only to estimate the object trajectory, while the capture was essentially done open-loop. In others [27] a single arm is used, and its motion is restricted to always capture the object in the same plane.

1.5.1 Supporting Research Activities

A project of this magnitude would have been impossible were it not for the availability and assistance of an array of research results and tools provided by the ARL team. Except where otherwise stated the individuals mentioned below developed their work at the Stanford Aerospace Robotics Laboratory.

Experimental Platform

Larry Pfeffer [131] designed and constructed the experimental platform that was used for the experiment. Although a few enhancements were made (longer links, redesign of the last two axes of the robots), the main features, supporting electronics and computer hardware required few adjustments.

Planning for Robotic Systems

Planning is a fundamental ingredient in the overall system operation. The requirement that the system incorporate planning, while maintaining on-line performance has had profound implications. Both the planning portion and the rest of the system have been designed with these goals in mind. Tsai-Yen Li [91] and Yoshihito Koga [82] of the Stanford Computer Science Robotics Laboratory developed the planning subsystem as part of their Ph.D. research in a joint research program with the Stanford Aerospace Robotics Laboratory.

Real-Time Software Framework

A software project of this magnitude would have been intractable without appropriate software tools and a solid design framework. As part of his Ph.D. research, Stan Schneider [154] developed the *ControlShell* software framework [160, 159, 158] to help the design of complex, interactive real-time systems. This framework supports systems that combine event-driven with data-flow computational models. Both aspects are essential requirements of a control system that must interact with its environment. *ControlShell* can be credited for the timely completion of this research.

Dual-Arm Cooperative Control

Cooperative control of objects with multiple manipulators is a fecund field. As part of his Ph.D. thesis at Stanford, Stan Schneider [154] developed the “Object-Impedance Control” concept which was later extended by Larry Pfeffer. In his PhD. thesis, Larry Pfeffer [131] also developed a system identification and control methodology to address the control of robotic arms with exaggerated joint flexibility. These methodologies were followed by the author to develop the lower layers of the control hierarchy, and this greatly simplified the lower-level control problem.

Real-Time Vision System

A robotic system that manipulates objects in its environment needs sophisticated sensing capabilities to detect, identify, and track the different objects. The requirement of tracking and capturing moving objects from a conveyor imposes constraints on the minimum bandwidth of the sensors used to track the object. Although this capability exists with off-the-shelf products, they are usually expensive and require substantial effort to be integrated into a closed-loop control system. As part of his Ph.D. research, Vince Chen [29] developed the *Point Grabber II*, an inexpensive board capable of tracking individual LED’s in a scene. Stan Schneider developed the *VisionServer* software to identify and track objects tagged with unique LED patterns. This hardware and accompanying software tools allowed easy integration of visual sensing into the overall system.

Graphical User Interfaces for Robotic Systems

The ability of a human operator to visualize and interact with the operational system using natural 3-dimensional views was a crucial part of the success and appeal of the experiment. As part of their Ph.D. work, Kurt Zimmerman and Howard Wang developed a graphical tool (*GraPhigs*) to facilitate building such interfaces from textual descriptions of geometry and kinematics of the manipulators

and other objects in the scene. This tool made it very easy to visualize the geometric models and create Graphical Human Interfaces and simulators.

1.6 Reader's Guide

This thesis is divided into eight chapters and nine appendices. Chapter 1 has provided an overview and motivation for this research. Chapter 2 presents the experimental dual-arm robotic workcell hardware (including the computer environment), Chapter 3 introduces the software system design methodology, the interfaces, system architecture, and complete-system operation. The next three chapters provide details on some fundamental elements of the architecture and subsystems: The communication system NDDS (now a general tool for distributed control applications) in Chapter 4, the world-modelling subsystem in Chapter 5, the hierarchical control system in Chapter 6, and the on-line trajectory generator in Chapter 7. Chapter 8 presents a summary of this project, draws conclusions, and provides suggestions for future research.

Appendix A details the calibration procedures used. Appendix B and C document the control gains used. Appendix D contains the dynamic parameters of the manipulators. Appendix E presents the "canonical" via-point paths used for computational complexity measurements in Chapter 7. Appendix F presents a constructive proof for the worst-case complexity of "classical" time-parameterization algorithms described in Chapter 7. Appendix G documents the distributed `matrixPlot` package developed in support of this research. Finally, Appendix H presents detailed documentation of several software packages: the Network Data-Delivery Service (NDDS described in Chapter 4), the proximate-optimal via-point trajectory generator (see Chapter 7), and supporting libraries.

Chapter 2

Experimental Dual-Arm Robotic Workcell

This chapter describes the dual-arm robotic workcell: manipulator hardware (an enhanced version of the one first built by Pfeffer [131]), workspace objects, conveyor, vision system and computer environment.

2.1 Introduction

The experimental workcell shown in Figure 2.1 contains two robotic manipulators. Each manipulator has 4 degrees-of-freedom (DOF) and is of the SCARA configuration. These arms are placed facing each other in an attempt to balance the total "reachable" workspace (of either arm) with the volume where cooperative manipulation can be performed. The arms move above a task table containing a set of objects and obstacles as well as a small conveyor. The height of the table is such that the grippers, at their lowest position, can barely contact the table surface.

2.2 Manipulator Mechanism and Sensors

The original manipulator mechanism was designed and built by Pfeffer (see [131] for a detailed description) to study cooperative manipulation in the presence of exaggerated joint flexibility. The new experiments required the redesign of the last two degrees of freedom (Z and yaw axis) and the extension of the shoulder and elbow links to increase the workspace of the manipulators. For

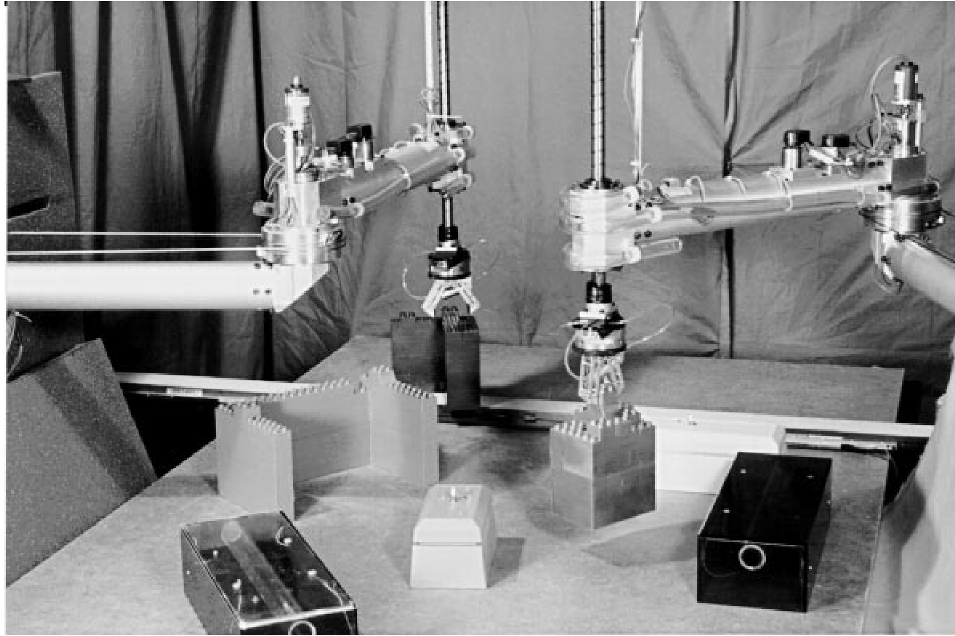


Figure 2.1: **Experimental Dual-Arm Robotic Workcell**

The experimental setup is composed of two SCARA-type manipulators, a conveyor, and a set objects.

completeness, this section will include the relevant details of the manipulator design as well as the modifications to the original design.

2.2.1 Kinematics

The SCARA[1] configuration contains the first two links in the horizontal plane connected serially by revolute (vertical axis) shoulder and elbow joints. The third is a prismatic joint along the vertical (Z-axis) while the last is again a revolute joint with vertical axis. This configuration is standard for many commercial robots (for instance Adept, Panasonic, Sony, and IBM all manufacture SCARA-type manipulators), and is shown schematically in Figure 2.3. The arm kinematics are described using the Denavit-Hartenberg parameters [41, 38] in Table 2.1. The first two degrees of freedom (shoulder and elbow) are actuated from motors located in the robot base using cable drives. Exaggerated joint flexibility was introduced in the drivetrain using torsional springs. This flexibility was incorporated for earlier research [132]. Controlling it represents an important dimension of control-system robustness.

[1]Selective Compliance Assembly Robot Arm

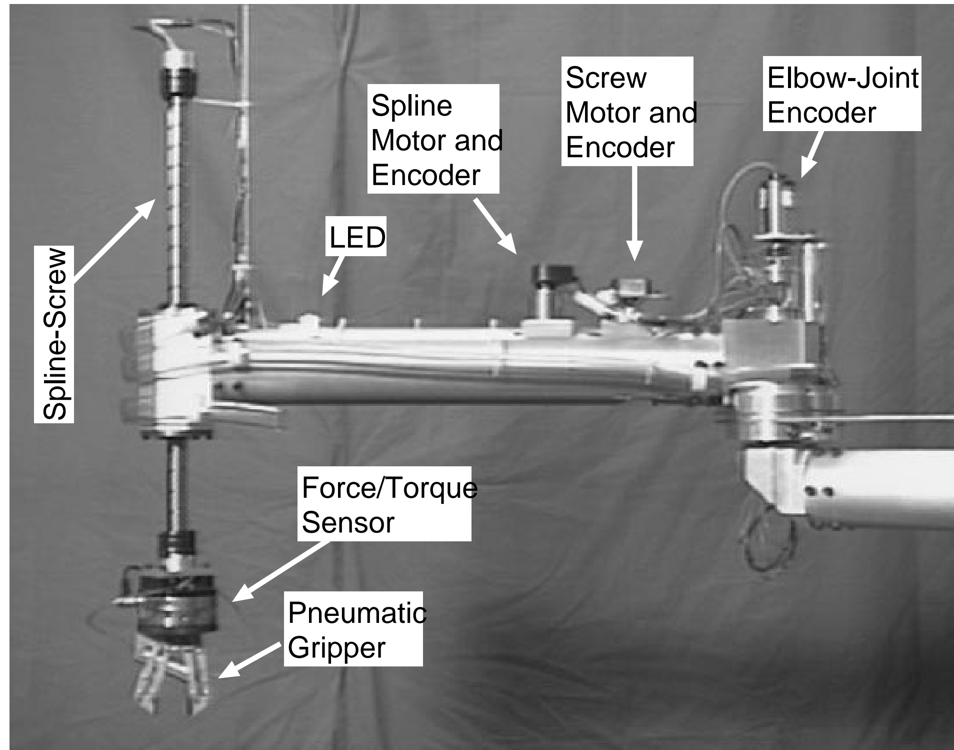


Figure 2.2: **Z and Yaw degrees of freedom**

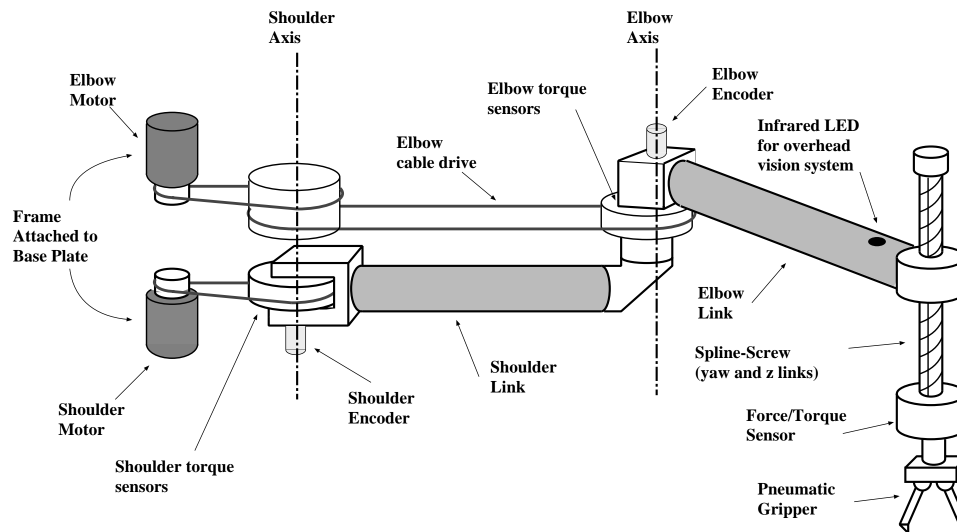


Figure 2.3: **Arm Schematic**

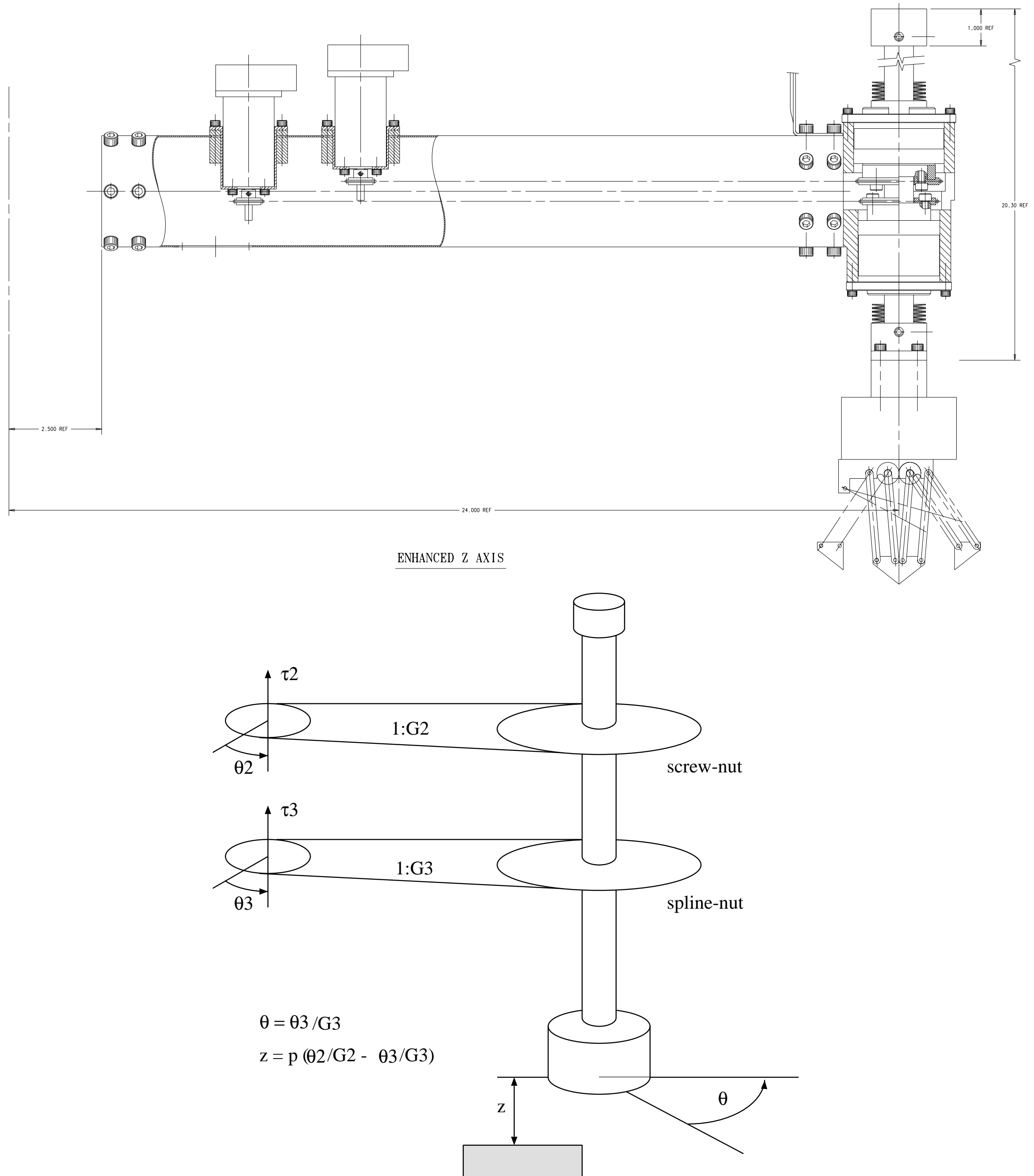


Figure 2.4: **Schematic of last Z and Yaw degrees of freedom and actuation**

The last two degrees of freedom (Z and Yaw) use a new screw-spline component manufactured by THK [94]. This mechanism contains both spline and screw grooves in the same vertical shaft. Two ball-bearing nuts each use one of the grooves to achieve both translational and rotational motion of the shaft. Both nuts are mounted on the elbow link using a custom-machined piece, and are driven using belts from two motors also located in the elbow link. This is illustrated in Figures 2.4 and 2.2.

The actuators and gear ratios were selected so that the arm had similar performance in all degrees of freedom. The original design, based on our estimation of the speed of the planning subsystem, called for a 1 second move involving a 1 meter x-y displacement, 2π rotation angle and full 0.3 meter vertical displacement. Located at the arm base, the shoulder and elbow motors had no limits on their size/weight; they were over-designed in the initial experiment, and had no trouble meeting the aforementioned requirements. In the last two degrees of freedom, however, the actuators are mounted in the elbow link and therefore it was desirable to make them as light (and small) as possible to minimize the inertia of the arm. The limitations on available screws and gear-ratios, and the desire to be able to execute full up and down motions of 30 cm in times of the order of 1 second[2] called for at least 50 oz-inch peak torque. These considerations motivated the use of high power-to-weight ratio rare-earth-magnet DC motors. Table 2.2 summarizes the characteristics of all the actuators and gear-ratios.

2.2.2 Joint-Torque Sensors

The shoulder and elbow joints contain joint-torque sensors built into the structure, so that the torque applied to each link can be directly measured. The use of joint-torque sensors allows closing fast torque loops on these degrees of freedom, so that joint dynamics and non-ideal characteristics of the actuators can be hidden from the upper control layers. The design of this sensor, its performance, and its frequency response are described in detail chapter 5 and in Appendix C of Pfeffer's thesis [131].

2.2.3 Joint Encoders

Each manipulator contains six optical encoders, four mounted on the motor shafts (dual-shaft motors where used for this) and two at the shoulder and elbow links (outboard of the flexibility). The shoulder- and elbow-motor encoders were designed to allow cogging compensation and to have

[2] This design requirement stems from the desire to balance the performance of all degrees of freedom in a typical pick-and-place operation. The first 2 degrees of freedom could move the arm across the workspace with good tracking in about 2 seconds.

Denavit-Hartenberg parameters					Limits	
i	α_{i-1}	$a_i - 1$	d_i	θ_i	min.	max.
1	0	0	0	q_{sh}	-1.2 rad	1.2 rad
2	0	0.6096 m	0.12 m	q_{el}	-2.5 rad	2.5 rad
3	0	0.6096 m	z	0	0.02 m	0.30 m
4	0	0	0	q_{yaw}	-6.0 rad	6.0 rad

Location of Arms in Workcell					
arm name	x	y	z	θ	LED position
right arm	-0.0831	-.9445	0.45	$-\pi/2$ rad	0.4750 m
left arm	-0.1057	0.76	0.45	$+\pi/2$ rad	0.4856 m

Table 2.1: **Arm kinematic parameters**

Nominal Denavit-Hartenberg parameters for the manipulator arms are identical for the two arms. The notation used follows that of Craig [38]. The arm locations denote the position of the base frame (origin for Denavit-Hartenberg parameters) w.r.t. the global frame. The LED offset is its distance to the elbow axis. Both the location of the arms and LED offset were computed from vision information. The procedure collected encoder-angle and vision data at a grid of locations covering the complete workspace. A minimization was then performed to identify the location of the shoulder axis, encoder offsets, and LED location that would bring kinematics and vision to their closest correspondence over the whole workspace (see appendix A for details). This set f data is the one shown in the Table.

Actuator	*Brand*	*Peak/Continuous Torque*	*Torque constant*	*Gearing*
Shoulder Motor	Contraves ACR103	40.0/8.0 N-m	40.0 N-m/A	9:1
Elbow Motor	Contraves ACR103	40.0/8.0 N-m	40.0 N-m/A	3:1
Z- Axis Motor	ESCAP 35 NT2R 82	0.72/0.11 N-m	0.05 N-m/A	0.06 m/rev
Wrist Yaw Motor	ESCAP 35 NT2R 82	0.72/0.11 N-m	0.05 N-m/A	2.75:1
Conveyor Motor	CP JDH-2250-BX-1C	2.45/0.29 N-m	0.10 N-m/A	0.0118 m/rev

Table 2.2: **Actuator characteristics**

The Contraves are AC brushless motors, the ESCAP are DC brush-commutated rare-earth permanent-magnet motors. CP stands for "Clifton Precision". All motors are driven by transconmutance amplifiers (voltage inputs command current forced through the motor windings), and hence act as torque sources.

fairly low resolution. The highest-resolution encoders are located at the shoulder and elbow joints. Together with the encoders on the last 2 DOF they allow determination of the tool position with respect to the robot base using rigid-link kinematics (the drivetrain flexibility of the last 2 DOF can be neglected). Table 2.3 summarizes the characteristics of the different encoders. The encoder

signals were accessed using quadrature-decoding interface boards developed by Uhlik [188]. These boards provide both incremental position and absolute velocity measurements.

The high-resolution shoulder and elbow encoders provide a velocity signal with low noise. Encoder resolution in the last two DOF was limited by speed limitations in the interface circuitry, and proved sufficient for all tasks under consideration.

Location	Brand	Lines on disk	Equivalent Resolution
Shoulder Joint	Cannon M-1	50000	0.03146 mrad
Elbow Joint	Cannon R-1 L	81000	0.01939 mrad
Z axis Motor	HP HEDS 5500	96	0.01736 mm
Wrist Yaw Motor	HP HEDS 5500	512	1.116 mrad
Shoulder Motor	BEI E203	2048	0.7670 mrad
Elbow Motor	BEI E203	2048	0.7670 mrad

Table 2.3: **Encoder types and resolution**

The number of lines on disk indicates the number of pulses in a complete revolution. Since they produce quadrature counts we get four times the number of counts per revolution, with the corresponding increase in angular resolution. The equivalent resolution takes into account the gear-ratio of the Z and wrist motors.

Pneumatic grippers were selected for their simplicity. These grippers were built by Zebra Robotics and are actuated by a pneumatic cylinder of 0.25 in^2 (1.61 cm^2) section. Shop air is used to pressurize up to 40 psi to achieve a grasping force of 10 lbs (44.5 N).

2.3 Workspace Objects

The workcell manipulators manipulate and interact with a number of objects, some of which can be grasped and manipulated while others are "pure obstacles" in the sense that they obstruct the motion of the other objects but cannot be manipulated by the robots. Each object is marked with a unique LED pattern (three LEDs, with unique distances). This unique pattern allows the vision system to identify and track the objects. As we will discuss in chapter 5, there is nothing special about these objects other than they have been tagged with LED's and described to the world modeler.

These objects are shown in Figure 2.5, their characteristics are summarized in Table 2.4.

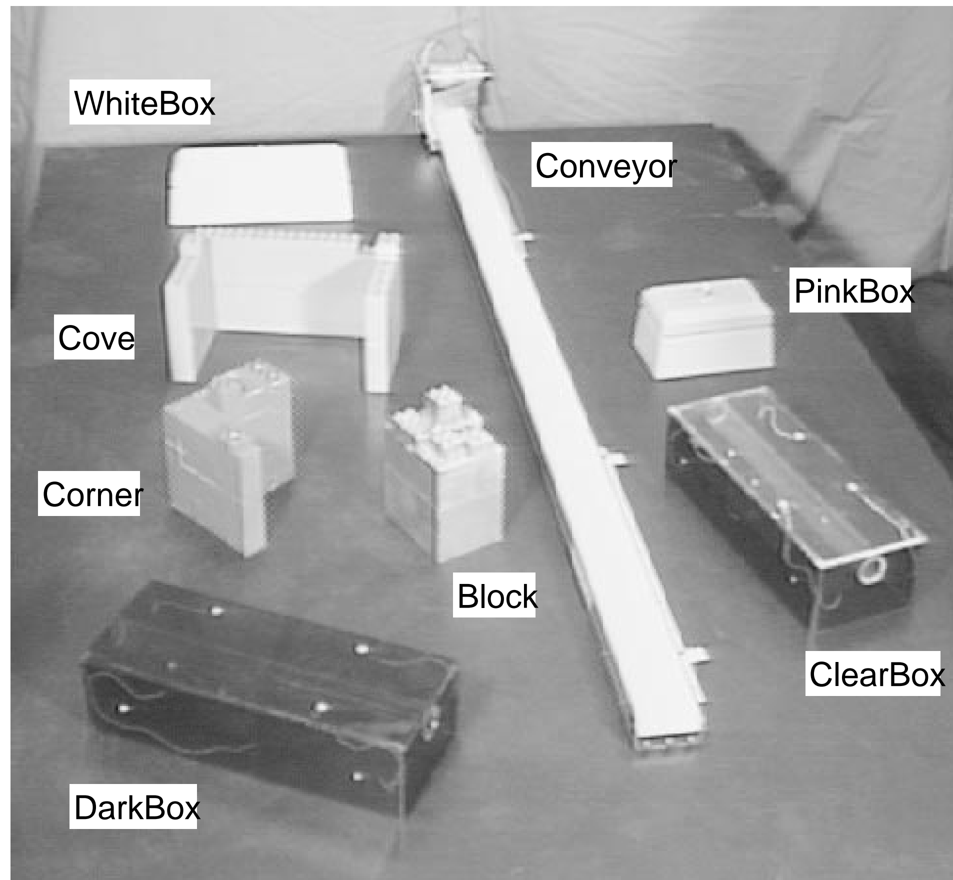


Figure 2.5: **The different objects the robot interacts with**

The workcell interact with 7 objects: Block, Cove, Corner, DarkBox, ClearBox, WhiteBox, and PinkBox. Of these only Block, Cove, and Corner can be grasped by the robot. Due to its size and shape, both arms must be used to manipulate Cove.

2.4 Conveyor

A small conveyor shown in Figure 2.5 is used to deliver objects to the workspace. The conveyor can be placed anywhere in the workspace, and is also tagged with a unique 3-LED pattern so that the system can know its location. The conveyor is actuated with a Clifton precision motor (see Table 2.2). The motor has an encoder which is used to close a velocity loop. The gear-ratio was chosen to allow a maximum conveyor speed of 0.5 m/s. Due to friction, the minimum conveyor speed that can be regulated is about 0.01 m/s.

name	*shape*	*purpose*	*mass*
block	prism: base 7.4x5.8 cm, height 17.5cm	Light object graspable with a single arm	0.53 Kg
corner	L-shaped: main body 16×6.4 cm, side 9.6×3.2 cm, height 15.5cm	heavier convex object graspable with a single arm	0.69 Kg
cove	U-shaped: main stretch 33×3.2 cm, sides 16×3.2 cm, height 15.5cm	Large convex object. Grasp requires 2 arms	1.05 Kg
bar	cylinder 43.2cm long, 2.7cm diameter	Long thin object. Grasp requires 2 arms	0.45 Kg
PinkBox	prism: base 18.6×11.2 cm, height 10cm	Pure obstacle	N/A
WhiteBox	prism: base 31.6×11.2 cm, height 10cm	Pure obstacle	N/A
ClearBox	prism: base 38.2×15.1 cm, height 10.2	Pure obstacle	N/A
DarkBox	prism: base 38.2×15.1 cm, height 10.2	Pure obstacle	N/A

Table 2.4: **Characteristics of the workspace objects**

2.5 Vision System

An overhead vision system allows tracking of the objects and robot manipulators. This system consists of an overhead camera (Pulnix 440S CCD), a VME-based vision board and a general-purpose VME-based processor dedicated to processing the vision data. The camera operates in non-interlaced mode, generating 60 frames per second. It is equipped with an internal electronic shutter which limits the frame exposure to 1/1000th of a second, eliminating blurring due to the object motion. The camera is mounted 2.83 m above the task table, and uses a Cosmo C-6 6mm lens to cover a field of 3m x 2.25m at the table level. An infrared filter is mounted on the lens so that the camera only “sees” the LEDs. The VME-based vision board *Point Grabber II*, was designed and built by Chen [29]. It provides a list of pixel coordinates and intensity values corresponding to the points in the image field above a certain (programmable) threshold. It is controlled through the VME backplane from a dedicated processor board (Motorola MVME-162, 68040-based single-board computer). This dedicated vision processor runs the *VisionServer* software developed by Schneider [154]. The VisionServer software tracks objects in the horizontal plane using their

known unique LED patterns and height, and sends the updated position to any clients at the frame rate[3]. The software can also track individually-described points once they are initially located.

2.6 Computer System

The computer environment combines dedicated single-board computers running the VxWorks real-time operating system with several general-purpose unix workstations. This gives the best of both worlds: an expandable, low overhead computer system for the real-time tasks, combined with the flexibility and tool-richness of Unix-based workstations.

Figure 2.6 shows the system schematically. The dedicated real-time computer system consists of VME-bus single board computers, the vision board and a number of I/O boards. The unix environment used anywhere from 1 to 3 Unix workstations (Sun Sparc-Station IPX, Sun Sparc-Station 10-51 and DEC Alpha). All development was performed on unix workstations, including cross-compilations for the real-time targets. All computers are connected in an Ethernet local area network which is also used by the real-time processors to load the executable code and access the file system (using NFS). This architecture was originally conceived by Schneider and Ullman [154, 189] and has been used with success in a number of other experiments at the Stanford Aerospace Robotics Laboratory. A detailed list of all the boards connected to the VME backplane is presented in Table 2.5.

[3]This software uses TCP/IP network communications provided with the VxWorks operating system to send its updates to processors in the same backplane or anywhere on the network

Slot	*Component*	*Model*	*Description*
1	Single Board Computer	MVME 167-1	33 MHz 68040 CPU, 4 MB RAM
2	Single Board Computer	MVME 167-1	33 MHz 68040 CPU, 4 MB RAM
3	Single Board Computer	MVME 167-1	33 MHz 68040 CPU, 4 MB RAM
4	Single Board Computer	MVME 167	25 MHz 68040 CPU, 4 MB RAM
5	Interface Module	MVME 712	Serial, Ethernet I/0 for board 1
6-8	Interface Modules	MVME 712	Serial I/0 for boards 2-4
9	Vision Board	ARL-PG II	Point Grabber with FIFO's
13	Analog Input	XVME-566	16 Channel differential 12 bit A/D
14	Analog Output	XVME-505	4 Channel, 12 bit D/A
15	Analog Output	VMI-4116	8 Channel, 16 bit D/A
16	Digital I/O	MVME-340	programmable multiple-channel digital I/O

Table 2.5: **Real-Time Computer Components**

The real-time computer components are connected on a VMEbus. Four single-board computers are used for real-time control and sensor processing. The remainder of the bus is occupied by interface and I/O boards.

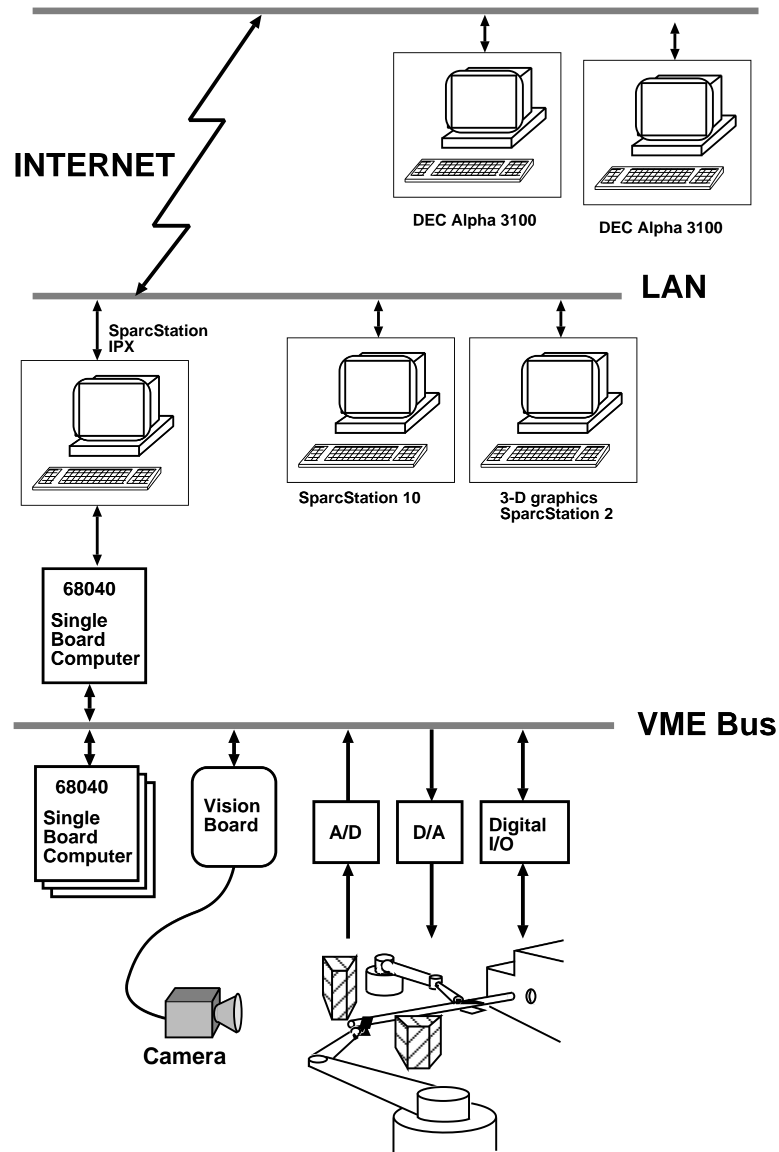


Figure 2.6: **Hardware Architecture**

The computer system combines a number of Unix workstations and a VME-based real-time computer system distributed over a network. The real-time computer combines four single-board computers, a vision board and both analog and digital I/O boards.

Chapter 3

Architecture, Interfaces and System Operation

> "It is common sense to take a method and try it. If it fails, admit it frankly and try another. But above all, try something."
> `Franklin D. Roosevelt` – *Speech at Oglethorpe University, May 22, 1932*

> "The greatest leverage in system architecting is at the interfaces."
> `Eberhardt Rechtin` – *Systems Architecting* [146]
> "The greatest dangers are also at the interfaces."
> `Arthur Raymond` – *Speech at USC, 1989* [146]

To be manageable, large systems need to be built around a solid architecture or framework. The advantages of modular system design are well established: simplicity (a large system is broken into smaller, simpler components which can be understood and developed in relative isolation), maintainability (failures can be isolated into the originating module(s) where they can be more easily addressed), reliability (redundancy can be introduced with replicated/redundant modules). However, modularity does not come for free. The design of the module interfaces and the interconnection of the different modules (i.e. the systems integration) becomes a large part of the overall development effort.

This chapter presents the system architecture and interfaces of the Robotic Workcell and the new concepts and methodologies introduced in this research to address the integration problem:

1. The "interfaces-first" system-design technique—similar to the approach taken in large software projects. It differs from the more classical "subsystems-first" approach in that the

interfaces are developed for the information flow in the system (as opposed to developing interfaces for the subsystems themselves). The subsystems then become users and providers of that information to collaborate towards the system goal.

2. Introduction of *modularity* at the interface level. The interfaces themselves are designed to be composed of multiple primitive modules which can be connected in a variety of ways to create new "custom" interfaces for each of the subsystem modules.

3. *Anonymous* interfaces where information is exchanged without the subsystem knowledge of the identity of the source/destination or even the number of subsystems involved in the exchange, creating in effect the illusion of a common information backplane.

4. Development of specific interface semantics appropriate for integrating remote on-line planning subsystems (with unpredictable planning and communication delays). In particular the use of strategic-level commands to interface the planner and control subsystems.

5. *Stateless* interfaces and subsystems that are robust to temporary disconnection, live-insertion or re-initialization of any subsystem and many-to-many interaction modes.

3.1 Literature Review

Many authors have looked at the architecture of robotic systems and tried to formalize "general approaches to the design of such systems". A detailed description of all the proposed architectures would take far too much space. For this reason, we limit the discussion to architectures that have been implemented and verified in experimental robotic systems.

The National Institute of Standards and Technology (NIST) has defined a reference model for Robotic System Architectures. RCS [139, 6, 7], and its telerobotics counterpart, NASREM [4, 5], build systems by interconnecting generic controller modules. Each module incorporates three functions: Sensory Processing (SP), Behavior Generation (BG) and World Modeling (WM), in effect creating three interconnected hierarchies. All modules communicate through global shared-memory. Modules execute periodically at fixed rates. The lower levels of the hierarchy take care of servo-level control and sensing issues, while the highest levels perform mission planning, high-level user interfaces, etc. RCS's approach is, by design, a strict hierarchy in the sense that each controller-module is restricted by the design to command a set of subordinates and receive status reports from them. The hierarchical design of RCS results on a tree-like topology which, according

RCS's designers, allows the assignment of responsibility and more predictive behavior than an agent-like architecture. RCS also assumes that global-shared memory semantics can be achieved in a distributed environment, and prohibits the use of interrupt (event) driven programming techniques in favor of periodic execution and polling (which the authors claim is more predictable).

Butler and Jones [134, 26] have developed an architecture called MICA (Modular Integrated Control Architecture). Their approach differs from RCS in that it provides support for both asynchronous (event-driven) interactions as well as synchronous (periodic) execution. MICA modules are interconnected with two mechanisms: HELIX [74, 73] (a distributed blackboard for synchronous-data exchange) and a dedicated event-network. Events are fixed-format messages, and are addressed to specific modules. Whenever an event cannot be handled locally it is broadcasted to all other modules. MICA is not a strict hierarchy like RCS, as any interconnection topology is allowed. MICA has been used to control the mobile robot HERMIES-III [134].

Hierarchical approaches use each layer to simplify and abstract the problem for the layer above, allowing the higher layers to be progressively more isolated from the actual hardware. The hierarchical approach structures and simplifies the design of the system, and allows the higher layers to be portable to different hardware. Hierarchical approaches are routinely used in control systems in the form of successive loop closures [50]. Hierarchical approaches have also been used to incorporate planning and control in a "command hierarchy" ranging from high-level (mission-level) tasks, down to servo commands. For instance Chatila, Herrb, Fleury and Noreils [119, 49, 118, 120] propose a three-level hierarchy composed of planner, control and functional layer. The planner level uses petri-nets to coordinate among multiple robots and to break tasks into smaller operations for a single robot. The control layer (a misnomer in my opinion) breaks down the commands into primitives, schedules them with the functional layer, and supervises their progress. Lastly the functional layer is responsible for sensor processing and feedback control. This architecture has been used to control the HILARE mobile robots.

Hierarchies are not the only model for interconnecting subsystems. For instance Zanichelli, Caselli, Natali and Omicini [204] propose an Agent-Based architecture where all agents communicate though a central white-board-like "world-model". The robot itself is just another agent. This is proposed in contrast with "layered" approaches or behavior-based architectures. They seem to advocate the "world" to keep models on behalf of the agents, but also allow the agents to do their own modelling. Communications are performed in a language similar to distributed Prolog containing primitives similar to those of LINDA[3]. Agent-based architectures are normally large-grained[1],

[1] That is, each agent is quite complex by itself.

each subsystem (agent) can itself be developed as a hierarchy and incorporate its own models. A fundamental difference with the hierarchical approach is that the agent approach treats other agents as peers, and any interconnection topology is allowed.

Autonomous mobile robots can be very complex systems as they integrate planning, sensing, and locomotion, and often operate in unstructured environments. Their complexity has motivated the development of many control architectures. Hinkel, Knieriemen, and Puttkamer have introduced an architecture composed of four subsystems (locomotion, sensor system, man-machine interface, and control system) to control the mobile robot MOBOT-III [63]. These subsystems are arranged as two parallel hierarchies: the command hierarchy (man-machine interface, control and locomotion subsystems), and the sensor processing hierarchy (sensors, combiner, map-building). The sensor information is processed synchronously, while the commands are issued asynchronously (presumably any synchronous feedback loop is handled locally by the lowest locomotion layer). Although the architecture is quite specific to their application, it reinforces the idea that command hierarchies are more naturally described as asynchronous or event-driven. Arkin proposes an autonomous robotic architecture (AuRA) [13], composed of planning, cartographic subsystem (world-modeling), motor subsystem (control), and homeostatic control (internal communications/broadcasts and change in control modes). AuRA introduces the idea of *schemas*: artificial potential fields instantiated by the planner to guide the behavior of the control subsystem (e.g. move-to-goal, avoid-static-obstacle, stay-on-path) without committing the controller to specific motions. Multiple schemas can be concurrently active, and their control commands are summed. This architecture bridges planning and reaction by providing mechanisms that allow the planner to “program” the reactive behavior of the system.

Other authors have concentrated their work in the higher layers of the command hierarchy, and have studied ways of plan generation, task sequencing and decomposition. Bonasso and Slack [21] propose a three-layer hierarchy for planning: deliberation, sequencing/scheduling and skills. The Task Control Architecture developed by Simmons [171, 169] uses task-trees and exception mechanisms to sequence and monitor the operation of a robotic system, and has been used to control several autonomous robots [170, 85]. Nakamura and Xu’s Intelligent Multi-sensor-based Robot Controller [115] approach is based in the concept of *skills* (operations that the robot is capable of performing). Each skill contains a description of how to achieve in terms of primitive actions and other skills. This relationship between skills constitutes an and-or graph that is used in the planning process. Production rules are used by other architectures, including Soar developed by Rosenbloom, Laird, Newell and McCarl [148].

Modular decomposition and interconnection topology are only two aspects of the architecture of a system; a related, critical aspect is the interfaces between the modules. That is, what are the syntax (format) and semantics (meaning) of the information exchange. The interface issue is not unrelated to module development, because the interfaces can characterize the modules as far as the remaining subsystems are concerned. Although in principle the interface is abstract and does not imply any specific implementation, in practice, interface definition severely constrains the possible module implementation and vice versa.

Object-Oriented Programming (OOP) provides facilities well suited for interfaces-first design and the construction of modular interfaces. Abstract primitive interfaces can be encapsulated in individual classes, and methods and multiple inheritance can be used to build a dedicated interface from the primitive elements. However, uses of OOP in robotic applications have used the objects to model individual subsystems not the information between them, yielding a subsystems-first approach. For instance, RIPE [109][2] defines a hierarchy of objects to describe the different physical elements in the system. Their architecture has four layers (task-level programming, supervisory control, real-time control and device driver). Within the object hierarchy, the object class definitions become interface specifications for the subsystems, but no abstract interfaces are provided for accessing sensor or world-modeling information, nor for task specification.

The interface between planning (task or path planning) and the control system is particularly critical because of the fundamental impedance mismatch between the two subsystems. Planning is asynchronous, event-driven, and can potentially take an unbounded amount of time. Plans tend to be geometrical in nature. On the other hand, a control system is a periodic sampled-data system, requiring a continuously available reference state (reference-state trajectory). External events (such as an unexpected obstacle, or the motion of an object on a conveyor) require real-time responses (i.e. with definite, bounded response times) which cannot be guaranteed by deliberative planners. As a result, some authors advocate abandoning planning in favor of "reactive" approaches, such as Khatib's potential fields [80], which have no deliberative phase, while others propose a planner-controller interface that incorporates reactive information. For instance, Payton, Bihari, Rosenblatt and Keirsey [127, 128] propose an interface where the output of the planner is a potential field that can be used to guide the robot. Quinlan and Khatib [137, 138] have proposed solving the mismatch by introducing an on-line algorithm that can continuously deform the path originally created by the planner as if it were a rubber band subject to repulsive forces by obstacles in the environment. All these methods attempt to achieve the correct balance between local reactive behavior (fast but

[2]RIPE stands for Robot-Independent Programming Environment, and was developed at Sandia National Laboratories.

sometimes misleading or without purpose) and deliberative goal-oriented planning (slower and computationally unpredictable but with a global goal-oriented perspective).

The KAMRO workcell [1, 67, 95] combines an on-line planner (FATE) and a real-time control subsystem. The input to the FATE planner is a petri-net describing the assembly task. FATE interprets the net and sends elementary commands such as (TRANSFER, FINEMOTION, GRASP) to the real-time controller. FATE monitors progress and has facilities for error analysis and recovery. FATE plays a role analogous to the planner-subsystem in our architecture. A fundamental difference of our system is that the commands generated by the planner are *strategic commands* not elementary operations. These commands are further decomposed into elementary operations by a finite-state machine facility *within our control subsystem.* In this manner, our system performs monitoring and error-recovery both within the control and planner subsystems. This extra layer allows our system to deal with moving objects and obstacles which KAMRO cannot handle.

NIST and NASA's JPL have embarked in an ambitious effort to define interfaces for all telerobotic applications. The Unified Telerobotics Architecture Project (UTAP) [99] defines an architecture composed of modules of 20 different types (operator-input-device, object-modeling, task-description-simulation, task-program-sequencer, sensor-control, robot/axis-servo-control, etc.), and specifies the interfaces to each one of the modules. Modules accept similar messages to set or get parameters and to control their operation (set, get, start, stop, post etc.), but the parameters differ among modules. Modules are connected on a strict hierarchy, and messages can only be exchanged between a parent and its children with the exception of the Object-Knowledge module (similar to a world modeler) which can exchange messages with all modules. The interface has provisions for requesting modules to post periodic updates at a given rate (but only a single rate/destination can be requested). Their philosophy differs from the one presented here in that interface has state (modules assume that previous messages have been received and processed), message exchange is not anonymous (sender must specify receiver), and messages are strictly hierarchical. Moreover, in UTAP there is a fixed, pre-established number of modules of each module-type (a single instance for most types). In several respects however, UTAP's interfaces are more powerful than ours since UTAP has language constructs which allow the creation of macros and message groups.

Previous work at the Stanford Aerospace Robotics Laboratory has addressed some of these interface issues. Schneider [154, 157] used strategic-level commands to interface a dual-arm robotic system with a graphical interface, but did not address the interfaces to planning subsystems. Ullman also used strategic-level commands to direct a dual-arm free-floating vehicle from a graphical

interface [189], and made the command-interface stateless, but his system was mostly self-contained and did not incorporate planning.

Several experimental multi-arm robotic workcells have been described in the literature. Fu [51] has developed a scenario similar to ours which deals with two robots and multiple moving objects on a conveyor belt. In this work, the assumption is made that only the last links of the arms may collide with each other, and that the parts are fed slowly enough that the robots can always pick up the objects from locations that do not deviate too much from their nominal positions. Our system does not have these restrictions.

3.2 Approaches to System Design

Modular system design is a well-established methodology. This approach breaks a large system into smaller, well-defined modules with specified functionality, which then can be developed and tested independently from the others.

While modularity facilitates the development of the individual components, the need to develop the interfaces (“glue”) between components makes the complexity of the overall system much greater than the sum of its parts. By careful selection of the functional boundaries, the resulting subsystems can be mostly independent, and the total system complexity can approach the ideal limit of the sum of its parts.

Subsystems-First. Traditional modular system design (*subsystems first*) follows the cycle illustrated on the left side of Figure 3.1. Following the analysis of the complete system, functional units are identified, and the system is broken into subsystems across these functional boundaries. These subsystems are further defined by their interfaces to the outside world, which specify their observable behavior. Once the subsystems have been developed and individually tested, custom interfaces (glue logic) are developed to interconnect the various subsystems. This cycle results in a complete system that can now be tested against the design specifications. This process repeats itself on each design iteration. Once the subsystems are designed, the overall system is integrated by developing appropriate interconnections between the subsystems (essentially building glue-logic that matches the different interfaces).

This *subsystems-first* technique requires the scope and functionality of the overall system to be known in advance. It emphasizes subsystems rather than their interfaces. It is therefore most appropriate for “sparsely-interconnected” one-of-a-kind systems that, once designed, are unlikely

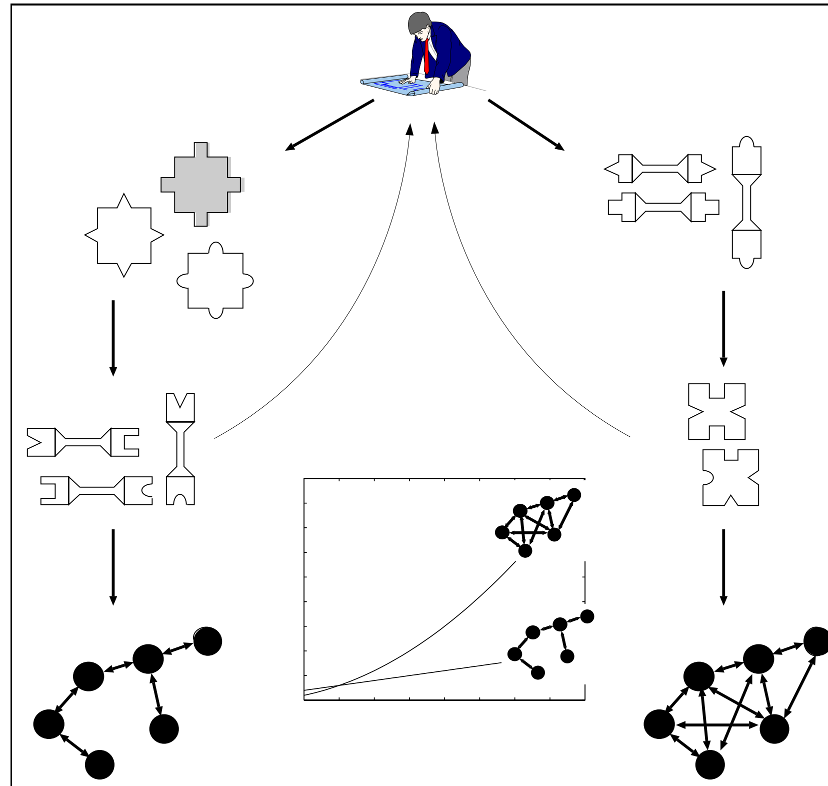


Figure 3.1: **Contrasted Design Techniques**

Subsystems-first design, on the left, concentrates on developing functional units. Information flow between the units requires custom interfaces. Interfaces-first design on the right starts with the information flow. Functional units are then tailored to the interfaces developed. Interfaces-first design is more appropriate when subsystems are highly interconnected. In highly-interconnected systems, the number of interfaces increases quadratically with the number of subsystems.

to be expanded with the addition of new subsystems. For example, a monolithic electronic board (say a graphics board) is designed as a unit from the onset, and the different functional units (CPU, video memory, graphic accelerator hardware, etc.) are designed with their own interfaces, and require custom glue-logic to be connected to each other within the board.

While the subsystems-first technique works well for systems in which subsystems have a low degree of interconnectivity, there are other systems whose full scope is not known in advance and/or that require a much higher degree of interconnectivity. This second class of systems is hard to design with the subsystems-first approach.

Interfaces-First. *Interfaces-first design* [125] is a methodology more appropriate for open-ended systems with a high degree of subsystem interconnectivity. In interfaces-first design, the types

of information flow between the subsystems (whatever they turn out to be) are identified from the outset, and interfaces for the information (*not* the subsystems) are designed first (*information-interfaces*). Later, subsystems are designed that use these interfaces to communicate and interact. This design is more typical of large software projects or computer systems. For example the busses in different computer families (e.g. PC bus, VME bus etc.) are designed first, and the different boards that interact through these busses are designed later to conform to the interface specification. In fact, new boards are designed at a later stage with functionality that was not initially anticipated. The interfaces-first design cycle, shown down the right side of Figure 3.1, is most appropriate for "heavily interconnected" systems and those that are "open-ended" in the sense that new pieces will have to be incorporated at a later stage.

The interfaces-first approach may seem less intuitive at first. After all, does not the information exchanged depend on what the subsystems are? The extent of this dependency varies depending on the characteristics of the system under development. If the overall function and requirements of the system are well known in advance (and unlikely to change much), and if the system is loosely connected (i.e. each subsystem will be connected to one or two other subsystems), the subsystems-first approach is probably more intuitive and simpler. However, many systems do not belong to this category; we may not be able to know in advance the full scope, and we may want to leave the door open for enlargement by future (not necessarily well-characterized) subsystems. For instance, in the design of a computer window system, the designers do not know (or expect to know) all the applications that will be run on top. However, one thing they do know is that these applications will need to exchange data by cut-and-paste-type operations. The window-system designers define interfaces to this information (e.g. text, graphics, voice, video) and specify its format. Current applications (word processors, databases, spreadsheets etc) and future ones use these interfaces to exchange information.

The "interfaces-first" methodology is better suited for strongly inter-connected systems and "open" systems that may expand in the future. Examples of this approach may be found in computer-bus design (SCSI, IDE, VME), desktop window system design (Macintosh, Windows), intra-tool protocols (ToolTalk), etc.

Although presented as "alternatives," the two approaches are the extremes of a continuum of design methods where the adequate mix depends on the characteristics of the system to be designed.

Traditionally, robotic systems have been designed with the subsystems-first technique, which has made extension and development of the system difficult due to their limited interconnectivity and the considerable amounts of custom interfacing required. To the author's knowledge this is the

first time the "interfaces-first" approach is recognized as an alternative design approach and applied to a robotic system.

3.3 Modular Interfaces

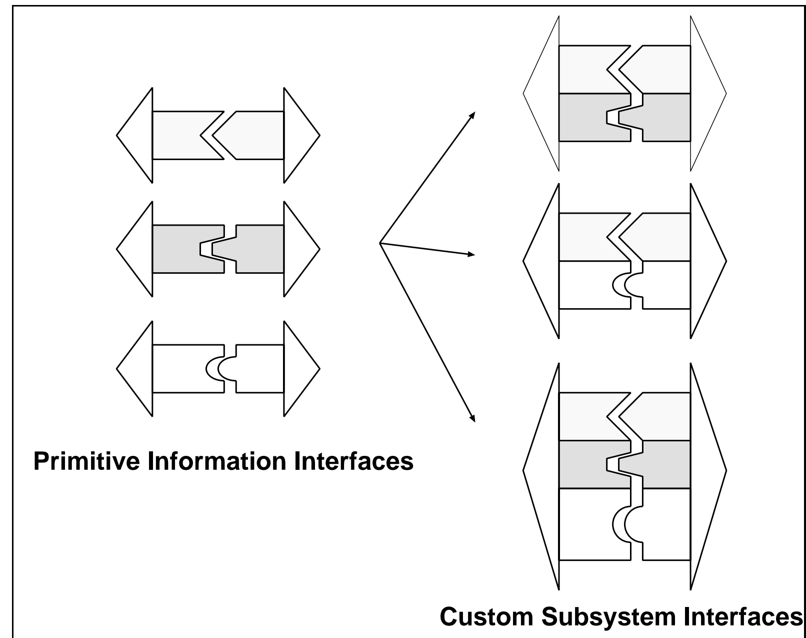


Figure 3.2: **Modular Interface Design**

First, the fundamental categories of information flow are identified, and information interfaces are designed for each (on left). Subsystem interfaces are then built from combinations of these primitive information information interfaces.

Design of the information interfaces is not a trivial task for robotic systems, as the information/command flow is much less structured (i.e. higher level) and varied than that modeled with a bus-type interface. For example, all the devices connected to a SCSI bus are expected to be able to operate at (almost) the same speed and accept the same type of commands. This is a poor model when the "devices" include things like graphical interfaces, teleoperator-input devices, planning subsystems, and sensor subsystems. Each of these subsystems requires fundamentally different types of information, and is able to process it at significantly different speeds.

The author proposes to address this problem with a "modular" approach to the design of the information interfaces themselves. This method first identifies the fundamental properties of the information flow. The information is then divided into categories based on the characteristics of the information-flow itself (bandwidth, persistence, idempotency). Next, primitive interfaces are

designed for each type of information flow. These primitive information interfaces can then be combined to create custom "compound" subsystem interfaces. Figure 3.2 illustrates this process.

3.4 Information Interfaces for the Manufacturing Workcell

In the context of a robotics workcell, we must examine world-state information flow, command flow, control signals, etc., identify parameters that characterize the information, and build primitive interfaces around each information category. This section examines the system-architecture design for the smart workcell project using the interfaces-first technique.

interface	*nature of the information*	*selectable parameters*
world-state interface	periodic, idempotent, unreliable, last-is-best, persistent until the next update	update rate, specific information desired
system-command interface	asynchronous, reliable, in-order delivery (exactly once), persistent until completed	priority (to resolve conflicting requests)
task-specification interface	asynchronous, persistent until cancelled, reliable but not necessarily in order	priority

Table 3.1: Characteristics of the information flow

Three types of information flow can be distinguished in this application: (1) world state, (2) system commands, (3) and high-level task descriptions. Table 3.1 summarizes the three primitive interfaces and selectable parameters for these interfaces.

3.4.1 The World-State Interface

World-state information includes data such as object positions and velocities (on and off the conveyor), arm locations, joint angles, and force signals. By its very nature this information is either *static* or *periodic*, and requires continuous updating because it corresponds to physical quantities that change over time. In addition, the information is *persistent* until it is invalidated by a new update of the same information.

World-state information also has *idempotent* semantics—getting two identical updates of the same information (say the position of an object) causes no side-effect, and is logically equivalent to a single update. Moreover, the natural semantics are *last-is-best*, i.e., we are always interested in the most recent value, even at the expense of missing intermediate values. The information

content and frequency of each specific instance of the interface must be *customizable*, as different subsystems require different subsets of the overall world-state information at widely different rates. For instance, a slow graphical interface animating the robot position may need new joint angles at only 10 Hz; a control or estimator subsystem will need updates at a much higher rate.

Object-state information	
location	object location in the global reference frame.
grasps	possible grasp locations within the object
properties	mass, inertia, limits on acceleration etc.
shape	shape (collision avoidance, graphics)

Robot-state information	
location	robot-base location within workcell.
joint val.	values of the joint coordinates (pos, vel, acc)
limits	kinematic (joint) and torque limits
kinematics	Denavit-Hartenberg parameters
status	moving, grasping an object etc.
trajectory	via-point and timing of the current arm trajectory

Table 3.2: The world state interface

The specific information encapsulated within the world-state interface is summarized in Table 3.2.

3.4.2 The System-Command Interface

System-command information is quite different from world-state information. First, commands are *asynchronous* and "hold their value" only until their completion (even if no new command is received). Commands are *not* idempotent, and must be delivered reliably exactly once and in the right order. For instance, if we are sending a trajectory to the robot, sending the trajectory twice will cause the robot to execute two motions. Similarly, missing an intermediate command (such as closing a gripper before lifting an object), is not acceptable.

In this thesis, no attempt has been made to create a comprehensive command language to cover a wide range of situations. Rather, the goal has been to investigate the specific issues that arise when a remote planner interacts with a workcell operating in a dynamic environment. These issues have been addressed experimentally within the context of the objectives described in Chapter 1.

Robot commands through the command interface	
move object	Move an object that is being grasped. This command will provide a via-point collision-free path for the object. The arm(s) is controlled using object impedance control [155]. This command also provides earliest-time constraints for the time of traversal of each via-point
move and release	Same as above with the addition of a specified release location (must be close to the end of the trajectory)
move arms	Move one or both arms. The arm(s) to move cannot be holding an object. Specifies a via-point collision-free path both in operational and joint-space so that the robot controller is free to use different control schemes (and kinematic ambiguities can be resolved). This command also contains earliest-time constraints on each via point.
move and grasp	Same as above with the addition of the specification of the object(s) to be grasped and the corresponding arm(s), and grasp(s) location(s) for each arm involved (any combination of arm/object, including both arms on the same object is allowed). This command also contains earliest-time constraints on each via point.

Table 3.3: System command interface

The specific commands that can be sent through the command interface and respective definitions are summarized in Table 3.3. Note two important aspects of the semantics of these commands: First they are *strategic-level* commands, and second, they allow description of *timing constraints* to be met by the workcell.

Strategic commands. The system commands are strategic-level, i.e. they specify multi-step actions without completely constraining the details and timing of these actions. Strategic commands require further processing, and must be decomposed into primitive actions by the strategic-level controller within the control subsystem (see Chapter 6). For instance, the command to move and capture an object only specifies a geometric path (and timing constraints) for the arm to approach the object, as well as the required grasp transforms. However, the details of the interception (velocity and acceleration matching, closing the gripper etc.) are left to the control subsystem. Similarly, the move-and-release command specifies a path to approach the release location and the desired final location for the object; fine motion in the release is left again to the control subsystem. Commands of this type could easily be extended to handle interaction with other objects and actual assemblies, because the (control-system specific) details requiring high-bandwidth feedback,

assembly strategies, force control, etc. are left to the control system. The development of these types of primitives is the subject of other work at the Stanford Aerospace Robotics Laboratory [150, 152].

Description of timing constraints. Interacting with a dynamic environment requires the planning subsystem to account for and communicate timing information to the control subsystem. For this research, a simple interface has been chosen. Each via-point[3] sent by the planner, is tagged with an "earliest-time" stamp. It is the planner's responsibility to ensure that any trajectory that meets this constraint is safe (see Section 3.5.5). The control subsystem will generate a trajectory that, aside from satisfying the dynamic constraints of the system, also satisfies the imposed "earliest-time" constraints. This is achieved in practice by delaying initiation of the trajectory until all constraints are met. This approach makes the system robust to unpredictable communication delays, which occur when the planner executes at a remote location over the internet, and also accounts for the fact that the planner executes on a non-real-time computer system. However, it is clear that this approach has limitations. In particular, it restricts the set of viable plans to those whose execution can be safely delayed, and also allows useless motions to proceed (e.g. the arm goes to a position on the conveyor to grasp an object well past the time when the object was there). One can contemplate adding a "latest-time" constraint to each via point. In fact this ability already exists in the system, but the current planner is not able to take advantage of it. Clearly much research remains to be performed in this area.

3.4.3 The Task-Specification Interface

High-level task commands fall somewhat between the extremes represented by world-state and system-command information. Task commands are *asynchronous* like the system commands. However, tasks are in a sense self-contained (i.e. specify a goal or desired system state) as opposed to specifying an action. As a result, tasks may be *persistent*. For instance, a task may specify to repeatedly pick objects from a conveyor, immerse them in a solution, and then place them on a second conveyor. This task does not end until explicitly cancelled. Multiple concurrent tasks may be active at the same time and compete for the system's resources. Tasks may also be built as logical combinations of smaller subtasks (e.g. put object O1 at location L1 and object O2 at location L2). Table 3.4 summarizes the task interface.

[3]Via-points (a.k.a. through-points or way-points) specify intermediate configurations (poses) for either the manipulators or the objects being manipulated. Via points are discussed in detail in Chapter 7.

Task commands through the task interface	
place objects	Place a set of objects at their specified destination. The objects may not even be in the workspace. The system must get the objects however it can, place them at their goal, and monitor to make sure they remain there until the command is cancelled.
make assembly	Similar to place objects, except the object goals are now in contact with each other.

Table 3.4: Task-specification interface

3.4.4 Custom Interfaces for the Manufacturing Workcell

The three primitive interfaces previously discussed constitute the information building blocks (modules) from which custom interfaces can be built to connect the different subsystems. These interfaces are *anonymous* (no assumptions are made regarding the number or the specific subsystems that will be connected through them). Instead, the primitive interfaces provide a mechanism for subsystems (whatever they are) to share and communicate information. Since the primitive interfaces encapsulate all the information required to monitor and direct the robotic workcell, they can be combined to provide all the information and interaction modes required by any subsystem.

Anonymous interfaces have several advantages: they allow the system to be easily extended and modified, because the subsystems are not hard-linked to each other. They also provide for subsystem replication (hot-standby, multiple monitoring stations) without additional effort. However, anonymity does have costs, mostly concerning safety and security. Since the interfaces are anonymous, there is no mechanism to specify (or restrict) who is allowed to access which information. Moreover, a negligent (or malicious) subsystem can corrupt the whole system by providing incorrect data. These tradeoffs must be carefully evaluated for each application.

The different subsystems in the workcell use the three primitive interfaces to develop custom interfaces for each of the subsystems. For instance the graphical user interface (GUI) (see Section 3.5.1), uses both the world-state and the task-specification interfaces. The GUI customizes the world-state interface by selecting the specific information required for graphical rendering, and subscribing to the rendering information at a rate slow enough to be handled by the three-dimensional graphics program. The GUI also uses the task-specification interface to describe the high-level user requests to any subsystem interested in this information (the planner in our case). The combination and customization of these two primitive interfaces constitutes a unique interface tailored to the

needs of the GUI. The remaining subsystems create their corresponding interfaces in an analogous manner.

On occasion, a new subsystem may be developed that requires different information, or a new sensor added that provides some other type of information. This could be handled by either augmenting one of the primitive interfaces or, if the characteristics of the information do not fit within any of the existing interfaces, a new primitive interface can be created. However, this addition should have minimal impact on the existing interfaces.

3.5 System Architecture

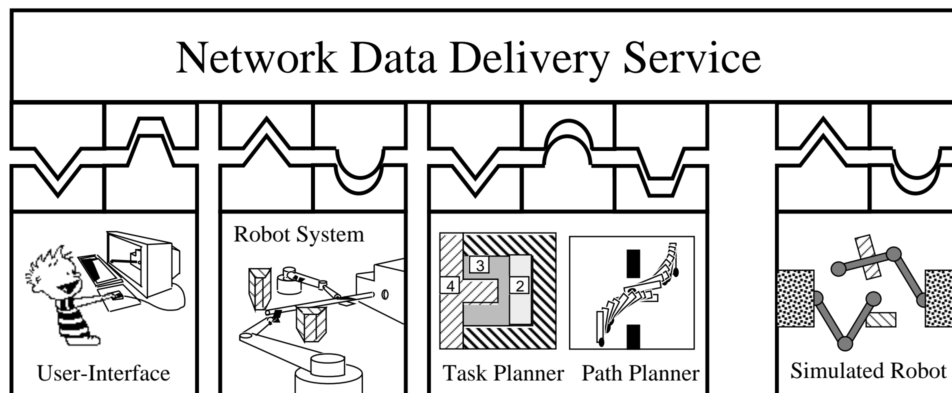


Figure 3.3: **System Architecture**

Four subsystems collaborate to operate the workcell. Each subsystem uses a combination of the primitive information interfaces to connect to the information flow. The interfaces are represented by connection icons: world-state information (triangle icon), task-specifications (trapezoid icon), and system commands (semi-circle icon). All the interfaces are built on top of a "software bus" called the Network Data Delivery Service (NDDS) [see Chapter 4].

The overall system architecture is illustrated in Figure 3.3. The system has been broken into four subsystems: (1) Human interface, (2) robot-control system, (3) task and path planners, and (4) robot simulator. The human interface receives world-state information and provides a graphical representation of the scene to the user. During operation, the high-level task is also specified using the human interface and sent to the planners. The task planner/path planner receives continuous updates from the robot and task requests from the user, and produces robot commands to implement the requested tasks. The robot controller executes these commands and processes the sensor signals to create and maintain a world model. The simulator can masquerade as the robot control system during development or, by running it concurrently with the robot, can be used to compare the

accuracy of the workcell models (kinematics, dynamics and state transitions) with the behavior of the actual system.

The three primitive interfaces described in the previous section are used to communicate between the four subsystems. For instance, the human-interface uses the world-state interface to obtain the kinematics and position of the arms and workspace objects in order to display them to the user. Tasks are sent to the planners through the task-specification interface. The planners also use the world-state interface at a lower bandwidth than the human interface.

The actual implementation of the interfaces over a computer network is a significant undertaking in itself, and is the topic of Chapter 4. The fundamental challenge is to provide efficient and transparent network connectivity consistent with the characteristics of the information encapsulated by the interfaces. The resulting outcome is similar to a "software bus," where the three primitive interfaces provide the means for accessing the bus. This architecture allows functionally identical subsystems to be replicated and new subsystems to be added to create different configurations.

The different subsystems are described in more detail in the following sections.

3.5.1 The Graphical User Interface

The purpose of the user interface is to provide an intuitive, easy-to-use mechanism for the user to command and monitor the workcell. Simple graphical user interfaces for robotic systems have been developed by previous researchers at the Stanford Aerospace Robotics Laboratory [154, 189], while ongoing research in collaboration with NASA Ames Research Center has explored the use of Virtual Reality [45, 179]. Researchers at other institutions have also addressed the human-robot interface issues [28, 64]. This work differs from graphical robot simulation-and-programming environments such as SILMA's CimStation [37], DENEB's IGRIP [16] and Tecnomatrix's RoboCAD [197], which are mostly used for off-line programming and are not directly connected with the running robotic systems. However, these programming systems could potentially be used as the graphical front to the user interface. This approach is being pursued by other researchers [193].

The fundamental difference in our approach is that the graphical user interface (GUI) is employed at the task-specification level, that is, it is used to tell the workcell *what* the user wants done, not *how* to do it. For instance, with our interface, the human specifies the goal locations for a number of objects which may not even be in the workcell at the time. The remaining subsystems (planner, world modeler, controller...) will determine how to achieve the task[4] and collaborate to achieve the

[4]How to achieve a task will depend on the current state of the workcell, the arrival of the needed parts on the conveyor, etc.

final goal. The user interface also serves as a monitoring device that allows the user to observe the workcell in action. Figures 3.4, 3.5, and 3.6 show the user commanding and observing the workcell through the graphical interface. It is important to note that these are not simulations or off-line animations, but rather displays of live data arriving from the physical system during operation.

The GUI uses 3-D rendering to display the current state of the workcell from any view angle, which can be changed at the user's discretion. Rendering is performed using the GraPhigs package (Zimmermann [207]) which is built on top of PHIGS [18]. In addition it incorporates logic that allows the user to select any number of objects and move their "ghost" images around the screen, and in this way, build an interactive specification of the task. Once the user is satisfied with the specification, he or she issues the command by clicking on the command button within the GUI. The fundamental tenets of this interaction paradigm were formulated by Schneider [154].

The new features of the GUI developed in this research, are the ability to specify tasks involving multiple objects, and the possibility (due to its stateless nature and anonymous interfaces) of running multiple copies simultaneously, as illustrated in Figure 3.7.

This GUI has been utilized in many experiments to run the system remotely, where the human uses it as the only means of interaction with the system. The GUI uses the world-state interface to get (slow) updates from the system (at a rate that our 3-D graphics can handle, currently 5 Hz) and the task-specification interface to issue the user-constructed tasks. This architecture enables the use of multiple copies of the interface simultaneously. Multiple interfaces has allowed several users to monitor and collaborate with each other [see Section 3.6.1].

3.5.2 The Simulator

The goal of the simulator is to allow efficient testing and debugging of the other subsystem modules without using the actual robots[5]. Subsystems communicate using the anonymous interfaces already described; therefore, none of the remaining modules is aware it is interacting with a simulator rather than the actual robot. The purpose of the simulator is not to develop the control system, and therefore there is only need to simulate robot kinematics, not dynamics. In fact, the simulator uses the rigid-robot dynamics to generate the same reference trajectories that would be generated by the real robot[6], and then assumes that the robot exactly tracks the reference trajectory.

[5]Aside from risking damage, using the robots is more time consuming than using the simulator, and can only be done by one person at a time.

[6]These trajectories use the dynamics of the robot to ensure that the manipulators can track the trajectory without exceeding any actuator limits. The algorithm used to generate the trajectories is described in Chapter 7.

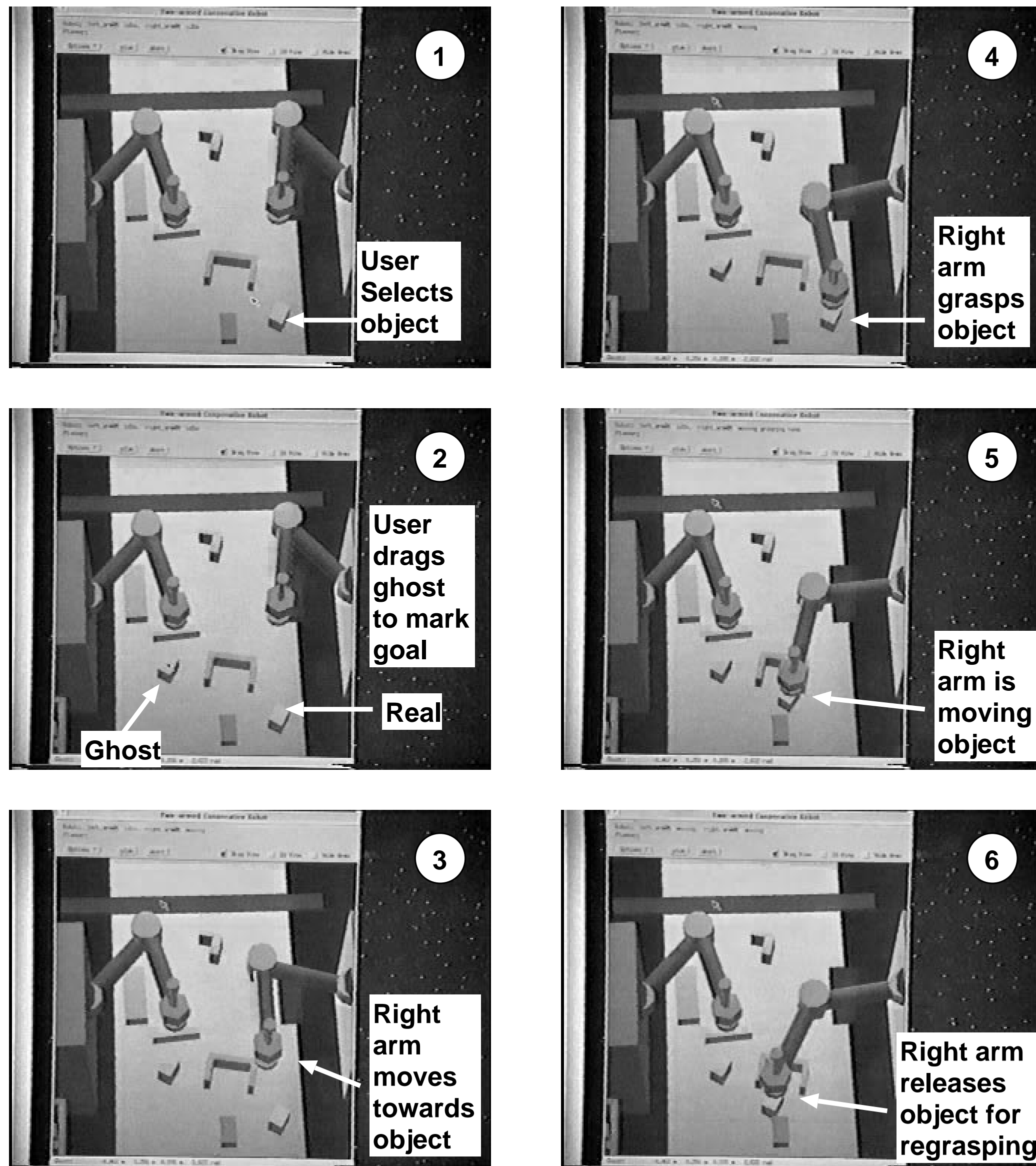


Figure 3.4: **Commanding and Monitoring the Workcell with the GUI (1-6)**

The above pictures illustrate the use of the GUI for task-specification and system monitoring. Note that the pictures represent a projection of a 3-D view, and the manipulators move above the objects in the workspace. The above sequence continues in Figures 3.5 and 3.6. First the user selects the "block object" (1) with the mouse. Selection of an object creates a ghost image which the user drags anywhere in the workspace to indicate the goal position for the part (2). Once this simple task has been defined, the user clicks the "command" button and the system figures out a plan and starts moving the right arm towards the object (3). The object is grasped in (4). Since the goal cannot be reached with the right arm, a hand-over operation is automatically performed. The hand-over consists of leaving the object at an intermediate location (5) and (6), and grasping it with the other arm (continued in Figure 3.5).

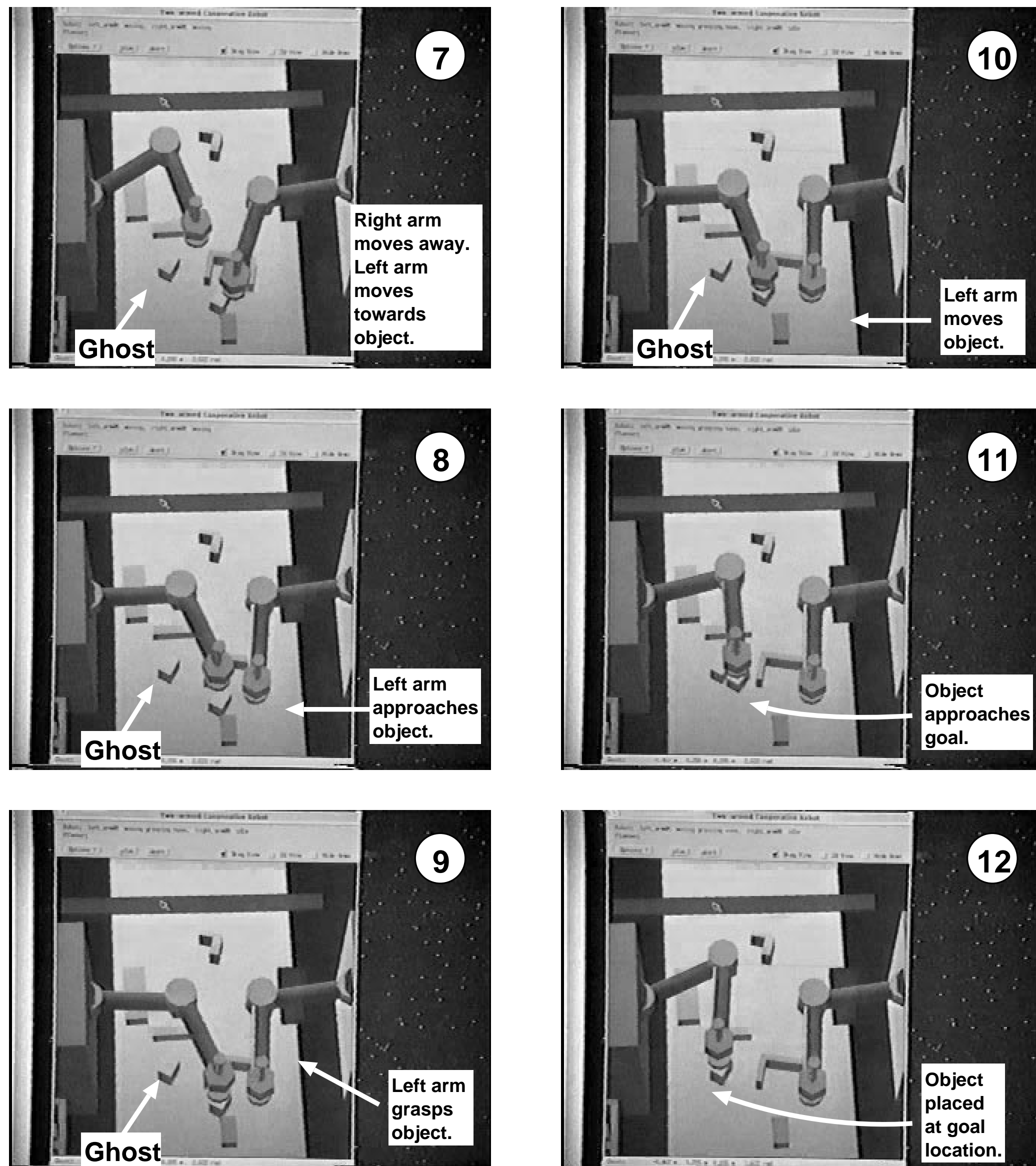


Figure 3.5: **Commanding and Monitoring the Workcell with the GUI (7-12)**

The hand-over operation (continued from Figure 3.4) completes in pictures 7 through 12.

The simulator has been used extensively to debug the planner subsystem. This not only provided safe development (simulated robot collisions are not quite as damaging as real ones), but also created a repeatable environment where identical situations could be easily recreated (this is hard in the real world because the "situation" involves not only the exact location of parts in the workcell but also the timing of the arrival of the different parts on the conveyor). The simulator graphical front-end

is very similar to that of the GUI, except the user is now allowed to modify the physical location of the objects, as well as change the location of the robot bases, conveyors, and arm configurations. These values may be saved and retrieved from configuration files so that situations can be easily reproduced.

The simulator subsystem was also extremely useful in the redesign of the workcell, as it allowed one to prototype manipulator location and kinematic parameters (link length), and to see the effects of these changes on the range of achievable tasks. This feature alone is one of the main functions of several graphical robot-simulation packages such as SILMA and DENEB—of course these systems are more general and have many other capabilities.

Lastly, the simulator has also been successfully used to explore the possibility of subsystem masquerading and live system reconfiguration. In this experiment, the robot-subsystem and simulator are both run concurrently. The design of the anonymous interfaces allows both to receive commands and export their world-state information. However, the interfaces (using the mechanisms provided by NDDS, described in Chapter 4) allow the world-state information from the robot-subsystem to take precedence while the robot is operational, so that the real system functions normally, and the operation of the simulator can be compared with that of the real system. The simulator can also be put in a mode in which it constantly resets its state with that received from the real system. If the real robot goes off-line (purposely or due to a failure or communications breakdown), the simulated data will replace the real data throughout the rest of the system (masquerading as the robot system). The benefits of this type of interaction have been demonstrated in telerobotic applications by the ROTEX experiment [64][7]. Our system could use a similar strategy to replicate the planning subsystem and allow multiple planners to compete with each other for the best plan.

3.5.3 The World Modeler

World modeling: The integration, interpretation, and representation of the sensory information using both á priori knowledge and other information exchanged by the different subsystems is a sophisticated process. The philosophy and implementation of this subsystem is presented in Chapter 5. The world modeler communicates with the rest of the system through the world-state interface.

[7] In the ROTEX experiment a graphical simulator was used to display the state of a robot in orbit and to interpolate its state between updates from the remote system. The state of the simulator was corrected every time an update from the remote robot was received. This approach allowed a user on earth to teleoperate a robot in orbit despite the long communication delay.

3.5.4 The Hierarchical Control System

This subsystem receives strategic-level commands and controls the workcell in response to these commands. The control of the workcell is quite sophisticated in that both arms must be managed and controlled in a variety of modes (independent trajectories for each arm, independent object motions, cooperative object motions, etc.). Furthermore, since the commands to the workcell are strategic-level commands, the control subsystem must be responsible for dividing them into elementary primitives, sequencing them and monitoring their progress. The control subsystem is explained in Chapter 6.

3.5.5 The Planning Subsystem

The planning subsystem is responsible for interpreting user task specifications, decomposing them into primitive subtasks (or strategic commands), and issuing them to the robot system. These plans include safe (collision-free) via-point paths for the arms and objects. The planning subsystem must also monitor the progress of the workcell so that the individual subtasks can be scheduled.

Planning is a notoriously hard problem, in this case exacerbated by the need to interact with a dynamic environment (objects approaching on the conveyor, planning for one arm while the other is already moving, etc.). The planning subsystem has been developed by Li from the Stanford Computer Science Robotics Laboratory as part of his PhD research [91, 92].

The interface to the planning subsystem is built by combining the world-state, system command, and task-specification interfaces. The characteristics of these primitive interfaces have several implications on the planner implementation:

- The planner uses the world-state interface to receive periodic updates of the state of the world (e.g. position of different objects, arm configuration) as well as notification of significant events (arms grasping an object, appearance of a new object in the workcell). This type of interface promotes the development of an event-driven planning subsystem.

- The planner commands the workcell using the system-command interface. This interface is stateless, so there is no explicit acknowledgment of individual commands. Therefore, the planner must use the world-state information as its only means to infer the status and progress of the system. As a consequence, the planner must also be quasi-stateless, using pre-computed plans only as hints on how to achieve the remaining tasks. I must always evaluate the next step based on the current (sensed) state of the workcell. This increases the planner complexity

but, in exchange, provides robustness to unexpected events (they simply represent a sudden change in workcell state). This approach also allows the planner to be (re)started at any time.

- There are unpredictable communication and execution delays on each command issued (some may be rejected unilaterally by the robot-subsystem). As a result, the planner groups individual plans into "transactions" that are delayed and executed as a unit. For example a coordinated two-arm motion cannot be sent as two individual single-arm commands, since the time synchronization among the two-arm motions would be lost, resulting in a potential collision.

These constraints have resulted in the restriction of the generated plans to those satisfying the constraint of *safety-under-time-delay* (SUTD). This constraint was introduced by the author in previous work [126]. It basically means that each individual plan transaction (strategic command) must be safe (assuming no unexpected events) under an arbitrary time delay. For instance, if an arm is already in motion, a plan that moves the other arm temporarily inside the region that will be swept by the already-moving arm can be safe, provided the motion of the second arm is coordinated in time so that the second arm leaves the swept region before the already-moving arm moves through that region. However, this plan will not be SUTD, because if the motion of the second arm is delayed, the arm may not leave the swept region in time to avoid a collision with the first arm. Figure 3.8 illustrates this concept.

The value of SUTD planning relies on the fact that these plans can be computed as easily as normal plans. The technique to compute SUTD modifies the configuration-space obstacles in configuration-time space ($\mathcal{CT}$-space) [87] by projecting future $\mathcal{CT}$-space obstacles back in time as illustrated in Figure 3.9. For any given time, this technique removes the region that will be swept by moving obstacles in their future motion from the present free space. Details of this technique can be found in Li's thesis [91]. As described in Section 3.4.2, the system-command interface incorporates these ideas by allowing every via-point path to be tagged with an "earliest-time" stamp (generated from the SUTD plan). So long as the controller ensures that the trajectory meets all the earliest-time constraints, the plan will be safe. The strategic control system will ensure that these constraints are met, or else the command will not be executed.

3.6 Experimental System Operation

This section illustrates the complete system in operation, and presents experimental data on the system accomplishing several tasks. Section 3.6.1 describes some typical operational scenarios, while animations and photographic sequences of the system in operation are presented in Section 3.6.2

3.6.1 Operational Description

Information interfaces together with the software bus allow combinations of modules to be easily used. Figure 3.10 illustrates some of these configurations. In the first three cases, we have replicated modules mentioned earlier. The top diagram indicates several users collaborating to control the robot or monitor its activities. The second diagram depicts the use of multiple simulators to simulate different aspects of the system, such as for command previewing. The third diagram shows the use of combined multiple planning strategies, each of which may be more appropriate for certain tasks. Finally, the last diagram indicates that modular interfaces allow us to build new interfaces such as one for a teleoperation subsystem.

In our operations, we have not experimented with multiple concurrently-active planners nor with teleoperation. Nothing in the architecture prevents us from doing so (since the remaining subsystems do not know where the data is coming from or going to), but the planners themselves require enhancement to be aware that they are not the only possible masters[8]. The actual arm trajectory is available to the planner through the world-state interface; however, the planner is not using this information. Similarly, a teleoperation subsystem would need to be developed (a significant endeavor by itself) and the system-command interface extended to allow specifying motion in ways more appropriate for teleoperation. For instance, we may want the reference state for an object to be directly tied to that of a hand-controller, or we may want to tie the reference velocities to a force-input device. Any of these mechanisms would require simple extensions to the system interface.

Figure 3.11 illustrates the command flow and resulting actions of the workcell during a typical task. This hypothetical task consists of placing two objects at some desired location. Actual experimental tasks are included in the following section. The task is surrogate to an assembly situation, except there is no contact between the objects. Although the system is able to manipulate objects in contact with their environment, neither fine-motion planning nor contact-specification

[8]For instance, assumptions currently made such as: "if you sent a command to move an arm, and the arm starts moving, then it must be moving along the specified trajectory," should be avoided.

commands have been addressed by the current research. These difficult and important issues are being addressed by other researchers in the ARL [152], (and elsewhere), and are promising areas for future research.

3.6.2 Experimental Tasks

The workcell can perform many operations semi-autonomously. These operations involve capture and delivery of objects in the workcell. The tasks specified by the user may require multiple steps due to the need to regrasp objects (to attain a more advantageous arm configuration by changing handedness), transfer objects from one arm to another (to deliver objects to areas accessible by only one arm), capture moving objects, etc. In this section we present experimental data for several representative tasks. The data set is presented in two formats: graphical animations and photographic sequences.

Graphical animations were generated by a drawing program using *experimental data* collected during system operation. The workcell is represented using the top view, and all objects are drawn to scale. The complete operation is divided into several segments, each corresponding to an individual system command from the planner, and each segment is animated by representing the position of the arms and objects at 0.8 sec intervals. Each segment is illustrated in two views (see Figure 3.12): The figures on the left include the robot arms while the ones on the right show only the objects. In this manner, pure arm motions are easily distinguished (no objects move) while the motion of the objects around each other can be observed unobstructed by the arms. These graphical animations contain a lot of operational information. They illustrate the system-command sequence and the location of every object in the workcell at small time increments. However they do not provide useful control-type information (tracking errors, reference velocities and accelerations, etc.). This data is presented in Chapter 6.

Photographic sequences have been generated by digitizing video taken during the operation of the workcell. Photographic sequences are used in combination with the corresponding graphical animations to illustrate certain operations as in Figures 3.13, 3.14. Detailed photographic sequences are also used to document complete tasks of the workcell as in Figure 3.24 thru 3.26. The pictures in the photographic sequences have not been taken at fixed time intervals (this would require far too many images). Instead, they have been chosen at a granularity such that the complete operation of the system can be inferred. In these pictures, paper cues have been placed on the table to represent

the user-requested goal location for the different parts. These cues are visual aids included only to enhance the illustrative nature of the photographs, and they are not used by the system.

To conserve space, most operations are illustrated as either graphical animation or photographic sequences, not both.

The workcell operates completely on-line without fixtures, á priori schedules, plans or assembly sequences. As a result, the same task can (and will) be executed differently depending on the exact timing of the arrival of the parts, and on CPU usage. This makes the workcell adaptive to the manufacturing environment, and opens interesting possibilities of part-driven workcell operation[9]. These facts have been illustrated by repeatedly commanding the same tasks, and observing several different strategies used by the workcell.

Operation of the workcell will be documented in an incremental fashion, beginning with simple tasks with static objects and concluding with more sophisticated tasks where objects are fed on a conveyor.

Maneuvers in a static workspace

A static workspace, one in which objects are not moving, is less challenging than a dynamic one because the timing of the operations need not be synchronized with anything external to the system. As a consequence, a plan taking longer to be generated, or a trajectory not being parameterized quickly results only in slower system operation, and no "deadlines" are missed. More importantly, there is no need for the planner to exchange timing information with the robot through the system-command interface, or indeed to consider the time domain at all. These operations are challenging nonetheless in that they are carried out interactively by the user, and it is important that the system appears responsive.

Figure 3.13 illustrates the result of the user requesting two objects to be placed at different locations. The first object can be immediately grasped and delivered by the right arm, but the second (square object) requires a "hand-over" operation. This occurs because the current location of the second object is reachable by only one arm, while the destination is reachable only by the other arm. The system automatically recognizes this situation and commands the right arm to grasp the object, place it at an intermediate location where it can be grasped by the left arm, and then commands the left arm to grasp and deliver the object to its intended destination. It is important to

[9]This may also make the manufacturing process less predictable. Compensating for these effects with the incorporation of learning and/or user-specified hints are all areas of promising research.

note that these decisions and coordinated maneuvers occur without user intervention. Figures 3.14 contains photographic images of the system during each one of the six stages.

Figures 3.15, 3.16 illustrate a more complicated scenario which may arise during assembly operations. The user has specified the "assembled" configuration consisting of the relative locations of the three objects shown at the end of Figure 3.19. Achieving this "assembly" involves moving all three objects[10]. The order in which these operations occur is important (some orderings may be impossible). The user does not need to be aware of any of these details and simply uses the GUI to instruct the system of the desired final configuration and to monitor system progress. Images from the actual assembly can be seen in Figures 3.17, 3.18, and 3.19. These images were taken by digitizing video recorded during the assembly operation.

[10]Note that the large object requires cooperative dual-arm manipulation.

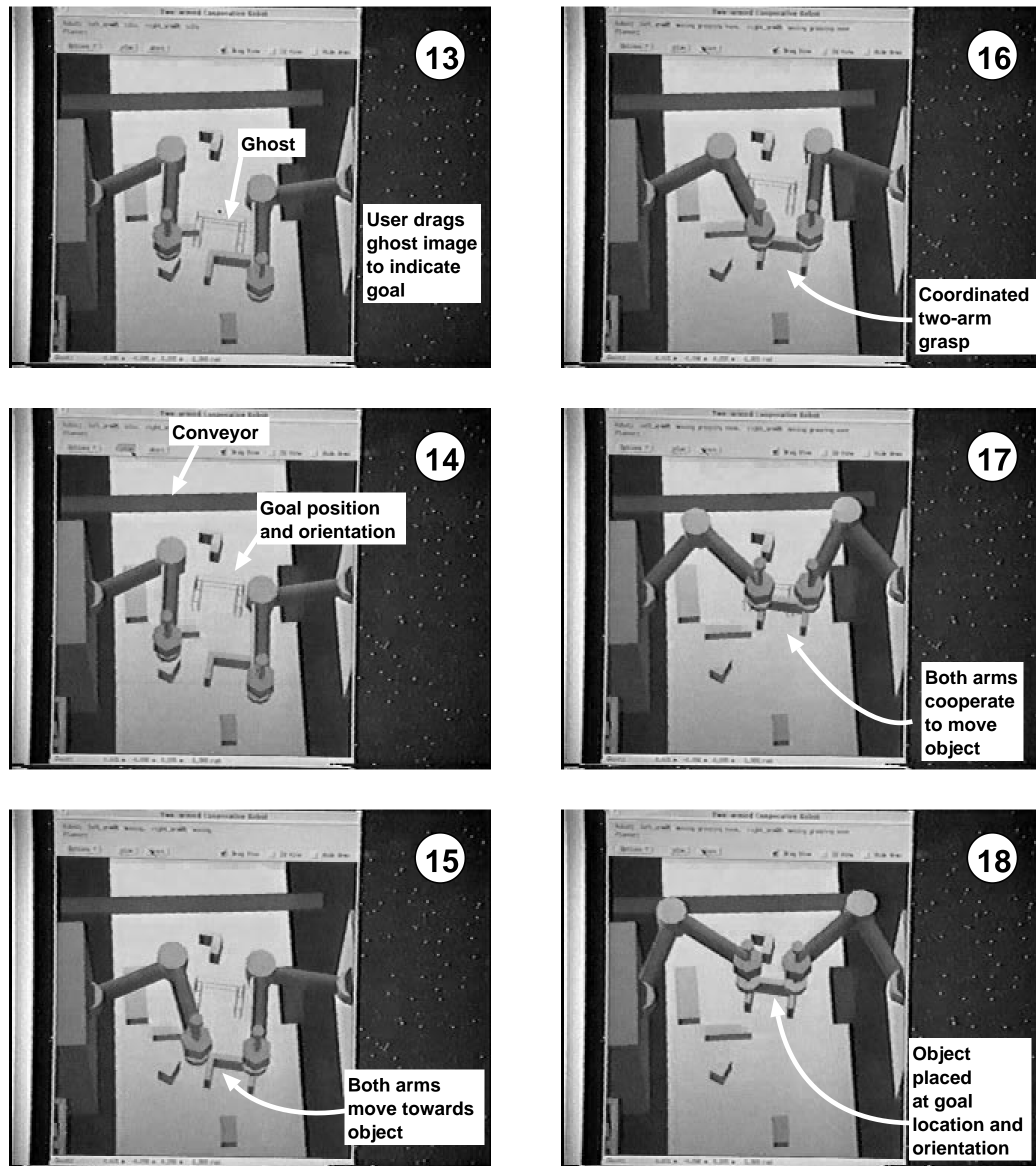


Figure 3.6: **Commanding and Monitoring the Workcell with the GUI (13-18).**

Continuation from Figures 3.4 and 3.6. In this sequence, the user commands to move the big "U-shaped" up towards the conveyor. The user selects the object and drags its ghost image back to a position and orientation near the conveyor (13,14). From there on the system proceeds automatically by first acquiring the object with both arms (15,16) and then delivering it to the requested position and orientation (17,18).

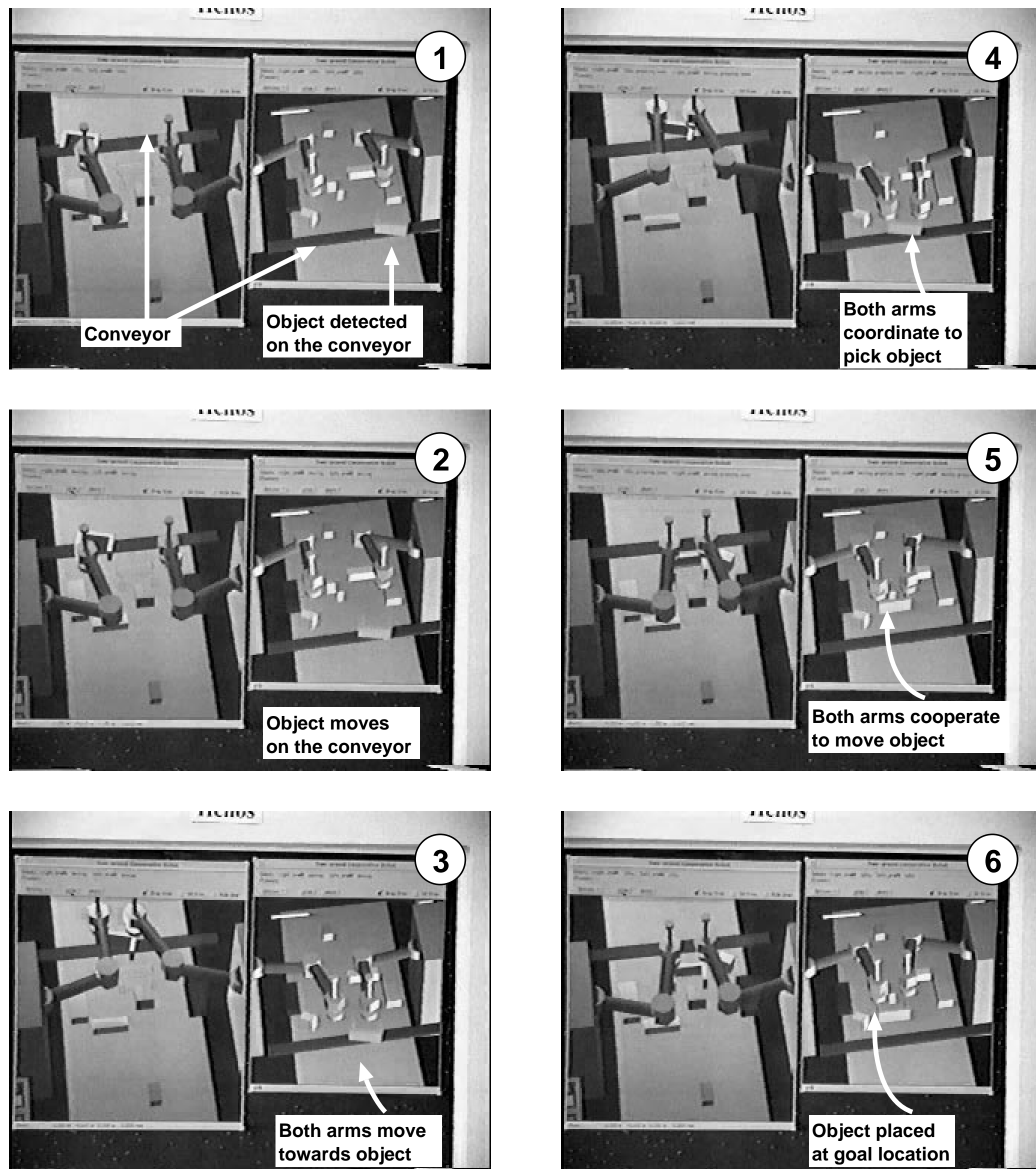


Figure 3.7: **Commanding and Monitoring the Workcell with the several User Interfaces**

This sequence illustrates the use of multiple GUIs to view the same live operation from different perspectives. The two GUIs are running on different computers (they are displayed on the same screen for documentation purposes only). Multiple GUIs could be used to monitor/command the workcell simultaneously from different locations. The GUIs need not be identical, and can be customized for each user.

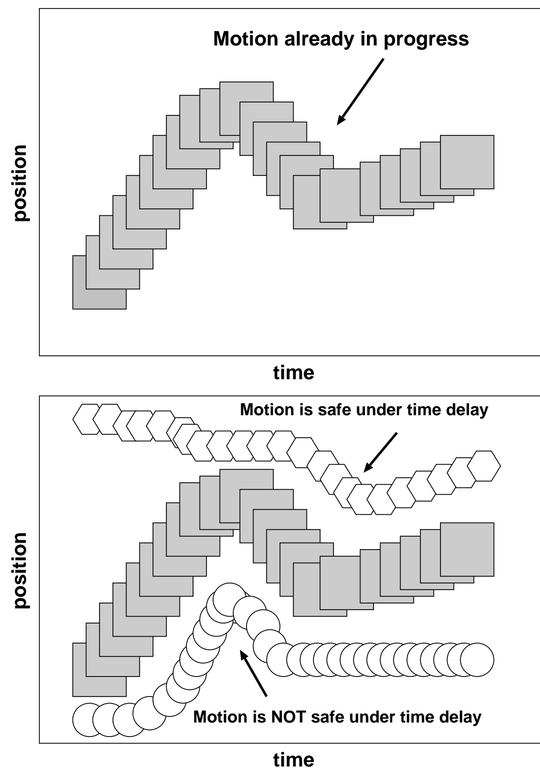


Figure 3.8: **Concept of Safe-Under-Time-Delay (SUTD) planning**

These figures illustrate the concept of SUTD planning. The top figure animates the future motion of an object which is already moving. The bottom figure shows collision-free plans for two objects. The plan for the circular object is not SUTD because if the motion was slightly delayed it would collide with the square object. On the other hand, the plan for the hexagonal object is SUTD because it can be arbitrarily delayed, and the resulting motion will still be collision-free.

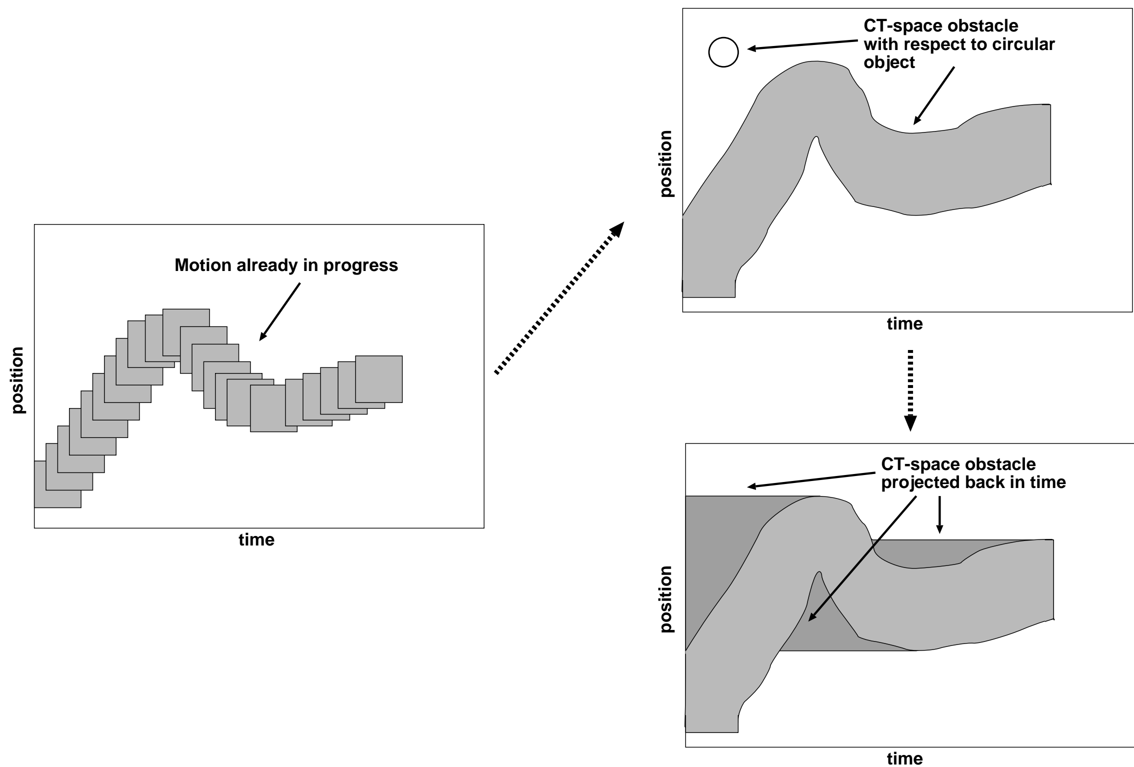


Figure 3.9: **Use of configuration-time space for safe-under-time-delay (SUTD) planning**

Normal configuration-time space ($\mathcal{CT}$-space) planning can be used to generate SUTD plans. First the normal $\mathcal{CT}$-obstacle corresponding to the motions in progress is generated (top figure). The $\mathcal{CT}$-obstacle is then projected backwards in time (bottom figure). Normal planning in $\mathcal{CT}$-space using this modified obstacle yields SUTD plans.

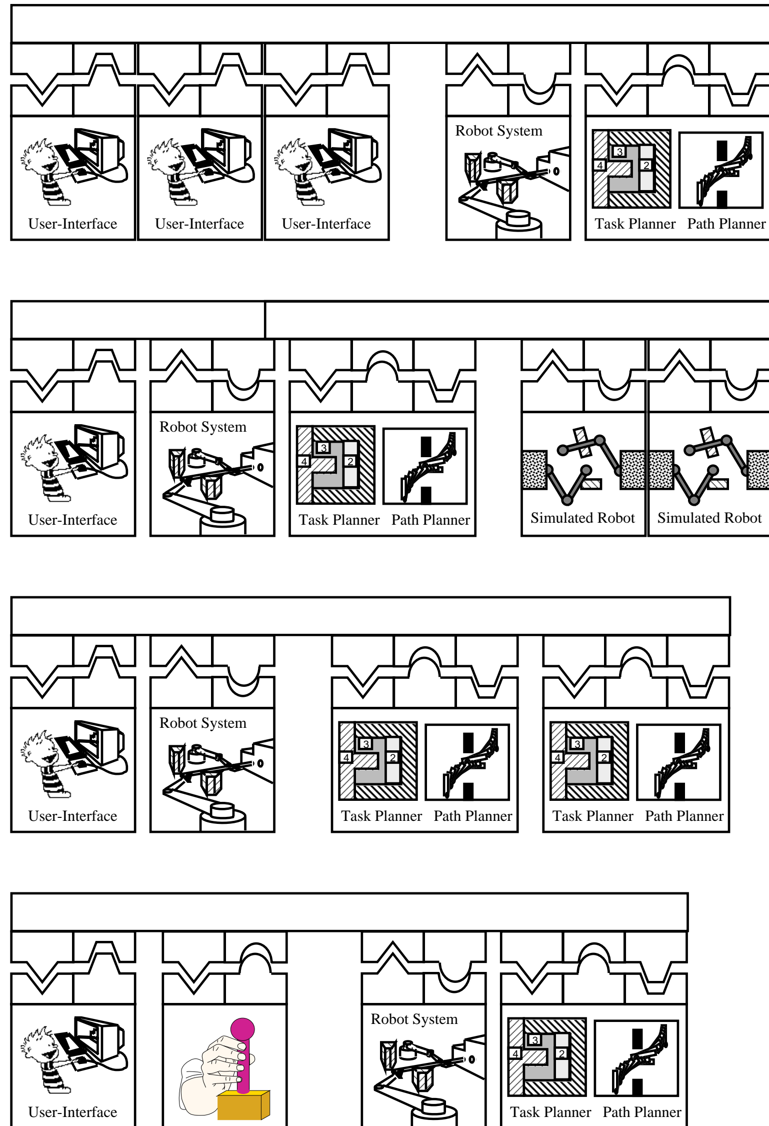


Figure 3.10: **System Configurations**

The anonymous interfaces allows replicated modules (top three diagrams) in the system. The modular information interfaces allow new subsystems to be added to the system (last line).

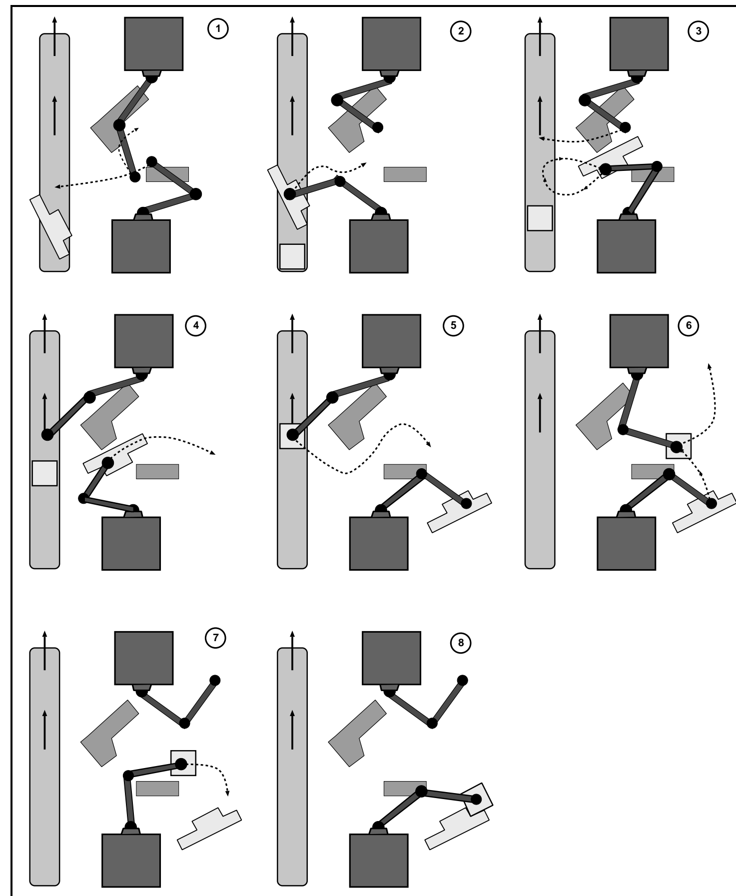


Figure 3.11: **Typical system operation**

The user has specified the simple assembly of two objects depicted in the bottom-right (8) by dragging the iconic representations of the objects on a graphical interface (not shown). After this the remaining actions are autonomous (left to right, and top to bottom): (1) The planner constantly monitors the workcell using the world-state interface, and as soon as a needed object appears on the conveyor, a capture trajectory is planned and sent to the robot [`move and grasp`* command]. This command specifies the top arm to be moved out of the way, and the bottom arm to grasp the object. (2) Once the object is grasped, the planner (which has detected the event through the world-state interface) plans a delivery trajectory and sends it to the controller [*`move and release`* command]. Since the arm cannot deliver the object to its goal location with its current handedness, the object is placed at an intermediate location where the arm can change handedness and regrasp it. (3) In the meantime, a new object has appeared on the conveyor, so the planner issues a* `move and grasp` *command for both arms (one for the conveyor object, the other for the just-released object). (4) While one arm picks the second object from the conveyor, the other delivers the first object to its final destination [*`move and release`* command]. (5) Once the second object is grasped, since the final destination is only reachable by the other arm, the arm grasping the object is commanded to place it at a location reachable by both arms [*`move and release`*]. (6) Next a* `move and grasp` *command moves the first arm away while the second picks the object, and finally, (7) a* `move and release` *command delivers the second object to its final destination (8).*

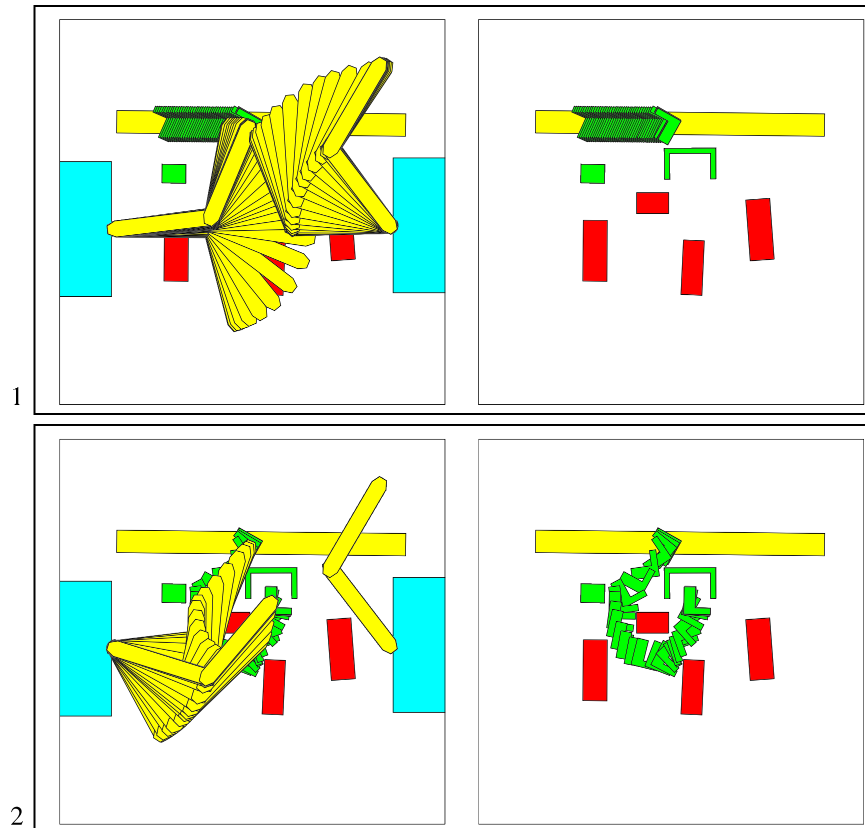


Figure 3.12: **Animation of capture and delivery**

Animation taken at 0.8 sec intervals of **experimental data** *collected during the capture and delivery of an object. Each one of the two figures corresponds to the response to a single strategic command from the planner:* (`move-and-grasp`) *for the top figure and* (`move-and-release`) *for the bottom figure. Side-by-side figures represent the same time sequence, except in the right figures the arms are not shown, so that the object motions can be seen in detail. The complete operation starting at the time the object appears on the conveyor took 20 seconds.*

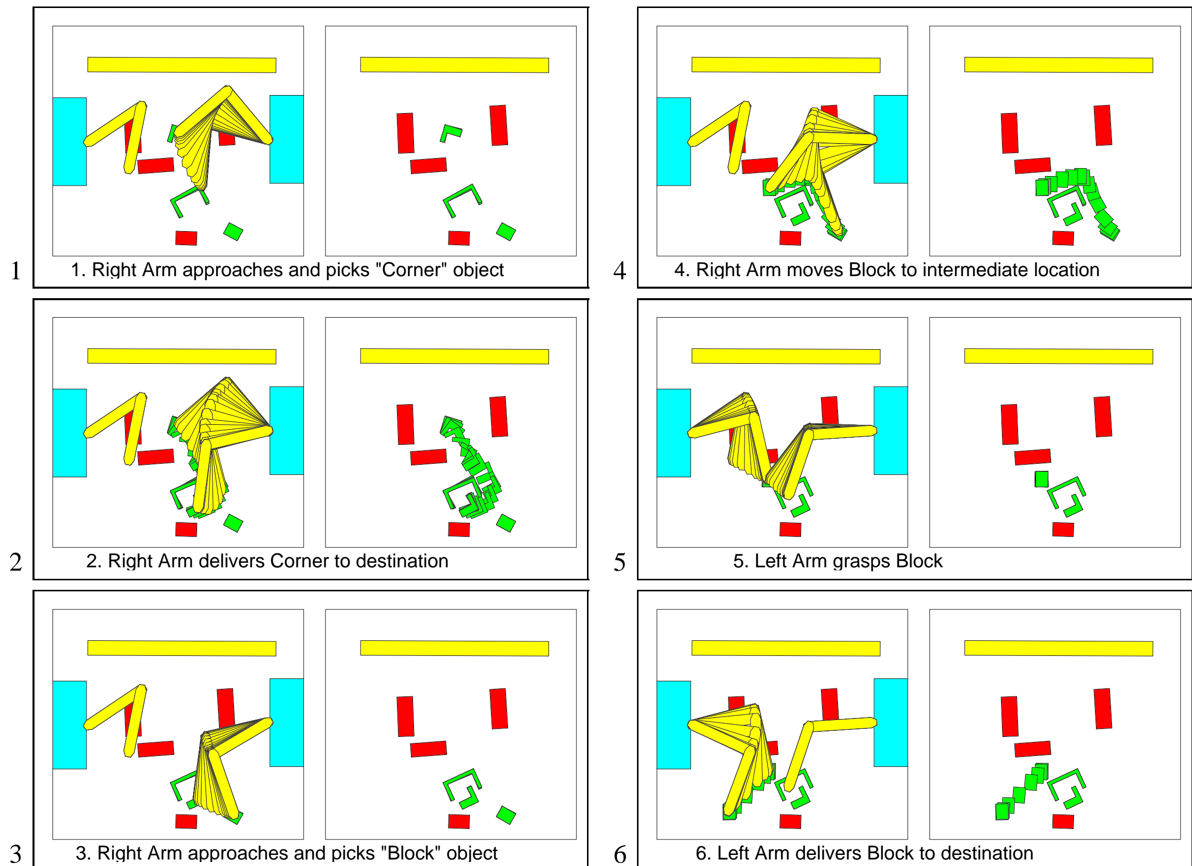


Figure 3.13: **Animation of hand-over operation**

The workcell has been commanded to move the corner-shaped object inside the "U" shaped one, and the small square object (bottom-right on first figure) to the other side of the workspace. Since all the required objects are in the workspace, the system can proceed immediately. The total delay from user specification to the beginning of the motion of the robot was 0.7 sec (this includes communication from the GUI to the planner, planning time, communication from the planner to the robot, command processing and trajectory generation). Six system-commands are needed for this operation (one per figure above).

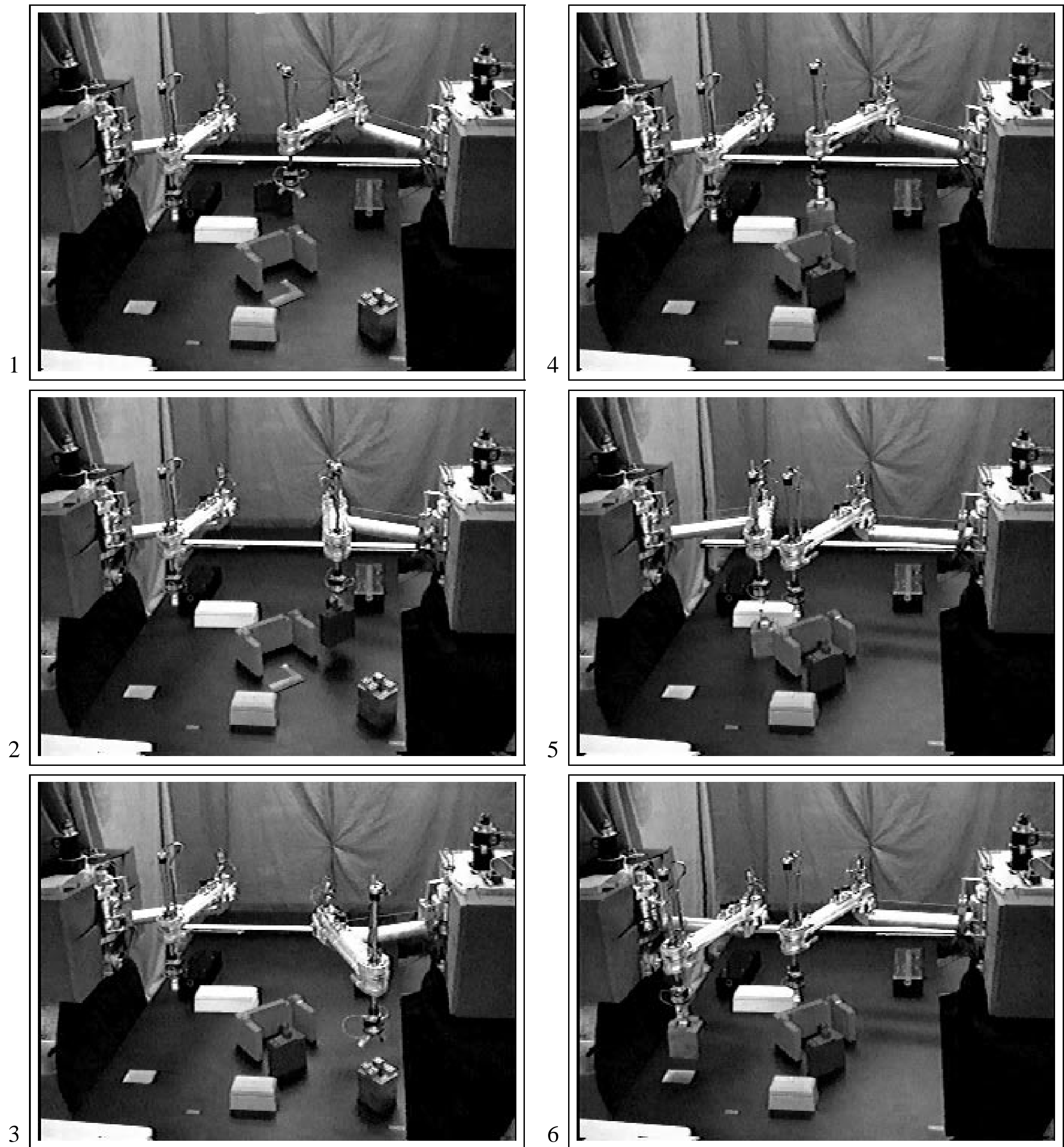


Figure 3.14: **Photographs during hand-over operation**

Digitized video images from the sequence illustrated in Figure 3.13

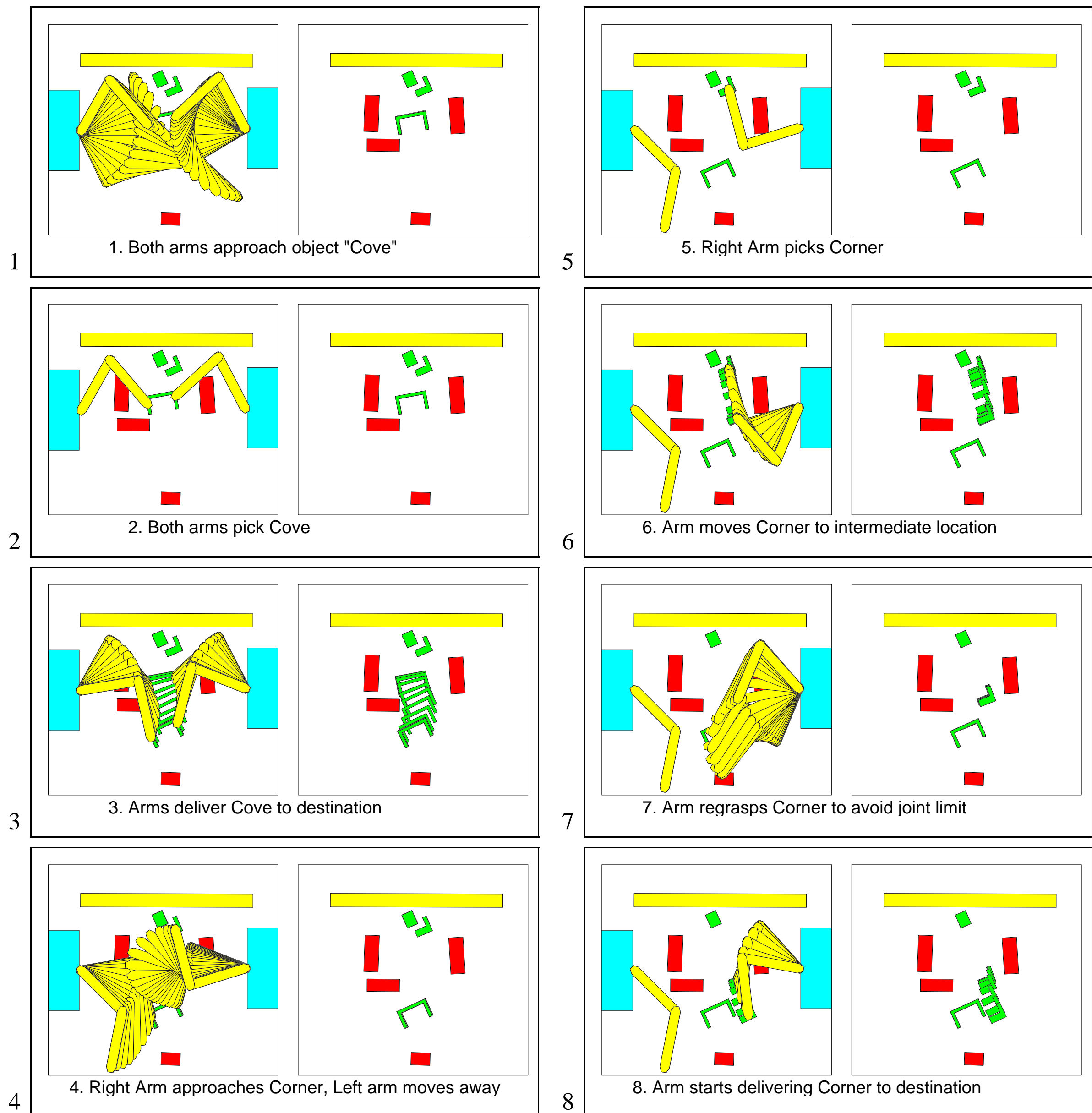


Figure 3.15: **Animation of multi-object-placing sequence with static objects (stages 1-8)**

This sequence automatically moves three objects to their new specified locations. This sequence continues in Figure 3.16. Representative pictures of the workcell during the operation are presented in Figures 3.17, 3.18, and 3.19.

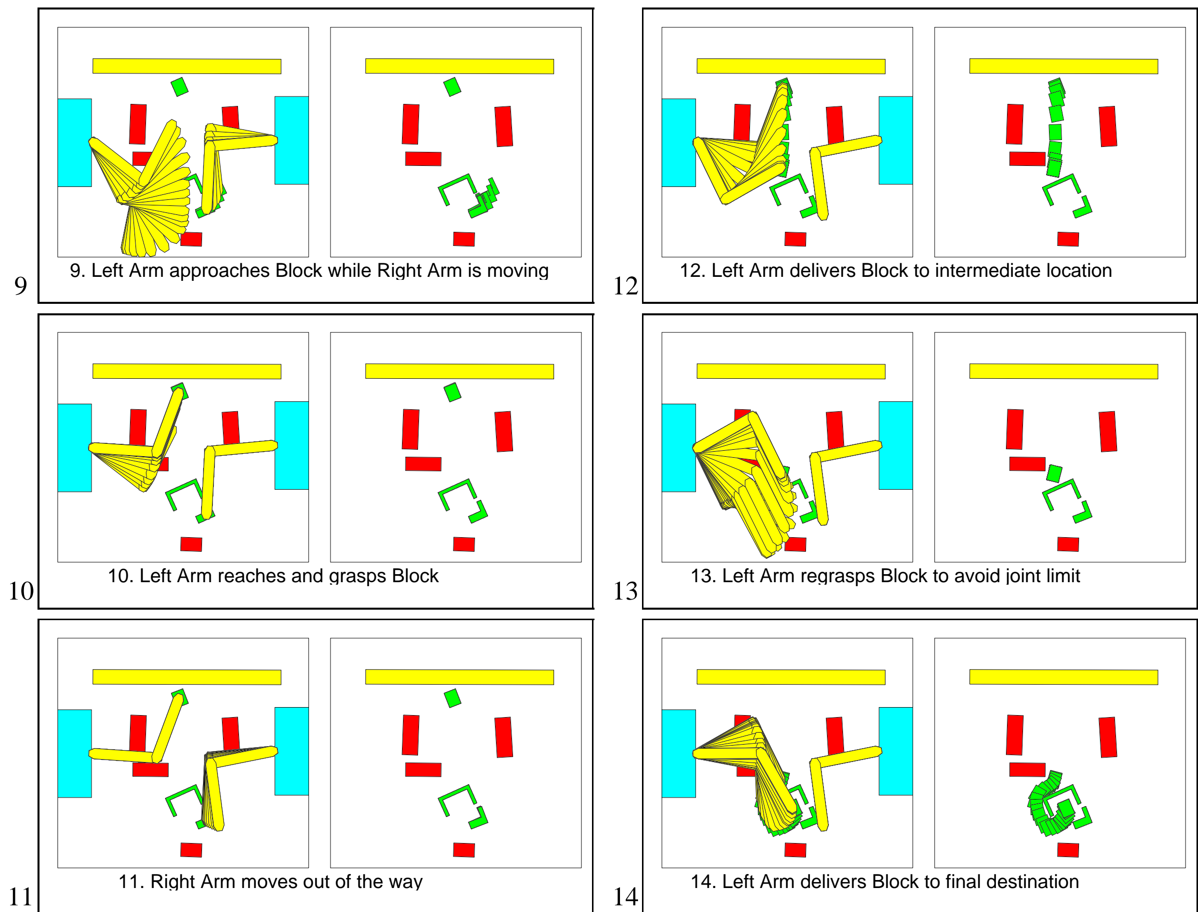


Figure 3.16: **Animation of multi-object-placing sequence with static objects (stages 9-14)**

Conclusion of the sequence started in Figure 3.15.

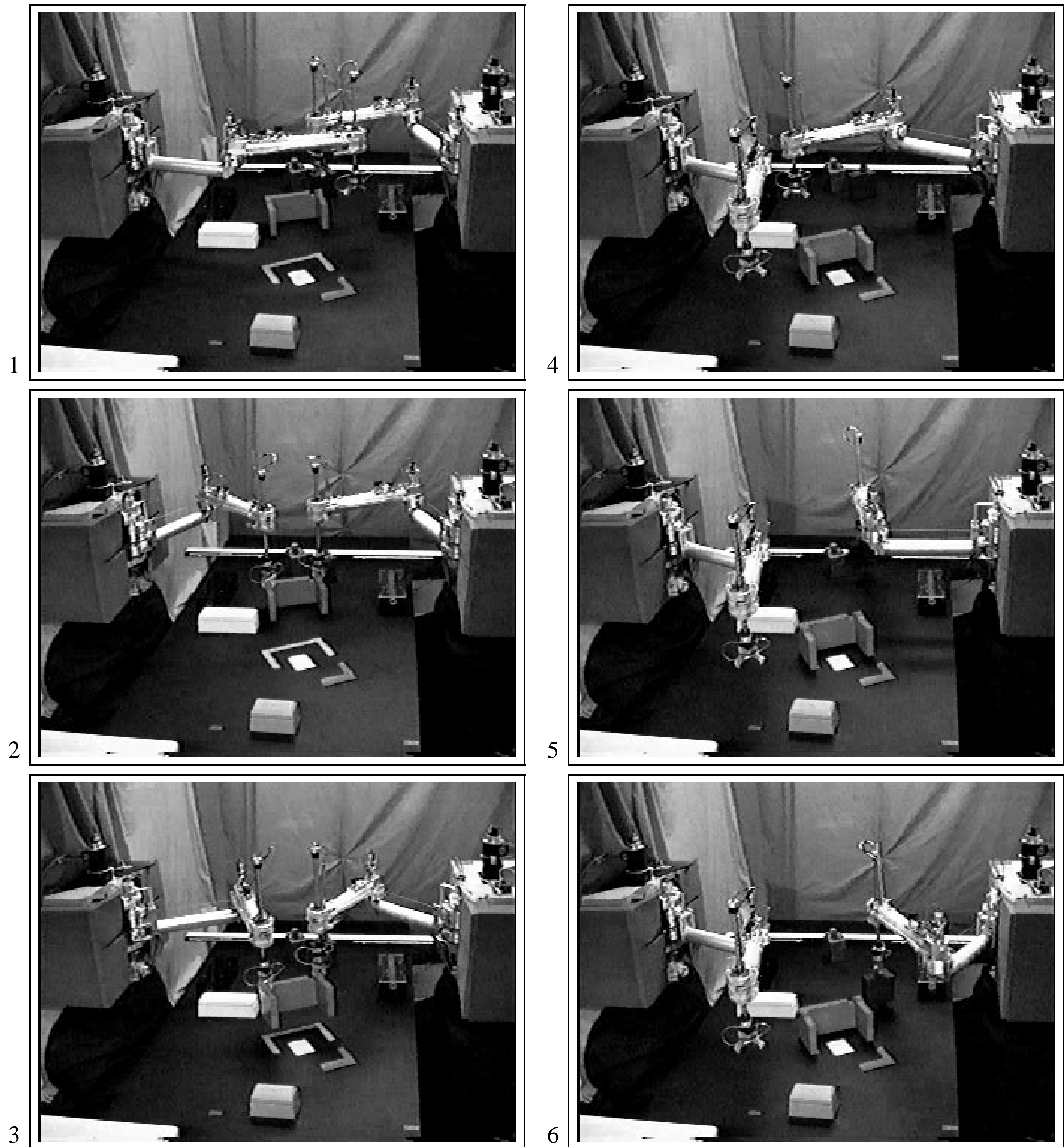


Figure 3.17: **Photographs during multi-object-placing sequence with static objects (stages 1-6)**

Snapshots taken during the same operation animated in Figures 3.15. Each picture contains the workcell at some point during the corresponding stage. The above six pictures are for the first six stages.

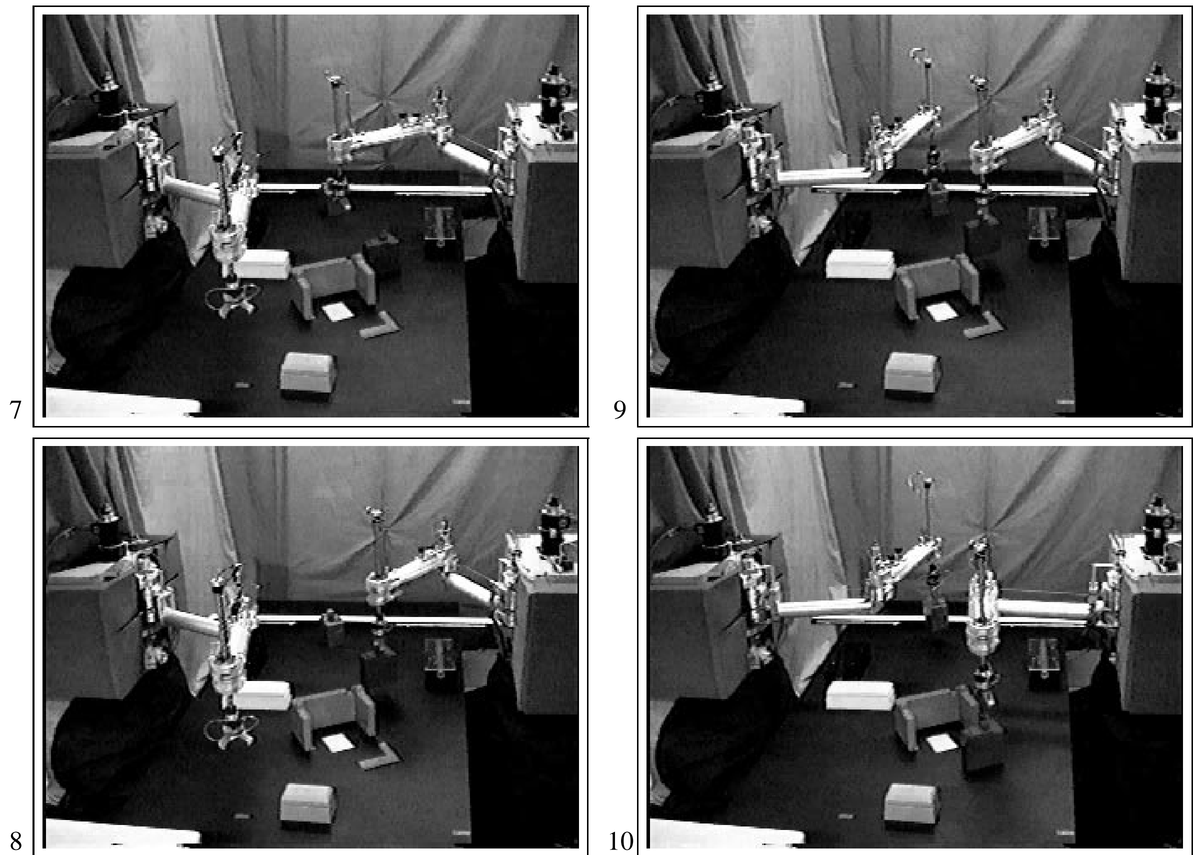


Figure 3.18: **Photographs during static assembly sequence (stages 7-10)**

Snapshots taken during the same operation animated in Figures 3.15, Each picture contains the workcell at some point during the corresponding stage. The above pictures are for stages 7 through 10.

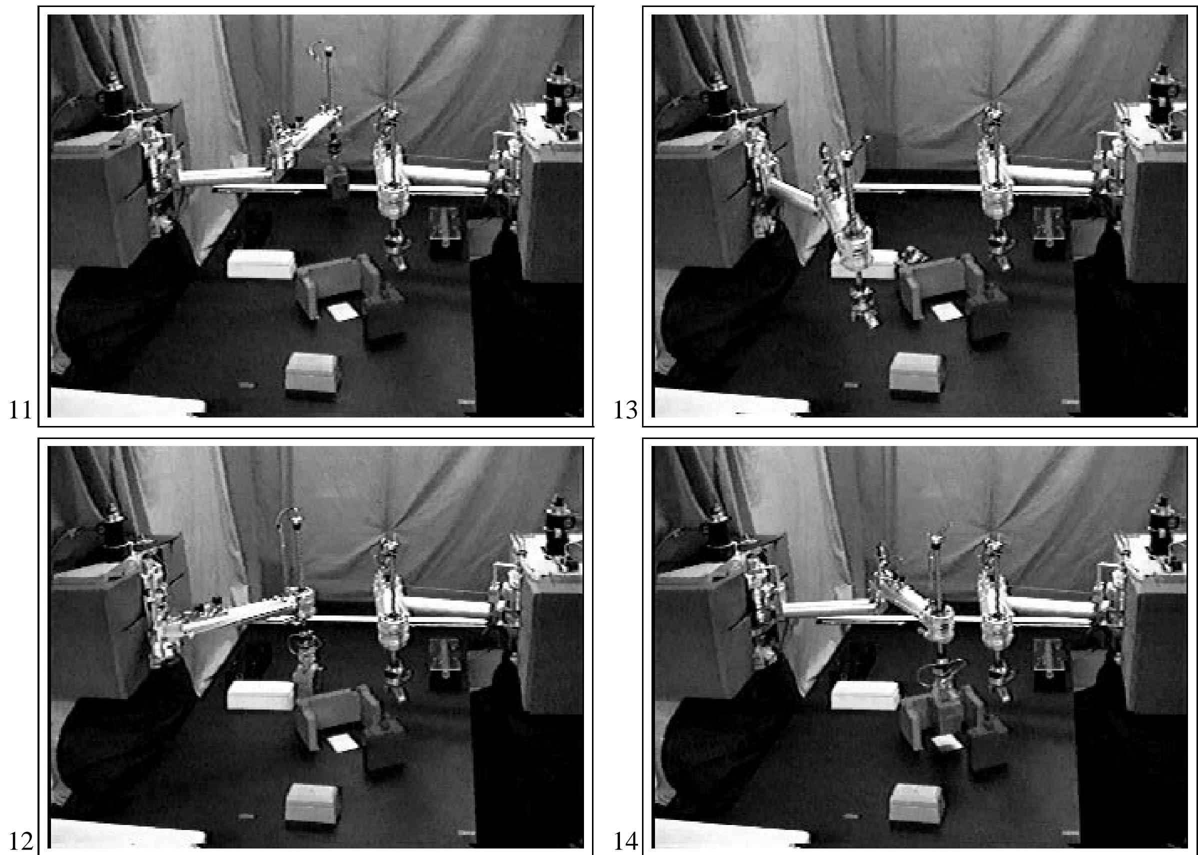


Figure 3.19: **Photographs during static assembly sequence (stages 11-14)**

Snapshots taken during the same operation animated in Figures 3.15, Each picture contains the workcell at some point during the corresponding stage. The above pictures are for stages 11 through 14.

Maneuvers in a dynamic workspace

The objects required for the assembly need not be present in the workcell at the time of task specification. The system remembers the task and will acquire the objects as they appear on the conveyor (in whatever order). Dynamic scenarios where the workcell interacts with moving objects are much more challenging than the static ones described in the previous section. In a dynamic environment, there are real-time constraints on the duration of certain operations. Time needs to be taken into account both by the planner and the strategic control system. The planner must incorporate the time "dimension" and certain "goals" become curves in configuration-time space (the "goal" of intercepting an object corresponds to a position that varies as a function of time). This results in an increase in problem dimensionality. Additionally, the planner must be aware of (and account for) the time consumed in its own planning process. In the context of this project, these issues have been successfully addressed by Li's PhD research [91]. The complexity of the dynamic environment not only affects the the planning subsystem, the interfaces themselves must provide the means for expressing the time constraints (section 3.4.2), and the trajectory generation must also trade off compute time versus efficiency of the generated trajectory (see Chapter 7).

Figure 3.20 animates data collected during a multi-object-placing operation (again an assembly-type scenario where objects are brought within close proximity but no contact is made). In this case, the required parts are not present in the workcell and will be delivered by the conveyor at some later time. As soon as any of the required parts appear on the conveyor, the system generates on-line plans to acquire and deliver the parts to their intended destination without further user intervention. Snapshot images during this operation are presented in Figure 3.21.

The system can be reconfigured without any programming. As an example of this, the location of the assembly and conveyor has been completely changed during system operation, from the configuration in Figure 3.21. Since the planner and remaining subsystems have subscribed to the relevant information, they are notified immediately of this reconfiguration without having to explicitly query for any configuration changes. Again, the required parts are not initially present in the workspace and are delivered by the relocated conveyor. The successful operation of the system under these new circumstances is illustrated in Figures 3.22 and 3.23.

As previously explained, because all planning and trajectory-generation occurs on-line, any changes in the order or timing of part arrival may cause the system to operate differently. For example, the detailed photographic sequence presented in Figures 3.24, 3.25 and 3.26 is taken during a different execution of the same user task-command (i.e. the placement of the objects is the same). However, the timing is different. In particular, the last part is fed on the conveyor shortly

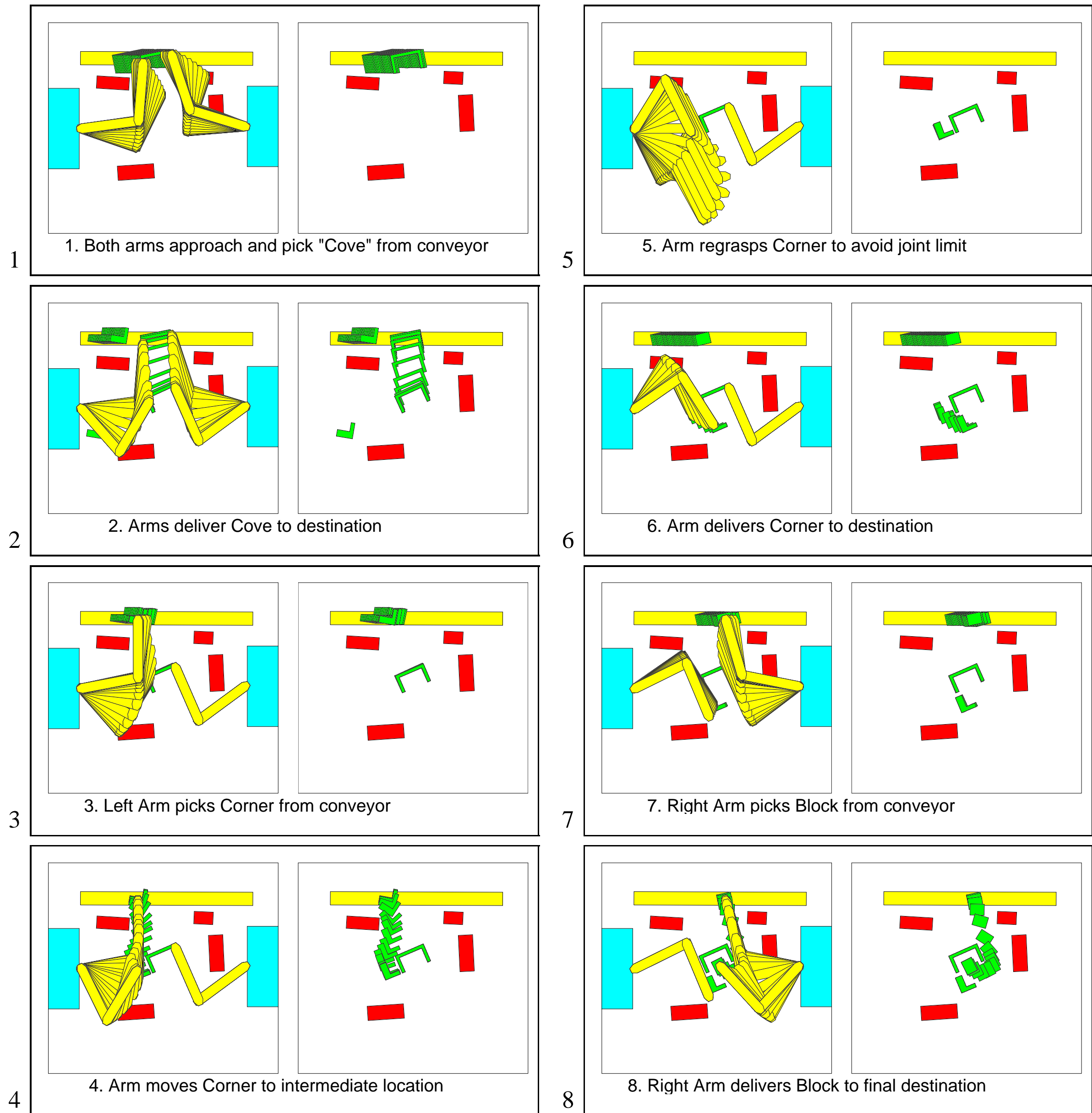


Figure 3.20: **Animation of multi-object-placing sequence with objects delivered on the conveyor**

This sequence animates data collected during system operation. The user has specified the placement of three parts in the workcell. The parts are not present in the workcell initially, but as soon as they arrive on the conveyor, the workcell automatically retrieves them from the conveyor and delivers them to their destination. These animations are taken at 0.8 sec. intervals, the complete sequence operation takes 40 seconds from the time the first object is detected on the conveyor. Representative pictures of the workcell during the operation are presented in Figure 3.21.

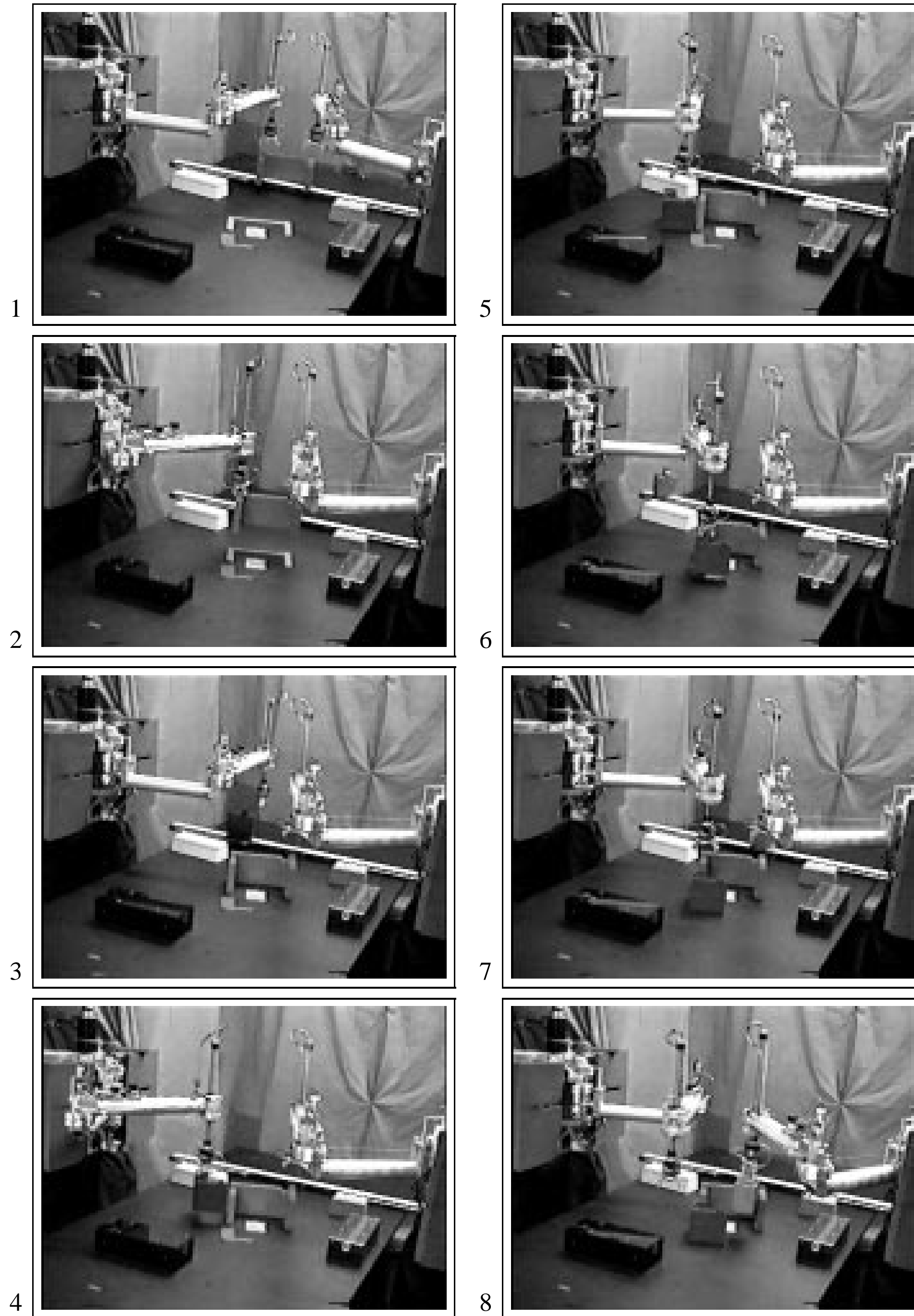


Figure 3.21: **Photographs during multi-object-placing sequence with with objects delivered on conveyor**

Snapshots taken during the same operation animated in Figure 3.20. Each picture shows the workcell at some point during the corresponding stages.

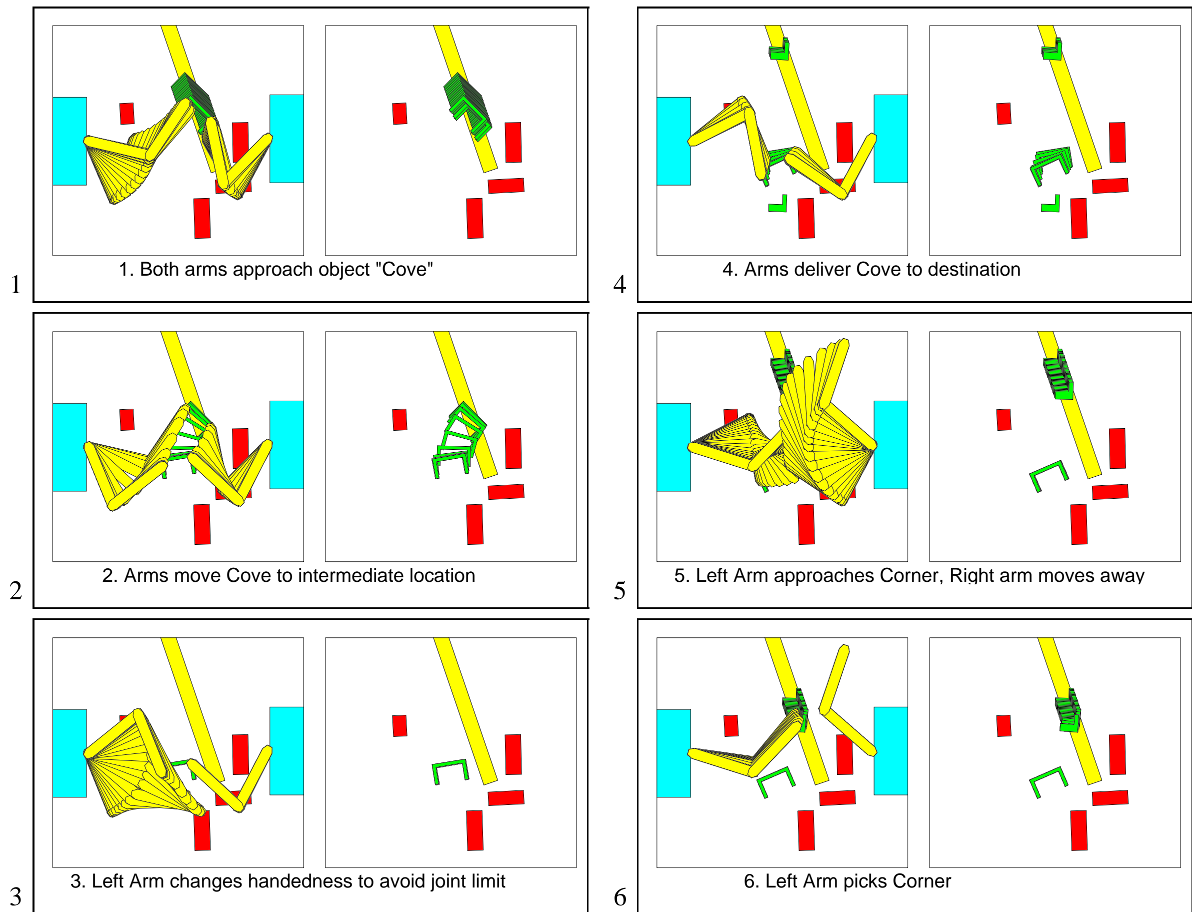


Figure 3.22: **Animation of multi-object-placing sequence with different conveyor position (stages 1-6)**

Sequence animating data collected during system operation. The user has specified a task involving the placement of three objects that are not currently present in the workspace. As soon as the first object is detected, the planner issues a command to pick the object (top-left figure). Note that the system knows that the object needs two arms to be manipulated. Due to joint limits in the left arm, a regrasp is required before it is delivered to its destination (stages 2,3,4). Stages 5,6 animate the capture of the second object. This sequence continues in Figure 3.23.

after the second one is fed, forcing the workcell to acquire it while it is still manipulating the second part. Otherwise the system risks being too late to grasp the last part. This sequence also illustrates the system ability to use both arms concurrently to increase throughput. This capability is enabled by proper design of both the planning and hierarchical control subsystems.

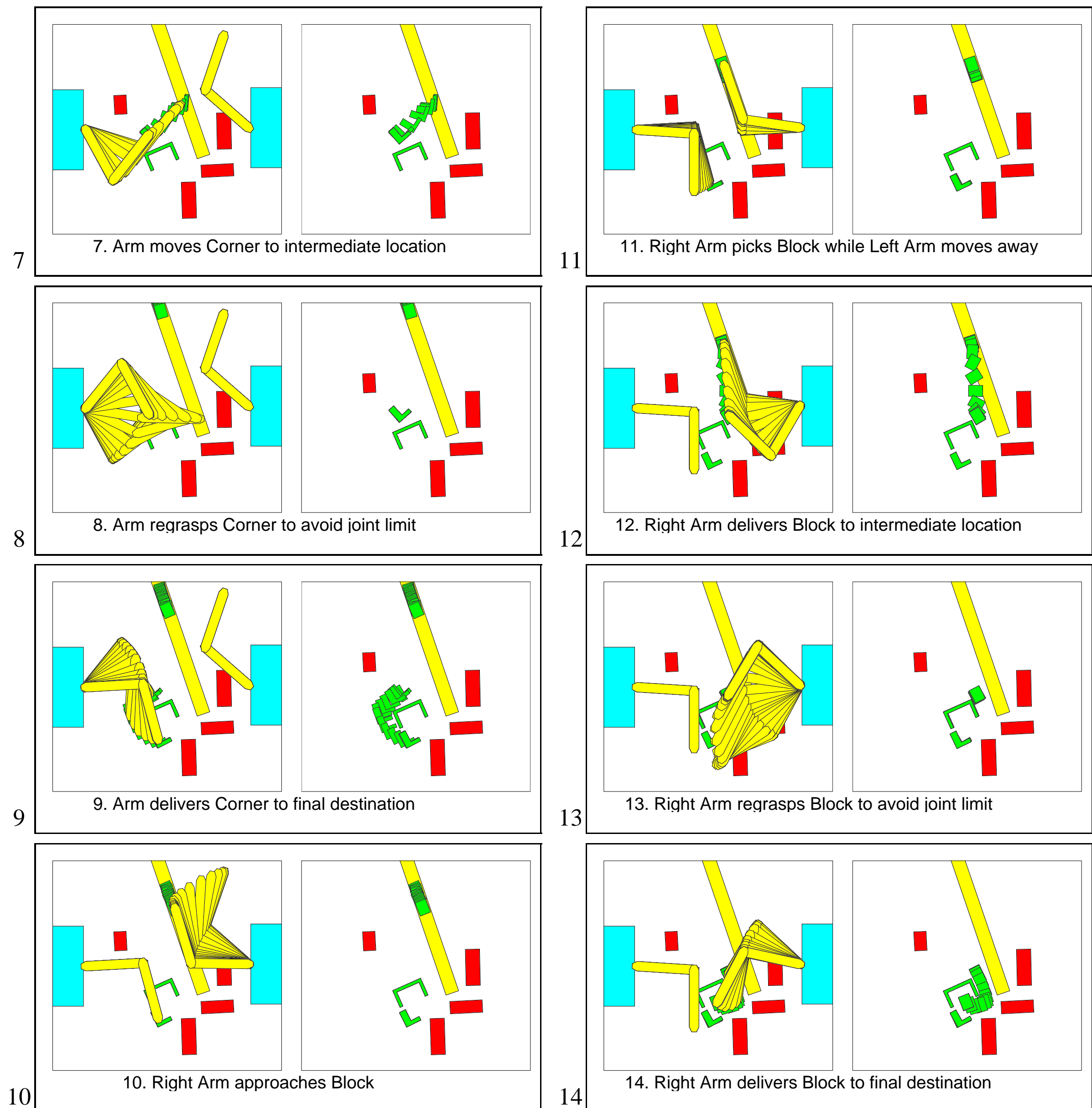


Figure 3.23: **Animation of assembly sequence with conveyor across the workspace (stages 7-14)**

Continuation of Figure 3.22. Delivery of the second object also requires a regrasp due to joint limits (top images: stages 7, 8 and 9). Finally, stages 10 though 14 animate the capture and delivery of the third object (which also requires a regrasp).

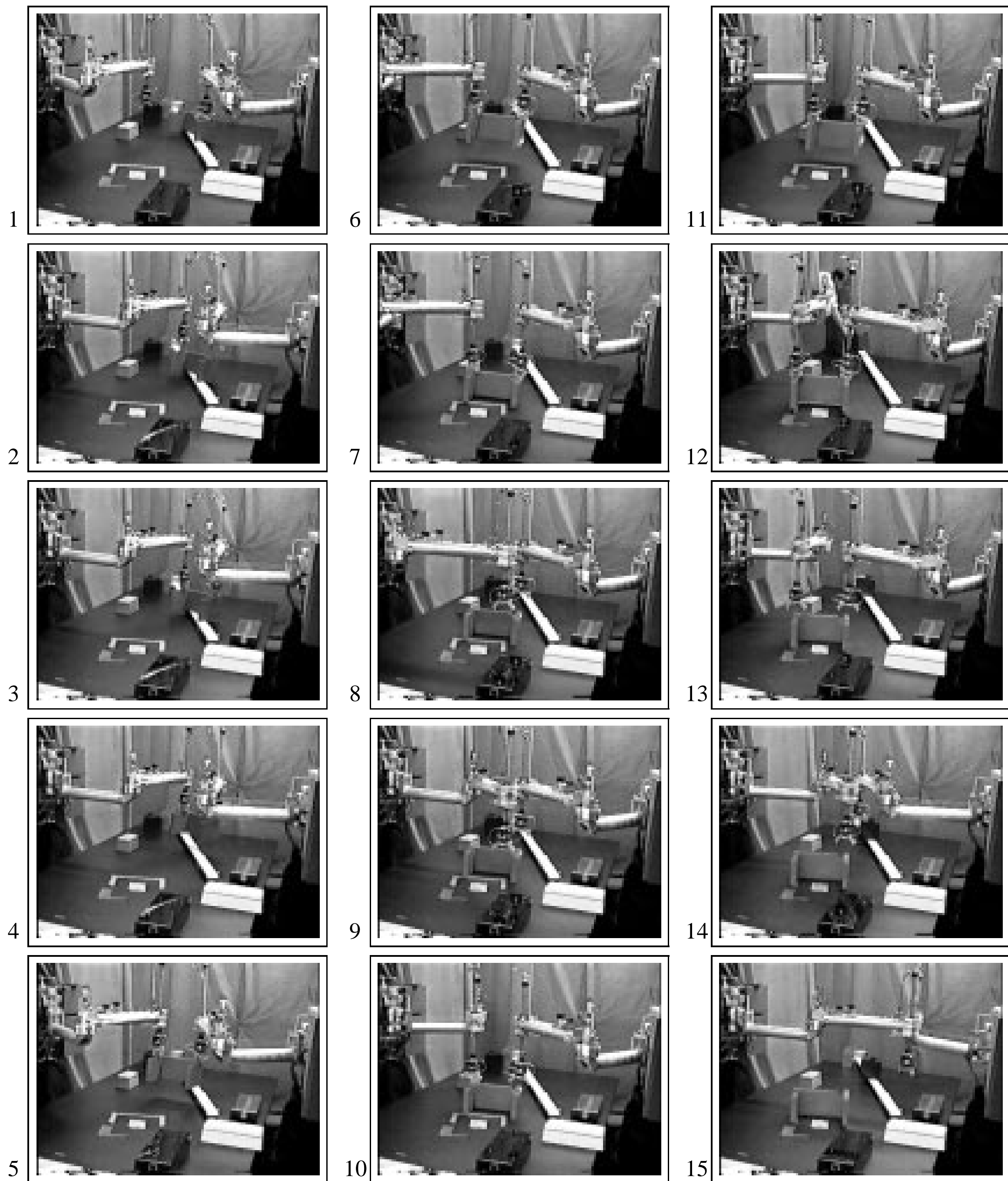


Figure 3.24: **Detailed strobe images of an multi-part-placement operation (1-15)**

This sequence documents the operation of the workcell that has been commanded the same multi-object-placing task illustrated in Figures 3.22 and 3.23, however the pictures **do not correspond** *to the same run. Since the timing of part arrival is different, the actual trajectories and command sequences are different. This sequence continues in Figures 3.25 and 3.26.*

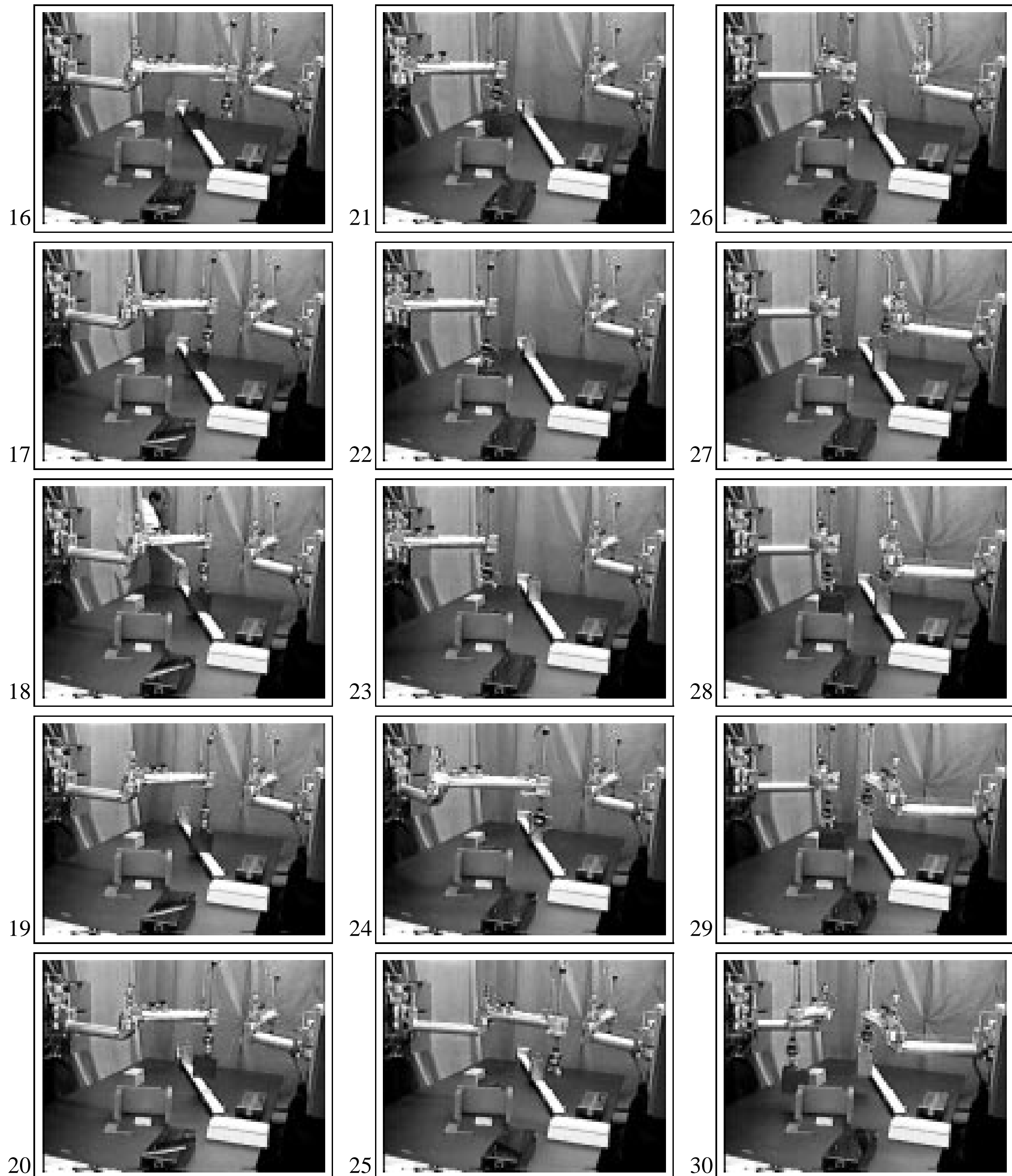


Figure 3.25: **Detailed strobe images of an multi-part-placement operation (16-30)**

Continuation of sequence started in Figure 3.24. Notice how the workcell is capable of moving both arms independently to pick the third object from the conveyor while it regrasps the second object (last two rows of figures). This sequence continues in Figure 3.26.

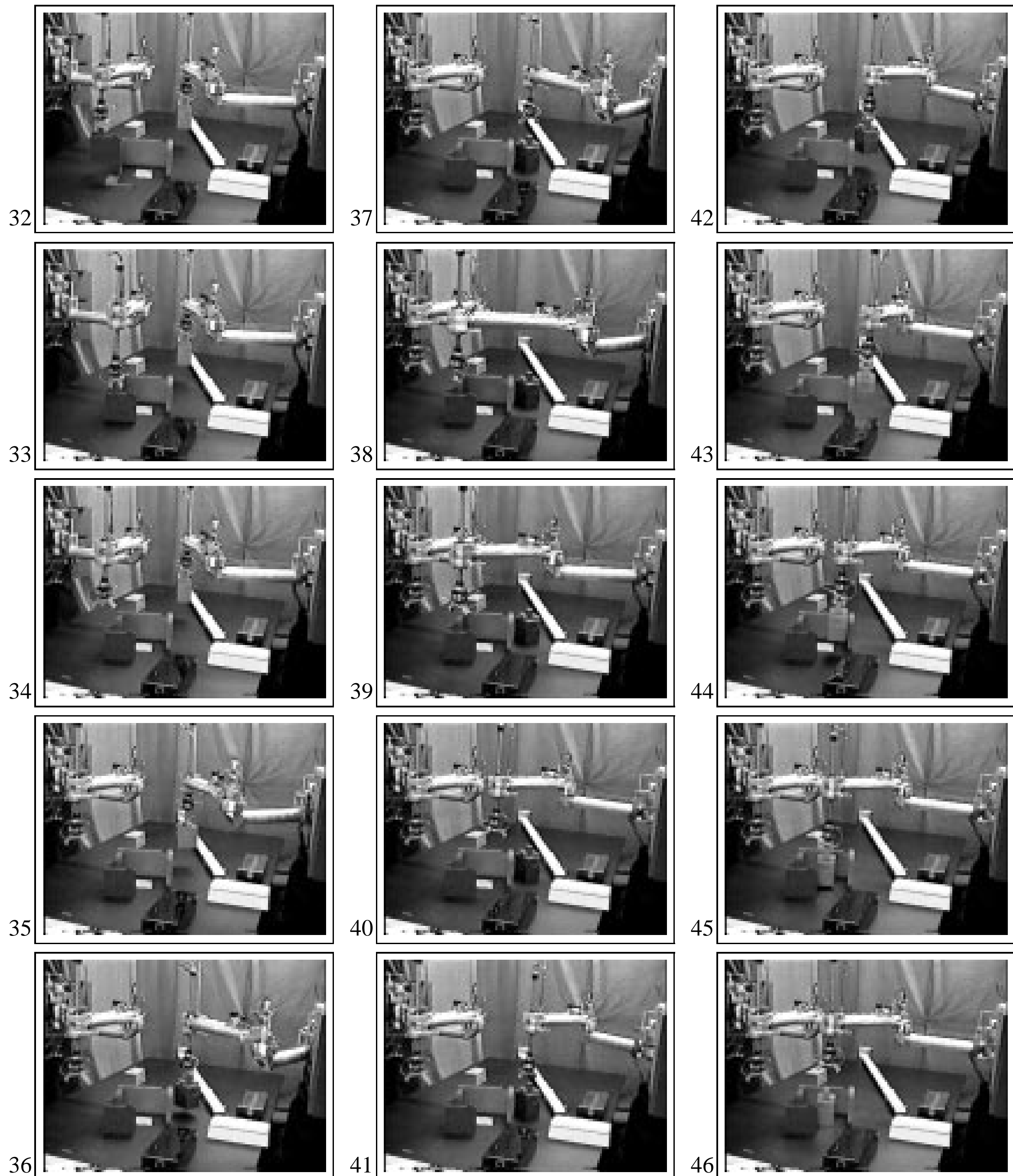


Figure 3.26: **Detailed strobe images of an multi-part-placement operation (31-45**

Conclusion of the sequence started in Figure 3.24.

3.7 Summary and Conclusions

This chapter has described the operation of the complete system as well as the architecture and interfaces that enabled its functionality. System operation has been described using graphical animations of experimental data collected from the system, as well as still images obtained from video sequences. The experimental results document complex, multi-step autonomous operations involving part-acquisitions from a moving conveyor, part regrasping, hand-overs, and dual-arm manipulation.

A novel approach to complex robotic system design called *interfaces-first* design was also a major contribution made in this chapter. Interfaces-first design starts by identifying the fundamental types of *information* flow in the system, and then encapsulates that flow into primitive anonymous interfaces. Subsystem interfaces are then built from combinations of these primitive information interfaces. This approach results in expandable systems with facile interconnectivity. The characteristics and information content of the specific interfaces designed for the dual-arm workcell are described in detail along with the operational examples.

Using these information-interfaces, the architecture of the workcell is built around five subsystems: (1) Graphical User Interface, (2) Planner, (3) Hierarchical Control System, (4) World Modeler, and (5) Simulator. Brief descriptions of the user interface and simulator subsystems have been presented in this chapter. The remaining subsystems are described in Chapters 6, 5, and in Li's thesis [91].

The design methodology and architectural concepts presented in this chapter are not limited to the specific architecture of the workcell. Rather, the experiments performed provide a perspective on the kinds of complex multifaceted systems that may be developed using this approach.

Chapter 4

Communications in Distributed Robotic Systems

Many control systems are naturally distributed. This is due to the fact that often they are composed of several physically distributed modules: sensor, command, control, monitoring, etc. To achieve a common task, these modules need to share timely information. Robotic systems are a prime example of such distributed control systems.

The need to share information in a distributed environment is common to many other application environments such as databases, file systems, distributed computing and simulation, banking and transaction systems, etc. However, distributed control systems have unique needs not fully addressed by the available approaches. The same way an inadequate mathematical formalism impedes our ability to formulate problems and prevents us from developing clean solutions, an inappropriate information-sharing paradigm hampers our ability to develop powerful interfaces and efficient solutions to distributed robotic applications.

This chapter describes the Network Data Delivery Service (NDDS) [124, 123]–a framework to communicate and distribute data tailored to distributed control applications–its role within the overall system and its relation to other research in the field. This communications layer was developed to support the information-interfaces described in Chapter 3. From its research origin, NDDS has evolved into a stable software package that has been released commercially [145][1]. In the remaining of the chapter, the following topics will be addressed:

[1]Appendix H contains a selection of pages from the NDDS manual.

1. Identification of the unique communication requirements of distributed control applications. Complex distributed robotic systems require sophisticated dataflow. The unique characteristics of the data exchanged in this context needs to be carefully identified in order to develop an approach specifically tailored to these applications.

2. Use of a publish/subscribe paradigm to model the information exchange. Periodic data exchange is quite common in distributed robotic systems (e.g. sensory updates). This data is commonly needed with minimum time delay. In this context, the publish/subscribe model provides a more efficient mechanism than client/server models because (1) a single subscription request replaces the continuous stream of client requests resulting in increased bandwidth and (2) the data exchange is initiated at the source of the data and can be synchronized with the availability of the data, hence arriving to its destination with minimum delay. The publish/subscribe paradigm is also more appropriate for event notification because it avoids the need for continuous event polling.

3. NDDS's support for multiple anonymous producers and consumers. In distributed robotic systems there are often multiple sources of otherwise identical data. A communications system must provide mechanisms for the end-user application to specify how to resolve such conflicts. In this respect NDDS characterizes data with a *strength* (priority of the data) and *persistence* (period of time during which the data is valid). These parameters can be specified by the user application for each data-item. Similarly, support for the information interfaces described in Chapter 3 requires the system to send information to multiple subscribers simultaneously without additional programming effort.

4. NDDS's realistic model of time. NDDS allows explicit specification of *custom update rates*, *deadlines* and the *actions* to take if a deadline is missed. Within NDDS all data is time-tagged and decisions are made based on the time when the data was generated, sent and/or received.

5. Distributed queries and reliable updates present significant challenges in an environment where any data item may be available from multiple sources and/or go to multiple destinations. NDDS provides mechanisms for the user to customize the semantics of this operations. For instance, the use of NDDS's *wait* and *deadline* parameters provides *return best, then first* query semantics, which offer a continuum between the extremes of *return first* and *return best*.

6. Robust architecture. NDDS's implementation is totally *symmetric* and *quasi-stateless*, absent of central servers or privileged nodes. All communicating peers are identical and use *data-aging* to decay the any cached state. All information regarding subscriptions and productions is refreshed periodically. This implementation is very robust to partial failures and communication dropouts.

In addition to the above, Section 4.5 provides a characterization of NDDS's performance relative to underlying (baseline) transport protocols such as UDP/IP and TCP/IP and a comparison of NDDS's publish/subscribe model with the client/server model provided by RPC[2].

4.1 The Role of Communications Within the System

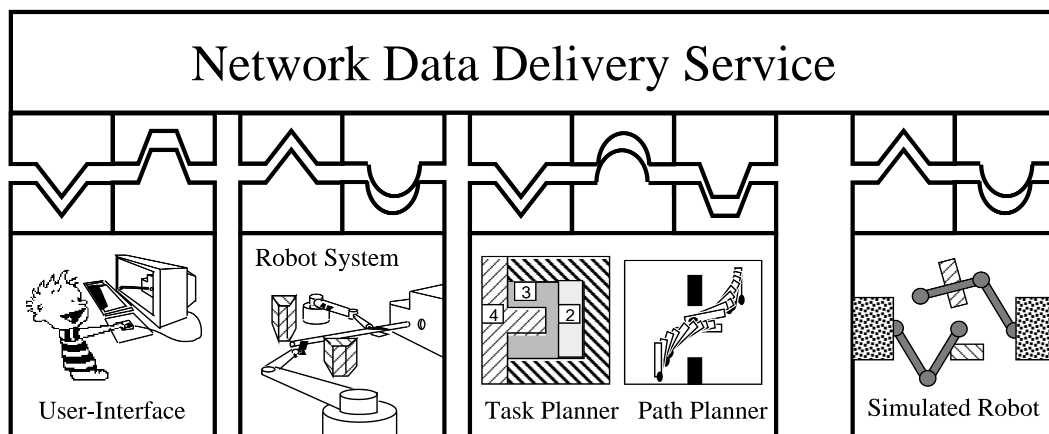


Figure 4.1: **Architecture of the Two-Armed robotic workcell.**

The main subsystems communicate using three information interfaces (represented as connectors) that are built on top of NDDS.

Chapter 3 presented the system architecture and subsystem interfaces. In that chapter, the concepts of anonymous and customizable interfaces were also presented and their advantages described. The overall architecture is shown again in figure 4.1. Rather than building each information interface from scratch, it is much simpler to develop a communications layer that supports the data-flow used by the interfaces. This layer must be flexible enough to support the different categories of information-interfaces described in Chapter 3.

The design goal for NDDS was to provide a natural model and mechanism for dealing with subsystem distribution in the context of real-time applications. No attempt is made to completely

[2]Remote Procedure Calls.

hide fundamental issues arising from subsystem distribution such as timing, delays, multiple sources of data etc. Instead, NDDS strives to provide mechanisms to deal with these issues in a natural way so that interfaces built on top can exploit these mechanism while being sheltered from the complexities of network communications, data formatting, addressing, data localization, conflicting data sources, etc. The challenges therefore encompass design and implementation of the appropriate mechanisms.

Requirements of Distributed Robotic Systems

As an application domain, distributed robotic applications have unique requirements that set them apart from other domains:

- Data exchange is often time-critical. For control purposes data must transfer from source to destination with minimum delay.
- Computations are often data-driven, that is, triggered by the arrival of new data. For instance, a collision-avoidance plan may need to be re-evaluated as soon as a new obstacle is detected.
- A significant portion of the data flow is repetitive in nature. This is true of sensor readings and motor commands. For this type of data, data loss is often not critical. Moreover, sending data is an idempotent operation and new updates simply replace old values. Considerable overhead can be avoided using a data transfer paradigm that exploits these facts.
- There are often multiple sources of (what may be considered) the same data item. For example, a robot command might be generated by a planner module as well as a tele-operation module. Similarly there can be many data consumers. A robot and a simulator are both sinks of "command-data." The network of data producers and consumers may not be known in advance and may change dynamically.
- Data requirements are ubiquitous and unpredictable. It is often difficult to know what data will be required by other modules. For instance, force-level measurements—normally used only by a low-level controller—may be required by a sophisticated high-level task planner in the future. The architecture should support these types of data flow. Thus, vital data should be accessible throughout the system.
- Most data flow can and should be anonymous. Producers of the sensor readings can usually be unaware of who is reading them. Consumers may not care about the origin of the data they

use. Hiding this information increases modularity by allowing the data sources and sinks to change transparently.

In addition, the communication system must be able to operate in readily available hardware and be portable across several architectures and operating systems (for instance a real-time platform for closed-loop control, a graphical workstation to provide a user interface, and compute engines to perform computation-intensive operations such as planning). As computers evolve, we want to maintain the freedom to select the most appropriate platform for each application.

The Network Data Delivery Service has been developed to address these unique requirements. NDDS provides transparent network connectivity and data ubiquity to a set of processes possibly running on different computers. NDDS allows distributed processes to share data and event information without concern for the actual physical location and architecture of their peers[3]. NDDS allows its "clients" to share data in two ways: subscriptions and one-time queries. NDDS uses the "publish/subscribe" model, supports multiple information sources (producers) and users (consumers). It provides clear semantics for multiple-producer conflict resolution, provides support for and guarantees multiple update rates (as specified by the consumers). NDDS's implementation is nearly "stateless" and internally uses decaying state to ensure inherently robust communication. Aside from this research, several other projects are using NDDS including an underwater robotic vehicle [195], and a self contained, two-armed free-floating robot originally described in [189].

4.2 Literature Review: Communications in Distributed Systems

When comparing the different approaches to distributed connectivity, it is necessary to examine both *what* is provided as well as *how* it is provided. The *what* refers to the functionality of the system and the resulting *interaction model* apparent to the user. The interaction model affects the assumptions the user can make about the reliability, ordering and coherence of the information as seen by the different entities that participate in the exchange. The *how* refers to the implementation aspects, that is the architecture and support mechanisms. Buried within the *how* is the *where*. Computer systems are highly layered; the operating system, high-level languages, tool-sets and mediators shelter the applications from the actual computer and communications hardware. Distribution-hiding can be performed on any of the layers. As a general rule, the lower the layer, the more efficient and

[3]NDDS is being used to communicate between Sun, HP, SGI and DEC workstations as well as VME-based real-time processors running the VxWorks operating system.

transparent the communications can be. Unfortunately this comes at the expense of flexibility and portability.

4.2.1 The applications' view of the information exchange

Independent of the actual implementation, the communications framework provides a model of the information exchange to the applications that communicate using the framework. The success of a specific model depends on how natural and efficient it is for a given application. While no model can be appropriate for all applications, several have had widespread acceptance:

The Shared Memory model presents the illusion that there is a global (shared) memory where data shared by different applications is stored. Communication occurs by writing into and reading from the global memory. In the shared memory abstraction any changes must be seen seen *simultaneously* and *consistently* by all the peers. This is a powerful and familiar model because it allows the application to treat the system as if it was not distributed. Algorithms and programs developed for non-distributed systems can therefore run in a distributed environment without changes. The attractiveness of this model has generated substantial research [88, 117, 89]. Simulation of a true physical shared memory on a distributed system can only be achieved by the operating system. Several research operating systems: Amoeba [53, 183], the V kernel [31], provide these facilities. The problem is that this model cannot be implemented efficiently without special hardware support [198]. As a result the model has been relaxed by increasing the granularity of the shared elements. In the *shared data-object* model, shared data is encapsulated in objects or other structures and accessed through operations on those objects. This abstraction can be provided by the operating system as in Clouds [46, 40, 11], Chorus [149], by a language and run-time system as in Linda [3], Eden [53, 9], and by dedicated environments such as Distributed Blackboards [61, 122].

Despite its familiarity, the shared memory model is not a natural paradigm for many applications. It is difficult to develop event-driven applications, notice changes in data and synchronize or wait for those changes. Constructs such as semaphores provide some of this functionality, but short of continuous polling, there is no mechanism for ensuring that no changes have occurred to specific data. In addition, the requirement that all participants maintain a consistent view of memory makes memory updates expensive and unpredictable in their delay (remote copies may have to be invalidated). This is inappropriate for real-time applications.

Request/Response model. In this model information exchange occurs by issuing requests and waiting for corresponding responses. Examples of such exchanges are the client/server model and the remote invocation model.

The client/server model is asymmetric and intended for master/slave type requests. A client (master) issues a request and waits for the response from the server. The server (slave) is normally idle waiting for requests from the clients. This model is very popular and has been used to implement file systems [178], window systems [44], and distributed databases among other things. Suites of protocols and supporting software have been developed, notably *Remote Procedure Calls* (RPC) [17, 107]. However the client/server model has several drawbacks. For example, the client blocks while waiting for the response precluding the client from issuing concurrent requests or even taking advantage of the time lapse until the response arrives. Also, two messages are required to receive data increasing the latency and diminishing throughput of the information flow, especially when communication delays are significant. It is also difficult for a server to notify a client of events (which may be useful to provide partial answers/results or status/progress information). These shortcoming have been recognized and addressed by augmenting the basic client/server model with the introduction of Asynchronous RPC [10] and related mechanisms such as Futures [192], Promises [93], Upcalls [33], Maybe-RPC [203], Sun's Batched-RPC [107].

The remote invocation model extends object-oriented programming to a distributed environment. Objects interact by exchanging messages unaware of their physical location. In this exchange, the object-to-object relationship is symmetrical (the sender and receiver roles are specific to each invocation). This abstraction can be supported by the operating system as in Sun's DOE[4] [181], Amoeba [183], Chorus [149], or at the language level as in Emerald [140, 76]. To locate an object or a service, a run-time system (or the operating system) must maintain a directory of the available services, objects and/or interfaces. To communicate between different machines and applications from multiple vendors, standard object interfaces, data-representations, and invocation methods must be used. Several standards such as Sun's RPC and XDR[5] [107, 108] and the Object Management Group's CORBA[6] [121] are in use.

Despite the power, buried within the client/server model, is the concept of peer-to-peer exchanges where one-request is followed by its response. For this reason, the client/server model

[4]Distributed Objects Everywhere, an initiative of Sun Microsystems.

[5]XDR stands for eXternal Data Representation and is a computer-independent format to represent data. XDR was originally developed by Sun Microsystems but has since become the industry standard.

[6]Common Object Request Broker Architecture.

is inappropriate for applications where most of the data flow is one-way, one-to-many and where computation is data-driven.

Publish/Subscribe model. In this model, information providers publish (advertise) the information they can provide. Consumers subscribe to the information they require and receive updates from the providers at the discretion of the sender. The consumer only needs to contact the producer if its information needs change. This model offers significant advantages in situations where data transferred corresponds essentially to time-changing values of an otherwise continuous signal (such as sensed data or control signals). A single subscription replaces a continuous stream of requests. Moreover the data is transferred with minimum delay since the exchange is one way and synchronous with the availability of new data. The publish/subscribe model is also notification-based. This is very beneficial when a system needs to monitor a great number of perhaps infrequent events. In a client/server or implicit model, the monitoring process must continuously poll for possible changes. One-to-many communications are easily supported because the sender decides when to send data and can take advantage of multi-cast and broadcast mechanisms. Publish/subscribe models have recently been used for financial applications [175], command and control systems [19, 48, 201, 177] and desktop tool communications [180, 181].

4.2.2 Implementation aspects

In general, all layers from the operating-system up to the application are involved in supporting the application's data exchanges. However, the decision as to which layer takes responsibility for hiding the distributed aspects from the layers above has deep consequences:

Operating Systems provide the basic communication services used by the applications to communicate among distributed computers. Distributed Operating Systems such as the ones described in [183, 113, 53], can hide the distribution from the applications and provide any of the information exchange models described before. For example Chorus [149, 71] and Clouds [46, 11] provide distributed virtual memory, while Amoeba [183], Argus [173], the V kernel [31], OSF[7] DCE[8] [153], Sun's DOE [181], provide transparent access to remote objects and services. To date no distributed OS supports the publish/subscribe model directly. The use of the OS to hide the distribution forces

[7]Open Software Foundation. A consortium of several computer vendors.
[8]Distributed Computing Environment.

all the applications to be running the same OS. This is inappropriate in situations where different operating systems must be intermixed[9].

Computer Languages provide another level of interaction with a computer system. Language constructs and run-time systems can be used to provide distributed connectivity [14, 32]. Languages such as DOWL [2], Emerald [90, 140], and Linda [3] offer the advantage of special constructs to make interaction between distributed applications natural (language constructs can be tailored to the specific communications model). The language approach is more portable and flexible than the OS approach because it can be implemented on top of multiple operating systems. One disadvantage is the requirement of porting all applications to use the same language. This makes it unsuitable for many applications such as the one in this thesis because none of the aforementioned languages has achieved widespread use and the corresponding development tools (compilers, debuggers etc.) do not exist for many computer architectures. To the author's knowledge, none of the language approaches support the publish/subscribe model.

Mediators and Run-Time Systems. Run-Time systems which may contain active mediators (brokers, agents, daemons) can be used to provide yet another layer on top of OS and languages. This approach is the most flexible and portable since a variety of operating systems and languages can be supported. Typically the applications are linked to interface libraries that use mediators and run-time systems to facilitate information exchange. This is the approach followed by NDDS and all systems that support the publish/subscribe model including TCX [48], TelRIP [201, 202], the Teknekron Software Bus [175], SPLICE [19], and CRONUS [15]. This is also the approach used by most implementations of the CORBA specification.

4.2.3 Review of Related Approaches

The following review will be restricted to approaches that use mediators and run-time systems to support the communications. There are two reasons for this restriction: First these are the most portable approaches, the only ones that can run on a variety of hardware and software platforms[10]. Second, these are the approaches most closely related to NDDS.

[9] For instance, the experiments in this thesis require the flexibility of a Unix workstation for the planning and human interface subsystems, and also require the predictable performance of a Real-Time Operating System running on dedicated processors to control the workcell hardware.

[10] This requirement is imposed by the robotic workcell experiment

The Common Object Request Broker Architecture (CORBA) is a specification [121] developed by the Object Management Group (OMG)[11]. CORBA provides a mechanism for language-independent definition of objects and network-transparent communications between objects. The main objective is to interconnect objects supplied by different vendors. It specifies both a language in which objects can be described in terms of their interfaces (IDL[12]), and a general architecture and set of facilities for accessing the objects that must be provided. CORBA does not specify any implementation and therefore a wide variety are possible (e.g. library based, agent based, use of a central server, implementation as part of the operating system, etc.).

CORBA differs from NDDS in that the paradigm for object interaction is request/response (not producer/consumer)—CORBA requires specification of the (unique) object to which a message is sent. Unlike NDDS, CORBA does not provide facilities for multicast nor multiple producers and consumers. Also, CORBA does not specify any mechanism for "notifications" which are critical for event-driven systems. Still, it would be possible to develop a CORBA compliant implementation providing some of NDDS's facilities. For example Sun's DOE [181] event management facility provides notifications and ORBIX[13] [69, 70] offers support for multiple operating systems. To date no CORBA implementation provides real-time facilities similar to those of NDDS.

POLYLITH was developed at the University of Maryland [136] as a tool to support and configure applications composed of modules running in a distributed system. Like CORBA, POLYLITH defines only the interfaces (using a custom language called MIL) between different modules. The actual implementation is not specified and can use a variety of communications mechanisms.

POLYLITH is very different in philosophy to NDDS, it is primarily a tool to configure static communication channels and distribute modules among the available processors; In POLYLITH the user must specify the topology of the module connections and processor assignments in a configuration file prior to running the system. It offers no support for subscriptions, custom update rates, nor multiple producers.

CRONUS is an environment for distributed programming developed by BBN Systems and Technologies [15]. CRONUS allows transparent access (creation and invocation) of objects in a distributed environment. The goals of the system are similar to those of CORBA. CRONUS has been

[11]The OMG is an industry consortium with the participation of several computer vendors such as Sun, HP, IBM and DEC

[12]Interface Definition Language

[13]A CORBA compliant product of IONA technologies.

used primarily to provide access to remote databases. CRONUS provides facilities for describing objects (using their own interface specification language from which stub code is automatically generated), replicating objects, and authenticating requests. The interaction model is request/response, but the use of “Futures” and “Future-sets” [192] allows multiple outstanding requests and out-of-order responses. Routines can be attached to Futures to provide a functionality similar to NDDS’s notifications.

CRONUS goals differ from NDDS. The emphasis is in allowing secure remote access to services (hence the request/response model, and authentication facilities), not in providing data distribution. An invocation is logically directed to a single server (multiple services are supported only for reliability and availability). When more than one server is present, they are expected to perform identically and a voting mechanism is used to resolve conflicts.

The Data Manager (DM). Was developed at MBARI[14] [103] to provide connectivity between applications running on top of the VxWorks operating system . Applications using the DM must declare themselves as either producers or consumers of *data items*. A data item is a named block of memory. Data items may have at most two producers (a regular one and an extraordinary one) and any number of consumers. Consumers may specify the update rate and a semaphore to be signalled when the update arrives. A *data connection* facility is provided allowing ’dynamic aliasing’ of data items (a data item becomes the source for another one, when the first item is updated the consumers of the second item are also updated). Actual communications occurs using TCP/IP and UDP/IP[15].

The DM is very similar in its philosophy to NDDS. It has, however, several limitations overcome by NDDS. Data items are simply a continuous memory buffer. This offers no data-type support, type checking nor external (machine independent) data representation; resulting in additional work for the user and increased error risk. “Regular” and “extraordinary” producers offer very limited support for multiple producers: only two producers are allowed. Moreover, each producer needs to be aware of its respective role. For instance, if the “extraordinary” producer fails, there is no mechanism to provide an alternative producer for the data. The DM has no support for consumer deadlines, queries nor reliable updates. The DM is only available for the VxWorks operating system.

[14]Monterey Bay Aquarium Research Institute.

[15]TCP is the Transmission Control Protocol, UDP is the User Datagram Protocol, and IP is the Internet Protocol. TCP and UDP are transport level protocols. IP is a network-layer protocol. These protocols are the ones used by the INTERNET, and are supported by most operating systems.

The Task Control Architecture (TCA) developed at CMU[16] [171] has a communications layer called TCX [169, 48] that provides distributed connectivity using the publish/subscribe model. This architecture runs on a variety of Unix and VxWorks platforms and uses a central-server responsible for setting up all communications. TCX uses TCP/IP for communications. The communications layer was originally developed to support TCA, a task-decomposition planner that has been used to control several robotic vehicles [85, 170]. In TCX, all applications maintain static connections with the central server where they register their interests and productions. Producers name the consumers explicitly but consumers can receive from any producer. A queue of messages is maintained by the receiver. There is only a single producer and consumer of each item. TCX supports typed data using self-describing types that are used to perform serialization and deserialization at each end.

TCX differs from NDDS in its use of a connection-oriented protocol (TCP/IP) that establishes and maintains connections, the existence of a central server, the need for producers to explicitly name consumers and, the lack of support for multiple producers, consumers, and multiple update rates.

The Teknekron Software Bus (TIB) from Teknekron Software Systems Inc. [175] was developed primarily to facilitate data dissemination for financial market applications. TIB is based on the publish/subscribe model and uses a hierarchical naming scheme so that related information may be grouped (subscriptions can be made to any data in the group). Data updates are self-describing using a custom data representation and sent using reliable datagrams. The distribution scheme is based upon “Active Repositories” that get periodic updates from producers and relay the information to all subscribers. These repositories can be placed as gateways to dedicated nets in order to increase fan-out and bandwidth of the dataflow.

TIB differs from NDDS in its support for hierarchical naming and use of intermediate agents (the repositories) to increase fan-out. It has been optimized for throughput to many (thousands) of consumers, not for latency with is increased by the intermediate repositories. TIB has no support for best-efforts delivery, multiple update rates, deadlines or multiple producers.

TelRIP[17] is an architecture developed at Rice University [54, 201] to communicate and share data in distributed telerobotic applications. TelRIP uses a fully distributed architecture without central nodes. Each network node contains a dedicated daemon process to serve as a gateway. These daemons are connected to each other using TCP/IP and route all the messages between applications.

[16] Carnegie Mellon University.

[17] Tele-Robotic Interconnection Protocol a.k.a. "Inter Agents".

This approach yields a very scalable architecture but increases communication delays. TelRIP does not support deadlines, custom consumer rates, best efforts delivery or multiple producers.

SPLICE[18] was developed at Hollandse Signaalapparaten [19] to provide the communications backbone for command and control applications. In SPLICE applications refer to data using unique identifiers. Data is typed and is declared using SPLICE's own Pascal-like language. Updates are self-described (the type of the data is sent along). The actual communications are handled by SPLICE agents tied to each application. SPLICE provides a mechanism that allows specific fields within a data-type to be designated as keys; consumers can choose to treat updates that differ in their keys as completely different items (stored and retrieved independently) or just different updates of the same item (only one the latest copy is stored). SPLICE also contains a mechanism similar to NDDS's for arbitrating among different producers of the same data. Each producer specifies the quality and persistence of its updates.

SPLICE is quite similar to NDDS in its philosophy and goals but has significant implementation differences: SPLICE's key-based hierarchical naming is more flexible than NDDS's flat-name scheme, but otherwise, the actual data instances (identifiers) produced and consumed by each SPLICE application must be known and specified at compile time using SPLICE's language. SPLICE offers no support for multiple update rates or deadlines and consumers must poll for updates (where NDDS uses notifications)—a more restrictive and inefficient mechanism.

Others. Many other researchers have recognized the need for a network communications layer allowing both data-transfer and event-signalling, many of these approaches have been tailored to specific applications or embedded within an architectural model. Examples of this are APHRODITE [185] and MICA [134].

4.3 The NDDS Communications Model

The NDDS system builds on the model of information producers (sources) and consumers (sinks).

Producers register a set of data instances to be produced, unaware of prospective consumers and "produce" the data at their own discretion. Consumers "subscribe" to updates of any data instances required without concern for who is producing them. In this sense the NDDS is a "subscription-based" model. The use of subscriptions drastically reduces the overhead required by a client-server

[18] Subscription Paradigm for the Logical Interconnection of Concurrent Engines.

architecture. Occasional subscription requests, at low bandwidth, replace numerous high-bandwidth client requests. Latency is also reduced, as the outgoing request message time is eliminated.

Function	Action
NddsProducerCreate	Create and specify producer parameters
NddsProducerAddProduction	Add an item to the list of productions published by the producer
NddsProducerSample	Take a snap-shot of all the items produced by the producer. For immediate producers also send updates to all consumers.
NddsConsumerCreate	Create and specify consumer parameters
NddsConsumerAddSubscription	Add a subscription to a consumer. Specify a call-back routine to be called when updates arrive
NddsConsumerPoll	Poll the consumer for updates. Will result on call-back routines being called when applicable. Required only of polled consumers.
NddsInstanceQuery	Issue a one-time query.

Table 4.1: **Functional interface to produce, consume and query data.**

Producing data involves three phases: Creating (declaring) a producer, declaring the instances the producer will publish (produce) and sampling the producer. Receiving data updates involves two (possibly three) phases: Creating (declaring) a consumer, declaring the instances that the consumer subscribes to along with the action to be taken when an update arrives and optionally (polled consumers only), polling for updates.

NDDS identifies data instances by name (their *NDDS name*). The scope of this name extends to all tasks sharing data through NDDS. Two instances with the same NDDS name are viewed by NDDS as different updates of the *same* data instance and are otherwise indistinguishable to the client. If two data instances must be distinguished by any NDDS client, they must be given a different NDDS name.

NDDS requires all data instances to be of a known type. NDDS has some built in types (such as strings and arrays) but most data flow consists of user-defined types. Creating a new *NDDS type*

involves binding a new type-name with the functions that will allow NDDS to manipulate instances of that type. NDDS provides a tool *nddsgen* that can be uses to automatically generate code for user-defined types from the type-specification given in the XDR language (see Appendix H for details).

NDDS treats producers and consumers symmetrically. Each node maintains the information required to establish communications. Producers inform prospective consumers of the data they produce. Consumers use this information to either subscribe to data or issue one-time queries. Table 4.1 lists the steps involved in becoming a producer or consumer of data.

4.3.1 Producer Characteristics

A producer can be compared to a multi-channel Sample-and-Hold. It is associated with a set of object instances (similar to the signal channels) that are sampled synchronously. Sampling a producer takes a snapshot of the values of each data item the producer has associated with it. Depending on the type of producer, the data is either immediately distributed or sent at a later time by a helper agent.

NDDS supports three types of producers: *asynchronous, signalled*, and *synchronous*:

Asynchronous producers send the updates immediately after the producer is sampled (in the context of the task calling NddsProducerSample()). This achieves minimal latency but slows down the sampling task which may be unacceptable for certain critical real-time tasks.

Signalled producers differ in that the task sampling the producer does not perform the network transaction, instead it signals a slave task to do the network transaction on its behalf. This achieves almost the same latency as the asynchronous producers case but doesn't slow down the producer task.

Synchronous producers are similar to signalled producers in that the network transactions are performed by a daemon in behalf of the producing task, but there is no signalling of the daemon. Instead the daemon collects all the updates together and sends them synchronously at a certain rate (provided there is something to be sent). Synchronous productions can potentially achieve higher bandwidth than the other types, because the daemon updates from different producers can be grouped together into single messages.

A producer is characterized by three parameters: *production rate*, *strength* and *persistence*. The strength and persistence parameters are used to resolve multiple-producer conflicts. Their meaning

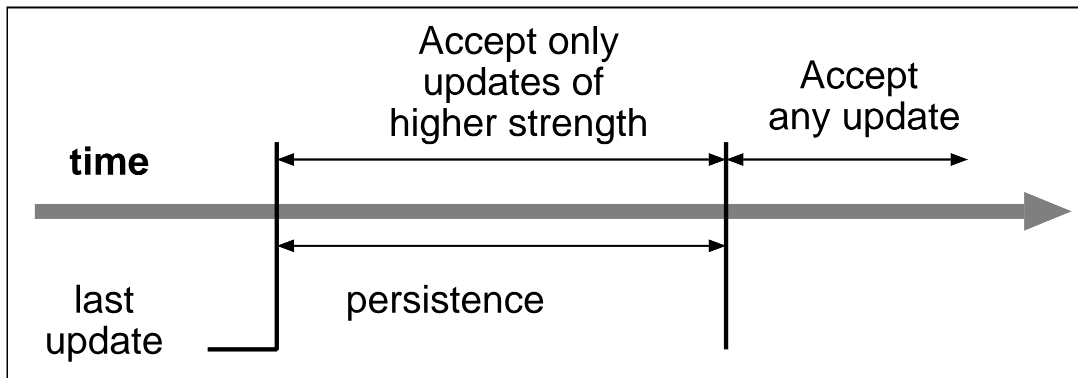


Figure 4.2: **Multiple producer conflict resolution.**

NDDS resolves the multiple-producer conflict by characterizing each producer with two properties: the producer's **strength** *and its* **persistence**. *When a data update is received for some object instance, it is accepted if either the producer's strength is greater (or equal) to that of the producer of the last update for that instance or, the time elapsed since the last update was received exceeds the persistence of the producer of the last update. In essence the strength is similar to a priority and the persistence is the duration for which the priority is valid.*

is illustrated in Figure 4.2. A producer's data is used while it is the strongest source that has not exceeded its persistence. Typically, a producer that will generate data updates every period T will set its persistence to some value T_p where $T_p > T$. Thus, while that producer is functional, it will take precedence over any producers of less strength. Should the producer stop distributing its data (willingly or due to a failure), other producers will take over after T_p elapses. This mechanism establishes an inherently robust, *quasi-stateless* communications channel between the strongest producer of an instance and all the consumers of that instance.

4.3.2 Consumer Characteristics

Consumers are *notification based*. They subscribe to a set of instances (identified by their NDDS name) by providing *call-back functions* for each instance to which they subscribe. When a data update arrives, the call-back function of every consumer is called with the data-item as a parameter.

Two consumer models are currently supported: *immediate* and *polled*. An immediate consumer will be called back as soon as the data update arrives. A polled consumer will not be called back until it itself "polls" for updates.

Consumers are characterized by two parameters, the *minimum separation* and the *deadline* (see Figure 4.3). These parameters are used to regulate consumer update rates. Consumers are guaranteed updates no sooner than the minimum separation time and no later than the deadline.

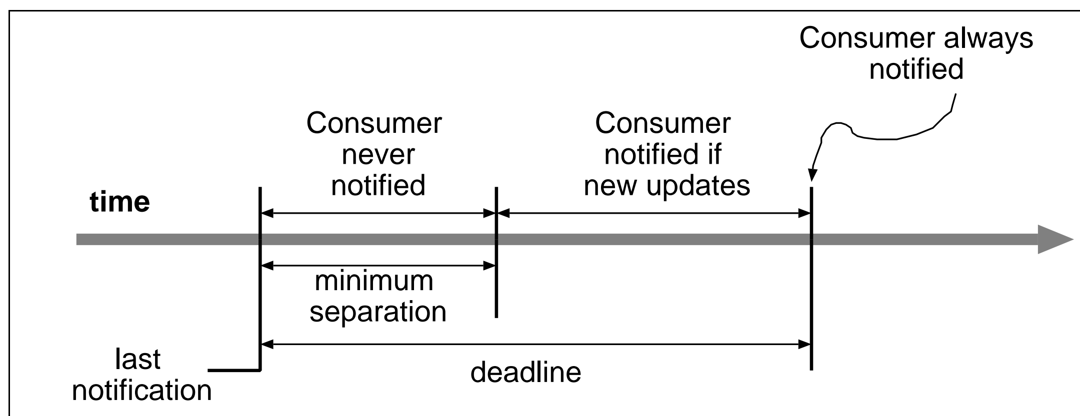


Figure 4.3: **Consumer notification rate.**

The NDDS characterizes consumers requests for periodic updates with two properties: the consumer's **minimum separation** *time and its* **deadline**. *Once the consumer is called with an update for an object instance, it is guaranteed not to be notified again of the same instance for at least the minimum separation time. The deadline is a the maximum time the consumer is willing to wait for a new update. Even if new updates have not arrived, the call-back routine will be called when the deadline expires.*

Typically the minimum separation protects the consumer against producers that are too fast whereas the deadline provides a guaranteed call-back time which can be used to take appropriate action (the expiration of the deadline typically indicates lack of producers or communications failure).

4.3.3 One-time Queries

A client task may issue one-time queries for specific NDDS data instances.

Queries are blocking calls. Aside from specifying the name and type of the NDDS data instance, a query contains two parameters: the *wait* and *deadline* illustrated in Figure 4.4. These parameters regulate the tradeoff between returning as soon as data becomes available and waiting for "better" data[19]. The use of these parameters makes the latency of this call predictable, allowing its use from real-time application code. Typically the wait is set to be long enough to account for communication delays from all producers to the consumer. The deadline provides a guaranteed call-back time in case no responses arrive. Setting a wait time to zero causes the first response to be accepted (accept first). In other words, these two parameters provide *accept best, then first* semantics to distributed queries. This semantics offers a continuum between the *accept first* and *accept best* semantics common to other systems.

[19]i.e. data from a stronger producer.

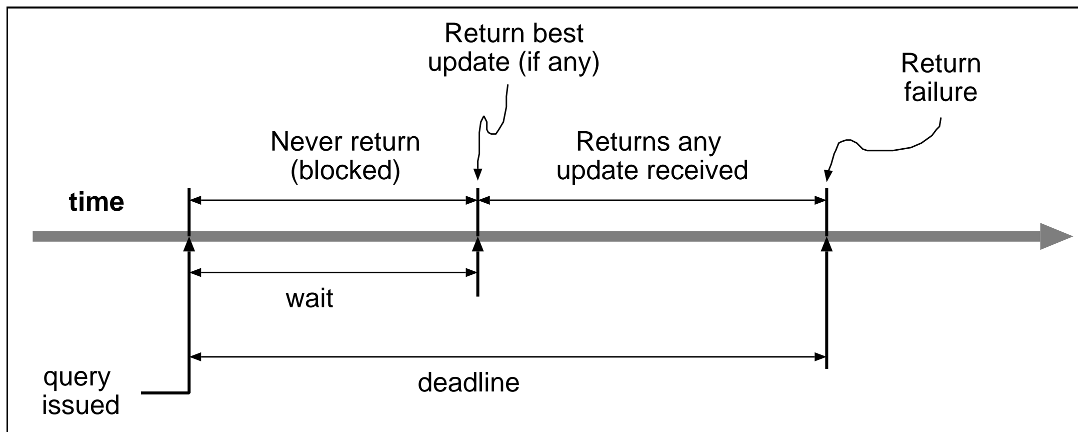


Figure 4.4: **One-Time Query Parameters.**

A one-time query specifies two parameters: the **wait** *time and the* **deadline***. A query will block for at least wait time. During the wait, data arriving from different producers is received and only the one from the highest strength producers saved. At the end of the wait time the query returns if any data has been received. Otherwise, query remains blocked until either an update comes or the deadline expires (whichever comes first).*

4.3.4 Reliable Updates

By default NDDS provides unreliable, unacknowledged data updates from producers to consumers. While this gives the most throughput with minimum overhead, it may result in occasional loss of updates or out-of-order arrival. For sensory-type data, having the latest data as soon as it becomes available is usually more important than occasionally missing an update. Other kinds of information, such as commands, often present in distributed control systems, would be better served with a reliable protocol.

NDDS supports reliable updates. A producer may specify any one of its productions to be delivered reliably. Reliable updates are grouped together in special packets that are individually acknowledged. The producer is notified if the update is not acknowledged by at least one consumer after a specified deadline and/or if another reliable production is attempted before the previous one was acknowledged. This guarantees in-order update delivery at the expense of reduced update bandwidth. A window size larger than one may be specified to compensate for longer communication delays so that multiple reliable updates can be sent before any acknowledgement is received.

4.4 Implementation

NDDS is symmetrically distributed, that is, there are no "special" or "privileged" nodes or name servers. All NDDS nodes are functionally identical and each node maintains its own copy of the NDDS database and contains the helper processes necessary to implement the communication model described above.

NDDS uses UDP/IP [135, 34] as a means of communication. To allow communications between computers with different data formats the External Data Representation (XDR) [108] is used.

4.4.1 Architectural Overview

An NDDS *node* is composed of one or more NDDS client processes (each with its respective NDDS Server Daemon) a copy of the NDDS database and three daemon (helper) processes that maintain the database and implement the NDDS communication model described above. See Figure 4.5.

The user task becomes an NDDS client by linking to the NDDS library. Each NDDS client process spawns a private NDDS Server Daemon process that will assist in establishing the subscriptions and informing the peer nodes of the productions. There is at most one NDDS node per address space so in operating systems that support shared memory threads (for example VxWorks), several NDDS client processes may belong to the same node (sharing the same copy of the NDDS database and helper daemons[20]).

The following is a functional description of the different daemons:

- **NDDS Forwarding Daemon** (NFD). There is one NFD per network host. All the Request Receiver Daemons running on the host register with the NFD. Production notifications and subscription requests received by the NFD daemon are immediately forwarded to all the Request Receiver Daemon(s) running on the host.

- **NDDS Server Daemon** (NSD). Each NDDS client (user-task) spawns its private NSD. The NSD is responsible for periodically informing all other NDDS nodes of both the subscription requests and the productions of the NDDS client.

- **Request Receiver Daemon** (RRD). There is one RRD per NDDS node. The RRD is responsible for maintaining the remote subscriptions and productions in the NDDS database.

[20] In operating systems that do not support shared memory threads, such as Unix, the helper daemons are not independent tasks but rather are installed as signal handlers.

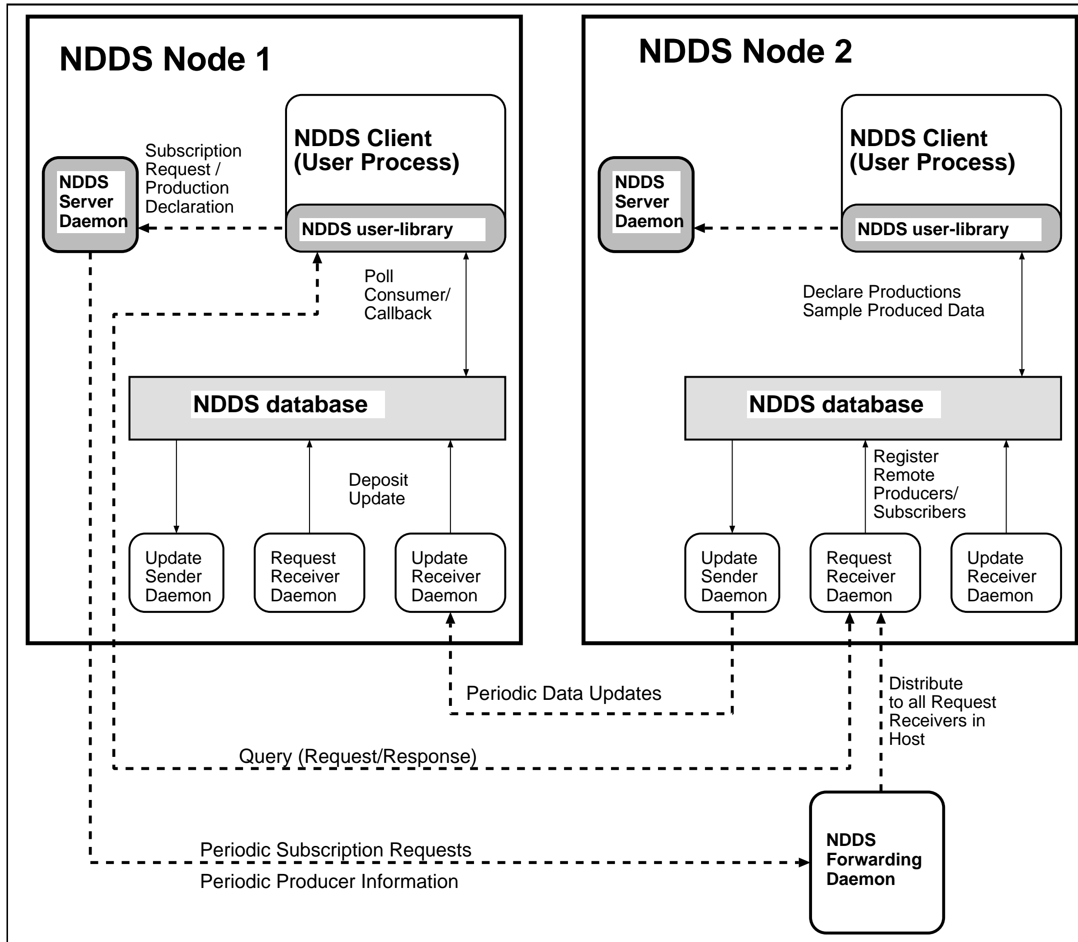


Figure 4.5: **Communication between NDDS nodes.**

A NDDS node *is composed of one or more NDDS* clients *and three helper daemons that share a local copy of the NDDS database. Each NDDS* client *has a private NDDS Server Daemon that informs other NDDS* nodes *of the productions and subscriptions of the client. The three helper daemons are responsible for maintaining the NDDS database, sending updates to remote subscribers, receiving updates and servicing queries. There is one NDDS Forwarding Daemon per Network host.*

Stale productions and subscription requests are aged and eventually dropped by the RRD. This daemon is also responsible for replying to one-time *queries* from other NDDS nodes.

- **Update Sender Daemon** (USD). There is one USD per NDDS node. The USD is responsible for sending the updates of locally produced data items to the subscribers in other NDDS nodes. This daemon also ensures that the timing parameters requested by the consumer are met.

- **Update Receiver Daemon** (URD). There is one URD per NDDS node. The URD is responsible for receiving updates for the local subscriptions of the nodes. The URD solves multiple-producer conflicts and, in the case of *immediate* consumers, executes the callback routine(s) installed for that data item. The URD also ensures that the timing parameters requested by each consumer in the node are met.

4.4.2 Data Management Overview

The NDDS database is replicated and maintained on each NDDS *node* by three helper daemons (the Request Receiver Daemon, Update Sender Daemon and Update Receiver Daemon). The database stores and cross references producers and the data they produce, consumers and the data they consume, remote productions, and subscriptions requested by the NDDS clients in both the local and remote NDDS nodes.

Consistency between databases across different NDDS nodes is not necessary and requires no special effort. Temporary inconsistencies between databases may result in subscription requests (or queries) not reaching all the producers of a given data item and, as a consequence, different nodes may receive data from different producers. A similar situation may result from the data loss due to communication failure. At worst this will be a transient situation that arises only if there are multiple producers of the same data.

All information about remote NDDS nodes is aged and is eventually erased unless it is refreshed. The NDDS Server Daemon associated with each NDDS *client* is responsible for the periodic refreshment of information that concerns that NDDS *client* . This mechanism is inherently robust to remote node failures, communication dropouts and network partitioning. Furthermore, it requires no special recovery mechanisms.

4.5 Experimental Results: NDDS

NDDS is a high-level service that provides a layer above the transport and network layers. In the OSI reference model [62, 182] NDDS logically belongs in the session and presentation layers. NDDS uses the transport-level facilities provided by the operating system (UDP/IP sockets). As a higher-level protocol, NDDS provides many useful functions: subscription management, network-wide location-transparent address (name) space, multiple-producer conflict resolution, clustering of updates into messages to diminish network traffic, deadlines etc. These facilities add extra overhead over the cost of the raw transport-level message traffic, and therefore, NDDS will be slower than

the underlying UDP/IP layer. The significance of these comparisons is to provide metrics for both overhead with respect to the underlying transport layer and expected performance benefits when compared with other higher level protocols such as RPC.

In this section, NDDS is compared with a group of four protocols: UDP/IP, TCP/IP, Sun's RPC and Sun's Batched-RPC. The four protocols in the group were selected because they are widely available and their performance is well known. This group contains the protocols used by NDDS and all the other approaches reviewed in Section 4.2.3. Therefore, comparing them with NDDS will provide two metrics: (1) the overhead introduced by NDDS and (2) the limits on the performance achievable by any high-level protocol that uses one of the transport-level protocols in the group.

NDDS's overhead stems from several facts:

- Extra information transmitted with each update. Each individual message contains extra information indicating the number and identity of the different updates as well as the relevant parameters (strength, persistence) and timing information. This overhead is approximately 50 bytes per message and 50 additional bytes per individual-item update.
- Computation to parse update messages and call the appropriate action routines for each update item.
- Computation to provide update-rate guarantees and arbitrate among multiple-producers. These services require gathering and processing of several time stamps per update.
- Extra data copies to support multiple asynchronous consumers and enable coalescing of updates directed to the same NDDS node.

Nonetheless, NDDS provides more than increased functionality at some performance cost, in cases where the characteristics of the data exchange match NDDS's model, as is the case on distributed-control systems, this overhead is balanced by the benefits provided by NDDS's optimizations.

Two performance measurements are critical when sending updates within distributed control applications: (1) the time delay from data sending to its arrival at the other end (latency) and (2) the fastest update rate at which data can be sent (throughput). In control applications, latency is important because it contributes to the phase lag in measurements and control commands impacting stability. Update rate is important because it affects the rate at which control loops can be closed, impacting achievable performance.

4.5.1 Experimental Context

All times shown in this section correspond to measurements taken on a network of Sun workstations. Although NDDS was specifically designed (and is commonly used) to communicate with computers running networked real-time operating systems such as VxWorks [200], the data presented here does not involve any real-time nodes. These computers are not as widely available and therefore the comparative results would be more difficult to relate to a familiar environment. All workstations are connected by a 10 MB/s Ethernet LAN .

The absolute times and rates are obviously dependent on the actual computer hardware (both speed of the computers and network). The real significance of the data arises when viewed as a comparison with the underlying transport delay. The relative performance is likely to remain more constant when faster hardware is used.

Unless otherwise specified, the producer of all updates is a Sun Sparcstation-10-51 (SS-10) computer, the consumers are mostly SparcStation IPX computers. In the few experiments requiring more than four computers, SparcStation IPC (SS-IPC) computers were also used. The SS-10 is approximately 3 times faster than the SS-IPX[21], which itself is about twice as fast as the SS-IPC. All computers are running SunOS 4.3.1. In the plots in this section, the adjective “fast” is used to denote a SS-10 computer (or similar) while “slow” denotes and SS-IPX or SS-IPC.

NDDS uses XDR to encode all data in a computer-independent fashion. The cost of XDR encoding varies widely across computer systems (e.g. it is essentially free if the internal machine representation matches XDR’s representation). To prevent this extra layer from obscuring the main issues, bulk data sent in all the messages is untyped (i.e. requires no encoding). However, NDDS sends some extra information with every update (about 50 bytes per message + 50 bytes per update) which is subject to XDR encoding.

All the data taken in this section requires comparing time-stamps taken in different computers. To do this meaningfully, the respective clocks must be adequately synchronized. Although there are standard time-synchronization protocols available, notably NTP [110], measured accuracy is on the order of 5 $msec$, while sub-millisecond resolution is required. Moreover, even if the computers were synchronized at a given point, the time drift between machines is so large that the sub-millisecond accuracy would be lost in a few seconds. A time-synchronization utility was developed to solve this problem that allows computation of inter-machine time offsets with accuracy on the order of 100 μsec under programming control. All tests in this section use this utility to re-synchronize and measure clock drift with sufficient frequency to bound each measurement error down to acceptable

[21] The SS-10 performance is 66 SPECint (136 MIPS), the SS-IPX performance is 21 SPECint (28 MIPS).

levels (typically 100-200 μsec). These errors are accounted for in the error bars shown in the figures and represent the standard deviation of a large (typically 100) set of measurements.

4.5.2 Latency of the Updates

The data presented in this section will show that NDDS's overhead pays off when either the receiving computer is at least as fast as the sending one, or else when several (in this case two or more) consumers want the data.

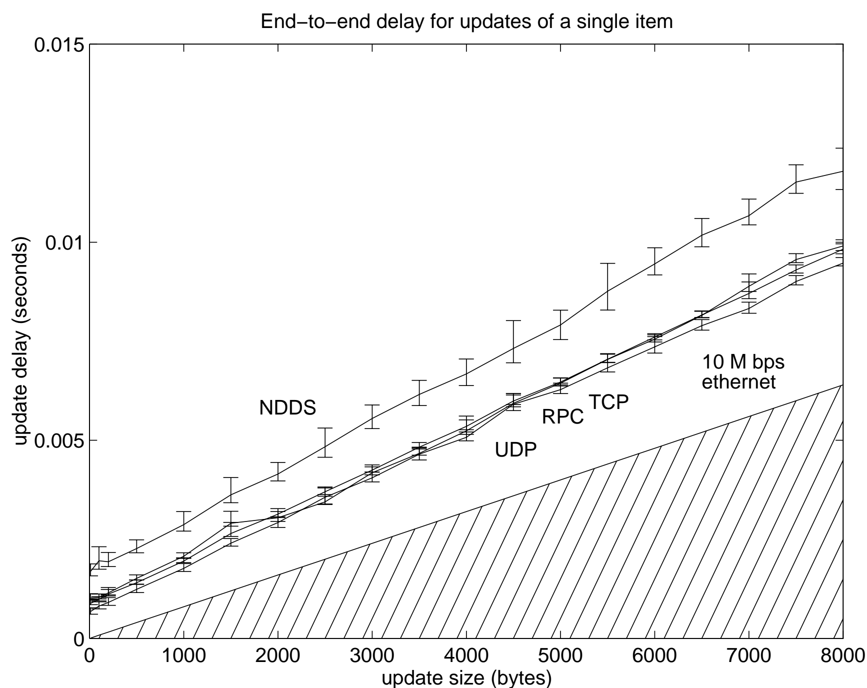


Figure 4.6: **Latency overhead of NDDS as a function of item size.**

This figure shows the total one-way delay for a single data item sent over the network from a single producer to a single consumer. The delay resulting from the raw communications bandwidth of the 10 Mbps Ethernet is shown hashed for reference. This figure shows that the overhead introduced by NDDS ranges from 1 ms for the smallest items to 2 ms for the largest ones. Overhead increases with packet size because of the extra copy performed by NDDS. This is a worst-case scenario for NDDS because it has been optimized for situations requiring multiple updates to several clients.

The first important metric for data exchange in distributed control applications is the latency or delay in the data from the time it is sent to the time it is received. For a given computer environment, this delay is a function of many parameters including data size, number of items sent, and number of hosts receiving (subscribing to) the updates. A detailed exploration of this four-dimensional space

is beyond the scope of this section. Instead, several two-dimensional cuts will be examined. In these cuts, two parameters are kept fixed and the delay is measured with respect to the remaining parameter.

Description of the experiment and data presentation. For all the measurements in this section the following simplified model is used: There is a single producer and N_c consumers, all subscribing to the same N_i items; each item is of size N_b bytes. The objective is to characterize the delay function $d = d(N_c, N_i, N_b)$.

The experiment consists of sending the N_i items from the producer to all N_c consumers. The total delay from the time just before the first update is sent to a consumer, to the time after the last update arrives to that consumer, is measured (by time-stamping the items and correcting for clock offsets). For example, for $N_i = 100$, the delay $d(N_i)$ represents the time lapsed until the arrival of the 100th item when the total number of items is exactly 100. In general, this will not be the same as say, the delay for the 100th item in the sequence when 160 items are being sent.

Each experiment is repeated many times (100 is a typical number) and the mean and standard deviation of the delay is shown as it varies with respect to the different parameters.

Influence of update size. Figure 4.6 illustrates NDDS's latency overhead for a single item sent from a producer to a consumer as a function of the size of the item. This delay represents the minimum latency or phase loss for a single data transaction and illustrates NDDS's added overhead for a single item. In the case of RPC, the delay corresponds to that of the one-way message from the client to the server. Since only one-way communication of a single item is involved, all transport protocols compared have similar performance.

In many ways, this is a worst-case scenario for NDDS, because it has been optimized to efficiently send multiple updates to several clients. The overhead incurred in support of this objective, is most damaging when there is a single data-item to be sent, because of the fixed costs incurred regardless of the number of updates. However, these measurements indicate that the overhead of NDDS is small, ranging from 1 ms for the smallest items to 2 ms for the largest ones. Overhead increases slightly with packet size because of the extra copy performed by NDDS.

Effect of number of items requested N_i. Figure 4.7 illustrates NDDS overhead versus the number of items requested by a single machine, for two different item sizes. In this example, the producer sends N_i individual items to the consumer. The delay from the time the sender sends the

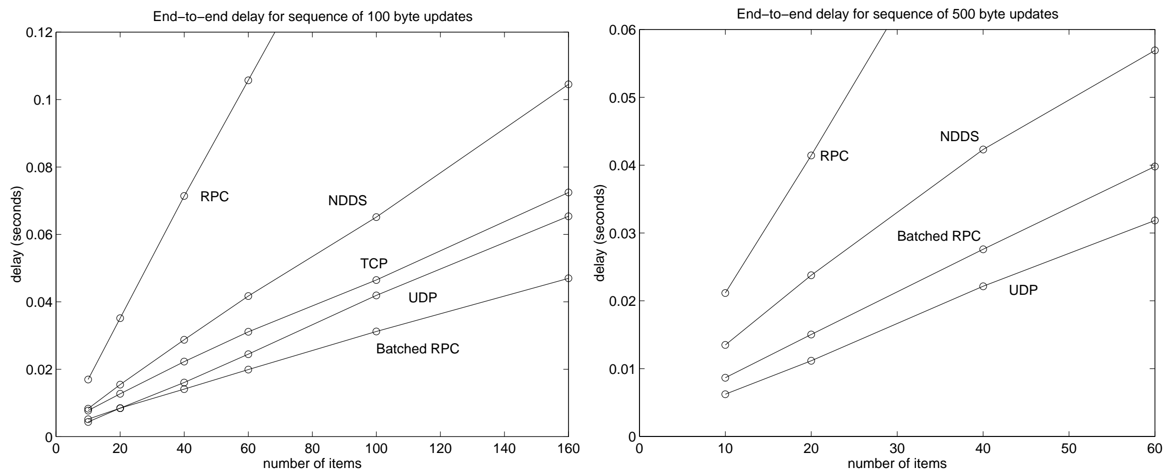


Figure 4.7: **Latency overhead as a function of the number of items**

This data represents total communications delay for a series of items sent from a single producer to a single consumer. This figure shows that NDDS's overhead increases linearly with the number of items sent. This is a result of the fact that the receiver must perform a lookup to identify the appropriate consumer/action routine for each individual update. This is only noticeable when a fast producer sends to a slow consumer (in this case the sending computer is 3 times faster than the receiving one) and when there are few consumers of each update. This remarks are substantiated by Figures 4.8 and 4.9. RPC's low performance is due to its client/server semantics requiring each individual update to be acknowledged before the next one can be sent.

first item until the time the last item is received at the consumer end is plotted as a function of the total number of items sent. The RPC, TCP, UDP and Batched RPC implementations send each item individually in rapid succession. Batched RPC is quite efficient because it pipelines individual messages without acknowledgment. After the last update is sent, a normal RPC is performed to flush the pipeline (to ensure the messages buffered in the pipeline are sent). UDP is also efficient because its messages are delivered unreliably and require no acknowledgment. In contrast, sending updates with the regular RPC requires acknowledgment of each individual update before the next may be sent, resulting in much lower performance. TCP has similar performance for the 100 byte updates but was not usable in the 500 byte case (i.e. its performance was so bad that it fell outside the range shown in the figure) because of the extra burden imposed by reliable delivery. The slope of the curves in Figure 4.7 represents the marginal cost of sending an extra item. NDDS's higher slope is primarily due to the increased computation involved in processing more updates at the receiving end and is most significant for small item sizes. Each update must trigger a notification of the appropriate installed callback function that imposes extra burden per update. In all cases its performance is within a factor of two of the fastest underlying transport protocol. Especially

remarkable is the poor performance (large delays) obtained using RPC. RPC's poor performance is due to its client/server semantics that require each update to be fully acknowledged before the next one can be sent, forcing sequential computation at the client and server sides.

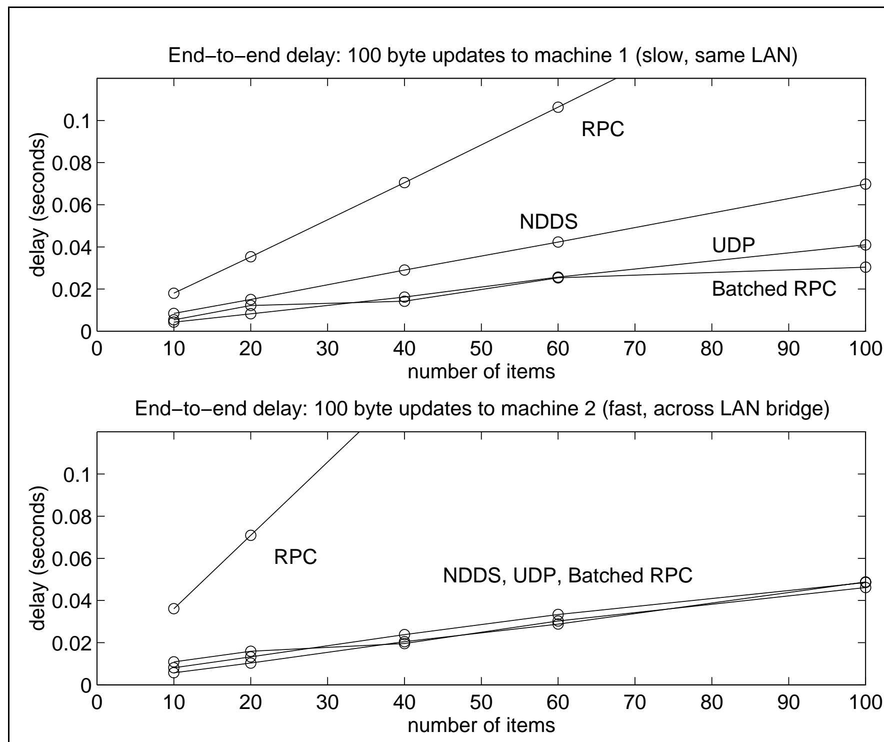


Figure 4.8: **Latency overhead as a function of the number of items for two consumers**

Total communication delay for a series of items from a single producer to two consumers. This plot illustrates that NDDS's overhead is only significant for consumers that are slow compared with the data-producers. The bottom plot corresponds to a computer of comparable performance to the sending machine for which there is no noticeable overhead.

Influence of the number of consumers N_c. Update delay as a function of number of consumers receiving the updates is illustrated in Figures 4.8 and 4.9. Figure 4.8 illustrates the case where two computes are receiving updates. Note that the data sent though NDDS has a higher delay arriving to the slow computer despite the fact that updates are sent first to the slow and then to the fast machine. This corroborates the statement that **NDDS's multiple-update-item overhead is only significant when the receiving computer is slow compared to the sending computer**.

Figure 4.9 illustrates another important point. Despite the fact that the last two receiving machines are slow, NDDS has no noticeable overhead over the underlying UDP transport layer.

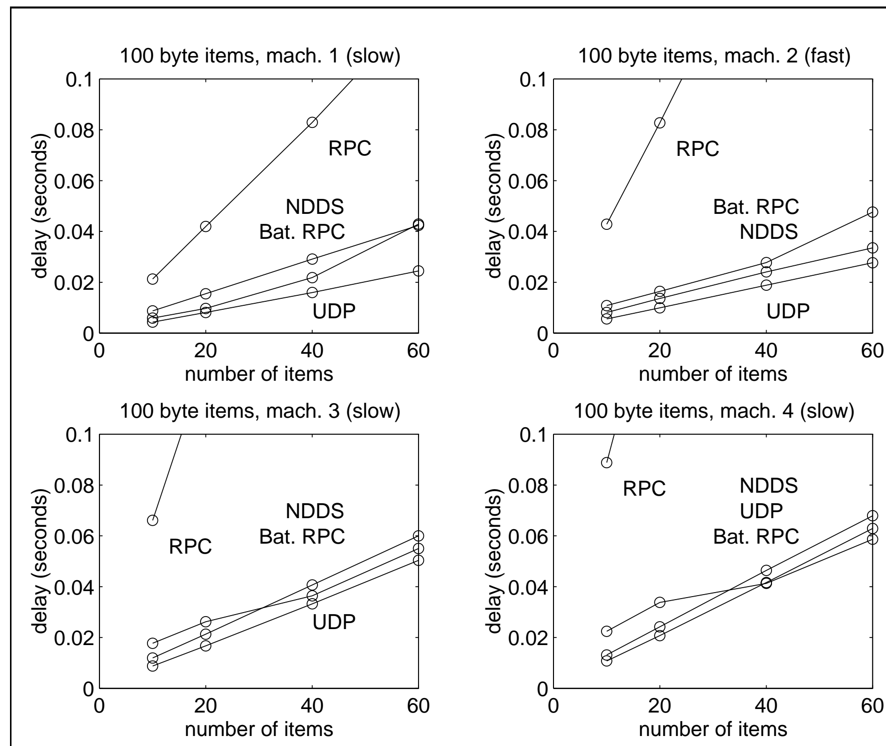


Figure 4.9: **Latency overhead as a function of the number of items for four consumers**

Total communication delay for a series of items from a single producer to four consumers. This figure illustrates that even for "slow" consumers, NDDS's overhead pays off when several consumers (in this case two or more) are subscribed to the same data.

This is *not* due to the sender becoming a bottleneck[22], but rather to the fact that much of NDDS's up-front overhead is caused by its attempted efficiency when multiple clients are involved (the normal case), and updates sent to each additional computer computer are quite efficient, resulting in delays similar to those of the raw protocols. Therefore, even in the presence of slow receivers, **NDDS's extra computation pays off when several** (in this case two or more[23]) **consumers subscribe to the same data**.

4.5.3 Maximum update rates

This section compares regular RPC using the client/server model to NDDS (publish/subscribe model) in terms of the fastest achievable update rates. This comparison demonstrates the advantages

[22]This is illustrated by the fact that the fast computer has the same delay as in Figure 4.8 where only two computers where receiving data.

[23]The number of computers for which this tradeoff occurs will depend on the relative speed of the sending and receiving computers.

of the publish/subscribe paradigm over the client/server paradigm for data updates that need to be sent repeatedly.

The previous section demonstrated that RPC's performance degrades significantly with the number of items requested by each computer N_i. This section will show that even for a very small number of requested items: $N_i = 2$ to 10 (i.e. the most favorable case for RPC), NDDS achieves updates rates that are a factor of four or greater than RPC, irrespective of the number of hosts N_c subscribing to the updates. The results also illustrate that the performance advantage of NDDS over RPC diminishes with packet size.

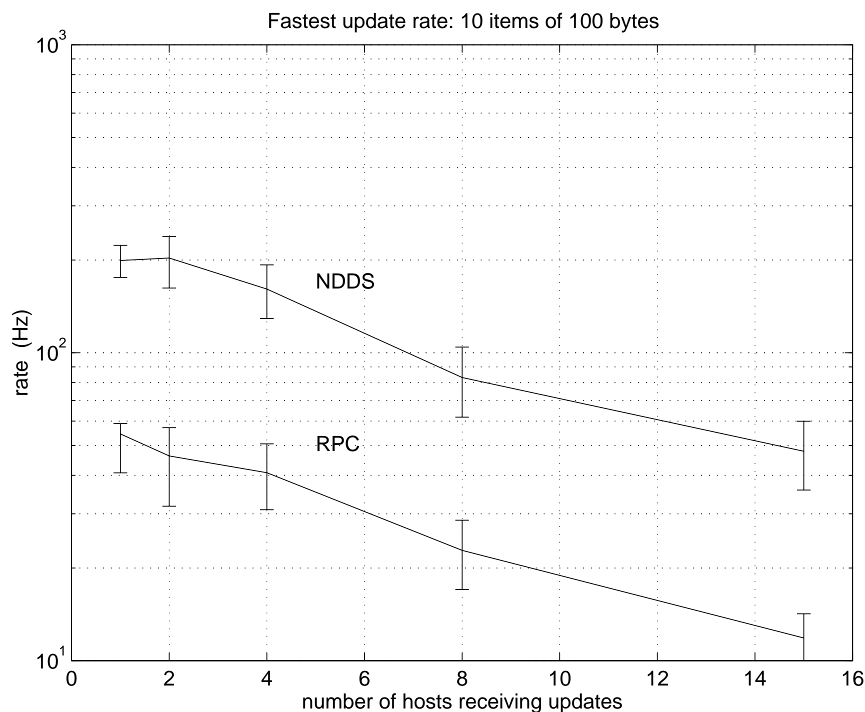


Figure 4.10: **Maximum update rate as a function of the number of subscribers.**

This experiment illustrates that for small data sizes (100 bytes), the (sustainable) update rates achievable using NDDS are a factor of four higher than those achievable with RPC regardless of the number of subscribers N_c.

The dramatic advantage of the publish/subscribe model over client/server stems from several facts:

- In the publish/subscribe model, data transactions are initiated by the sender and can therefore be synchronized with the availability of new data (assuming data is produced at a certain

periodic rate). At the receiving end, the phase-loss in the data will be exclusively due to the delay within the communications layer. In the client/server model, data transactions are initiated by the client polling for data asynchronously with the availability of new data. As a result, the expected phase-loss will be $1/2$ of the sample period *in addition* to the communications delay. The experiment in this section compare just the additional delay.

- Publish/subscribe also allows faster update rates than client/server polling because the information needs to flow only one-way (from producer to consumer); whereas in client/server, the client must first request the data which requires a message with the consequent delay. This is most significant when the underlying network is slow compared to the time taken to process the data.
- Client/server exchange is serial. Servers wait for clients to issue requests, and clients remain blocked while their request is processed. While parallelism can be achieved at the server end with the use of multiple threads, the client still remains blocked every time a service is requested. This limitation is what prompted researchers to introduce extensions to the basic RPC protocol as described in Section 4.2.
- Publish/subscribe is inherently one-to-many (producer to many consumers), this also allows the use of broadcast and multicast techniques that can be much more efficient in shared-media networks (such as Ethernet), especially as the number of consumers increases. Client/server is one-on-one.
- The client/server relationship is asymmetric. It is difficult for a server to turn around in the middle of processing a request and become a client to the same process that issued the original service call[24]. Publish/subscribe is symmetrical in the sense that any task can be simultaneously a producer and a consumer of data.

Description of the experiment and data presentation. This experiment assumes that a data source contains information that several "users" need at the fastest possible rate. Using NDDS, the data source becomes the producer, and the "users" become consumers requesting the data as fast as it is made available. Using RPC, the data source becomes a RPC server and the "users" become RPC clients issuing requests as fast as they can. This experiment compares the update rate of these two methods as a function of the number of items requested by each "user" N_i and the

[24]This situation could easily lead to a deadlock.

total number of remote "user" tasks N_c. In each case, the "user" tasks record the rate at which all N_i updates are received over a large number of iterations (typically 100) and compute the average, standard-deviation and minimum (worst) of all the rates. This data is plotted in logarithmic scale (to account for the large ranges). The error bars indicate both standard deviation (top segment of the error bar) and the minimum (worst) rate (bottom segment of the error bar).

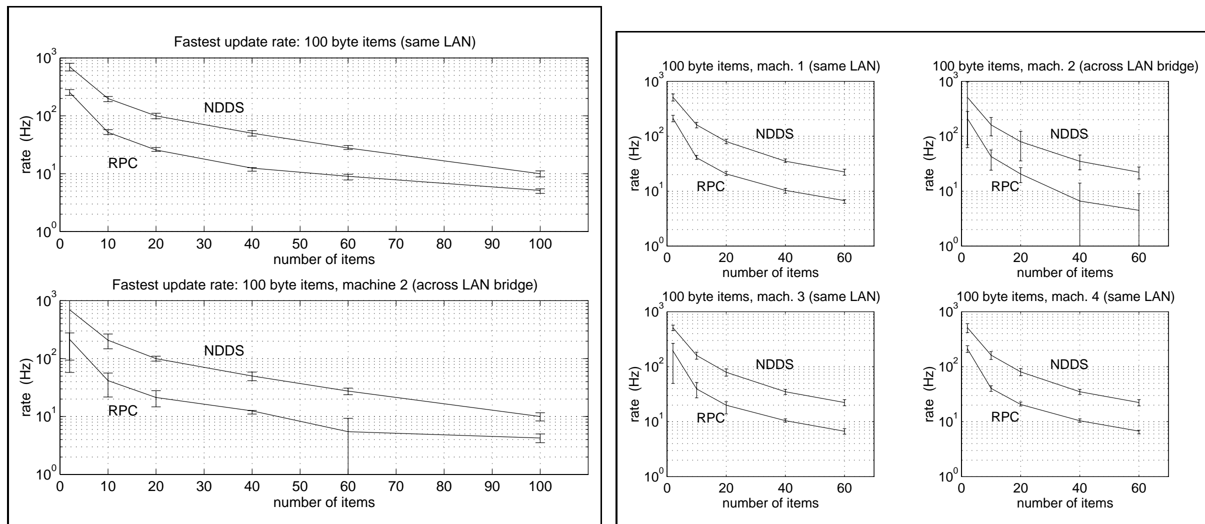


Figure 4.11: **Influence of number of items requested on the maximum update rate.**

This figure shows that, for small item sizes, NDDS outperforms client/server RPC by a factor of three to four independently of the number of items requested by each computer. This fact is illustrated in two scenarios: On the left only two computers receive updates; on the right, four computers receive updates. It is also noticeable that RPC is particularly bad when one of the computers is located across a LAN bridge (worst-case rates can be less than 1 Hz).

Overall performance Figure 4.10 illustrates the main result of this section: The maximum update rate achievable with NDDS is significantly higher (a factor of four in this case) than that achievable with RPC. In the experiment represented by the figure, the number of items requested by each computer remained constant ($N_i = 10$), as did the size of each individual item (100 bytes). The experiment allowed the number of computers subscribing to the data N_c to change from 1 to 15. For each sample in that range ($Nc = 1, 2, 4, 8, 16$), an experiment was run to determine the maximum update rate that could be consistently provided to all subscribers. Note that for one or two subscribers, NDDS's performance is limited by the speed of the (slow, SS-IPX) receiving computers (which as we will see in Figure 4.12 limits this rate to around 200 Hz even for a single consumer). For larger numbers of subscribers, NDDS's update rate is primarily limited by the sending machine.

Effect of number of items requested N_i. Assuming the computation overhead and transmission delay scale linearly with the number of items, maximum update rates are expected to be inversely proportional to the number of items. This trend is observed in Figure 4.11. This figure also confirms that the performance advantage of NDDS over RPC remains approximately constant independently of the number of items N_i. Notice also that RPC's performance can become quite unpredictable when some of the participating computers are separated by a LAN bridge.

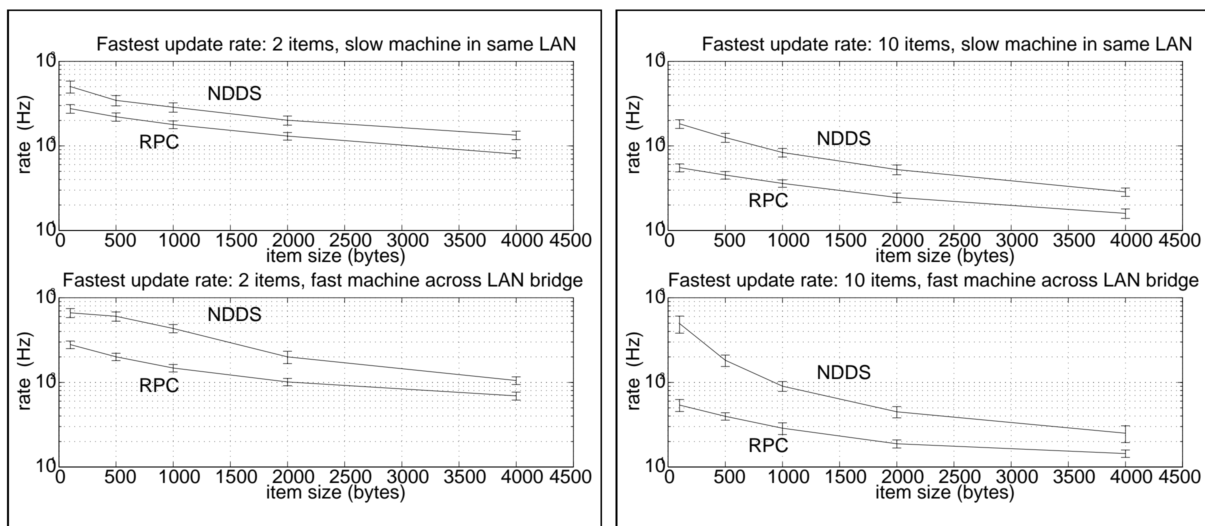


Figure 4.12: **Influence of computer performance and update size on the maximum update rate.**

This figure shows that the performance advantage of NDDS over RPC increases with the speed of the computers involved. This is because NDDS's advantage is dues to the protocol (publish/subscribe) and is only countered by the extra computational overhead that diminishes with computer performance. In the top figure, the (slow) receiver is the bottleneck. This figure also illustrates that NDDS's performance advantage over RPC diminishes with increasing item sizes, because at this point the producer becomes the bottleneck.

Influence of update size and computer performance. NDDS's performance edge over RPC is due to the advantages of the publish/subscribe model over client/server for repetitive data updates. On the other hand, NDDS's overhead over the raw transport delay diminishes with increased computer performance. As a result, the faster the computers involved, the more dramatic the difference between NDDS and RPC. Figure 4.12 corroborates this analysis. In both plots the same (fast) computer is used to send the updates. In the plot above, a slow computer receives updates while in the one below the receiving computer has the same speed as the sending one. For small update sizes, the maximum update rate achievable is 180 Hz for the slow receiver and 500 Hz for the fast receiver (this corresponds almost exactly with the speed difference between the two

machines). RPC's performance on the other hand remains constant because it is limited by the protocol (client/server) more than by the computer speed.

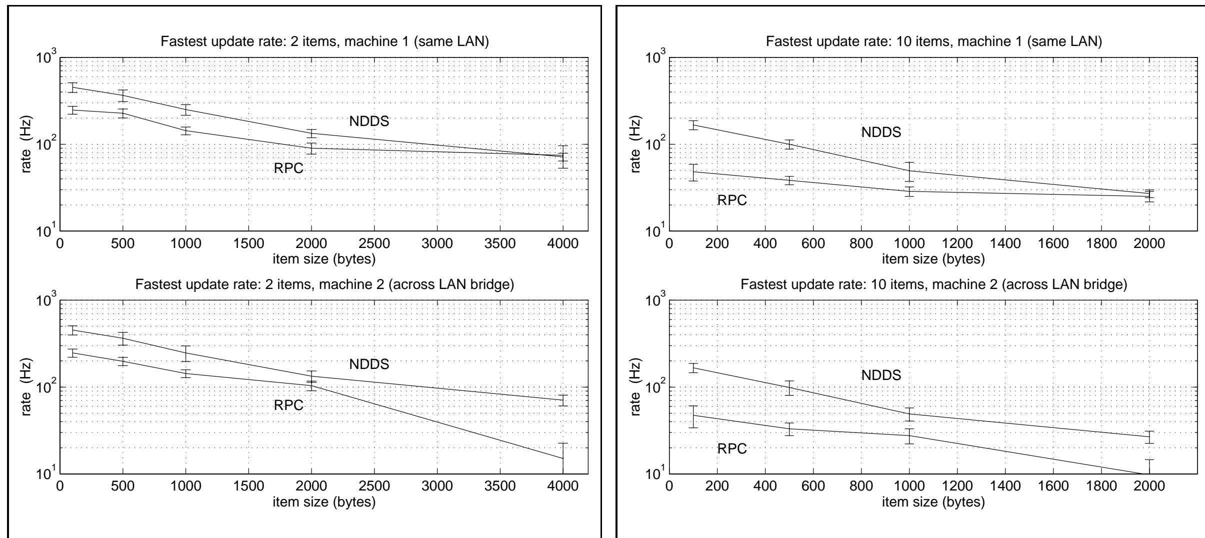


Figure 4.13: **Maximum update rate for two clients as a function of the item sizes.**

This figure shows that NDDS provides a more predictable, fair service than RPC does. This is because the producer initiates the transactions, eliminating contention and ensuring all subscribers get a fair service. When RPC is used, clients essentially contend for service and some of them (e.g. the one separated by a LAN bridge) may receive unfair service. These observations are corroborated also by figure 4.14.

For larger update sizes, the advantage of NDDS over RPC diminishes as illustrated in Figures 4.13 and 4.14. This is because for the computers used in this experiment, the producer becomes the bottleneck for updates of 500 bytes or larger (this can be seen from the fact that the update rate changes significantly in going from 2 to 4 consumers). This problem will become less significant as computer speed increases and can be eliminated with the use of broadcast or multicast techniques, which are difficult with RPC.

Notice also that NDDS provides a more predictable and fair service to the clients, all of which receive updates at the same time. This is because NDDS regulates the update rate to a value compatible with the producer and consumer characteristics (sample rate and minimum separation). In this example, the consumer has specified to receive all updates (zero minimum separation) and therefore, the rate is fixed by the common producer. NDDS avoids contention and provides orderly and fair delivery. On the other hand, when RPC is used, each client independently polls the server for updates This results in contention and one of the clients (the one separated from the LAN by a router) is systematically receiving updates at a lower rate.

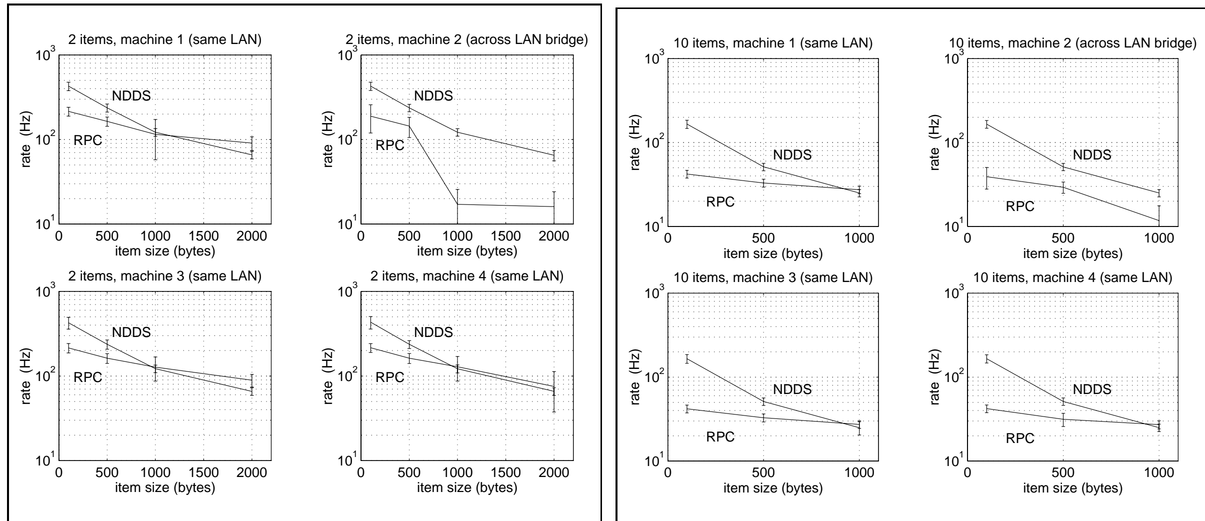


Figure 4.14: **Maximum update rate for four clients as a function of the item sizes..**

This figure shows that NDDS's performance advantage over RPC diminishes for larger update sizes. This is because in this case the producer becomes the bottleneck and it is essentially doing the same function for NDDS and RPC. This figure also confirms that NDDS provides a more predictable and fair service to all subscribers than RPC does.

RPC does have some redeeming features: first, the semantics of remote procedure calls are familiar to programmers (due to their similarity to ordinary procedure calls), second it provides automatic flow control and regulation of the information exchange. For instance, a client can be programmed to request data as fast as it can process it, and this program will work unchanged regardless of the speeds of the computers in which they run, the speed of the network communications and even adapt to changes in computer load. This is because each request/response exchange in effect provides flow control keeping client and server in lock (not allowing one to run ahead of the other). This is hard to do with NDDS because the requested update rates and deadlines are set explicitly by the application.

4.6 Summary

This chapter has presented NDDS, a communications system specifically developed to address the distinct needs of distributed robotic applications. NDDS allows programs distributed on a computer network to share data and event information unaware of the location of their peers. NDDS has several important features:

- Publish/subscribe information-exchange model.

- Support for multiple anonymous producers and consumers of any data item.
- Realistic model of time, allowing consumer-specified custom update-rates and deadlines.
- Clear semantics for multiple-producer arbitration.
- Flexible semantics for distributed query.
- Fully distributed symmetric implementation without central nodes or name servers.
- Quasi-stateless implementation using decaying state at each node.

This chapter has also analyzed two aspects of NDDS's performance: (1) overhead introduced by NDDS on top of the underlying transport protocols (UDP/IP, TCP/IP), and (2) maximum sustainable update rates compared with those using a standard client/server communications layer (RPC).

The analysis shows that NDDS's overhead becomes negligible when either the receiving computer is at least as fast as the sending one, or else when several (in this case two or more) consumers want the data.

When compared with client/server RPC. NDDS achieves update rates a factor of four greater irrespective of the number of hosts subscribing to the updates.

Chapter 5

World Modeling

This chapter describes the World-Modeling subsystem (WM). The WM is responsible for gathering, processing, and managing the raw information collected by the system sensors. In this task, the WM uses the available domain-specific knowledge to transform the information into a form suitable for use by the remaining subsystems[1]. The WM also serves as a centralized repository of all á priori known "configuration" data (e.g. object geometries, manipulator kinematics and mass properties). This á priori knowledge comes in many different forms: (1) it may be in the form of a database containing facts known to be immutable and cannot (perhaps due to cost) be sensed, (2) it may be used in model-based sensor integration algorithms, or (3) it may be embedded within the sensor processing algorithms themselves.

World modeling is needed due to the impossibility of directly measuring all the quantities and states required for the operation of the various subsystems comprising the manufacturing workcell. This need is common to systems of significant complexity that need to interact with the external world. The term "complexity" is used to (vaguely) describe the fact that the system interacts with its environment in many different ways, and is capable of a wide variety of actions. In this context, "complex" systems are contrasted with systems that use few sensors, and have very limited ways to interact with the environment. Even if their mechanical or algorithmical complexity is great. For instance, a car engine is a mechanically complex system, but its interaction with the external world presents few variations (amount and mixture of the fuel, ignition, delivered torque).

[1]The selection of such "common denominator" representation usable by the remaining subsystems is one of the difficult issues addressed by the WM.

5.1 World Modeling for the Manufacturing Workcell

The manufacturing-workcell experiment requires world modeling to characterize both the robotic equipment and the different objects in the workspace. In particular, the different subsystems require world-model information to support their function as described below:

- The Graphical User Interface (GUI) must render the robot, objects, and workspace for the user. It therefore needs to be provided with the position and shape of the objects in the workcell; and the geometry, kinematics, and kinematic state of the robot manipulators.
- The Planner requires the kinematics and kinematic limits of the manipulators in order to generate paths. It requires object and manipulator geometric shape (from a solid-model perspective) to guarantee that the generated paths are safe. Sufficient information must also be provided to generate possible object grasps. Finally, the planner must be informed of the workcell *status*[2].
- The Control Subsystem must know manipulator and object states (positions, velocities, accelerations). It also requires other information such as contact forces during grasping, manipulation and assembly.
- The Sensor Processing Subsystem uses domain-specific information to perform sensor integration/fusion. For this reason, this research encapsulates sensor processing within the world modeler[3].

Objective: The design and development of a comprehensive world modeling system for robotic workcells could itself be the focus of a complete thesis. The prototype system described in this chapter is not so ambitious. Rather, it provides simple, solutions addressing several of the major issues arising in such systems. The design described here could be used as a first step in the development of the aforementioned comprehensive system.

Philosophy: The described requirements demand a world-model subsystem that can serve a variety of roles as illustrated in Figure 5.1. First, it must *provide information* both in subscription and query

[2]The word status is used to denote information about the logical state of the workcell: What is it doing? Are any manipulators holding objects? If so which is object and grasp transform(s)? Are any arms moving? If so along which trajectories?

[3]This encapsulation will be discussed in more detail in Section 5.3

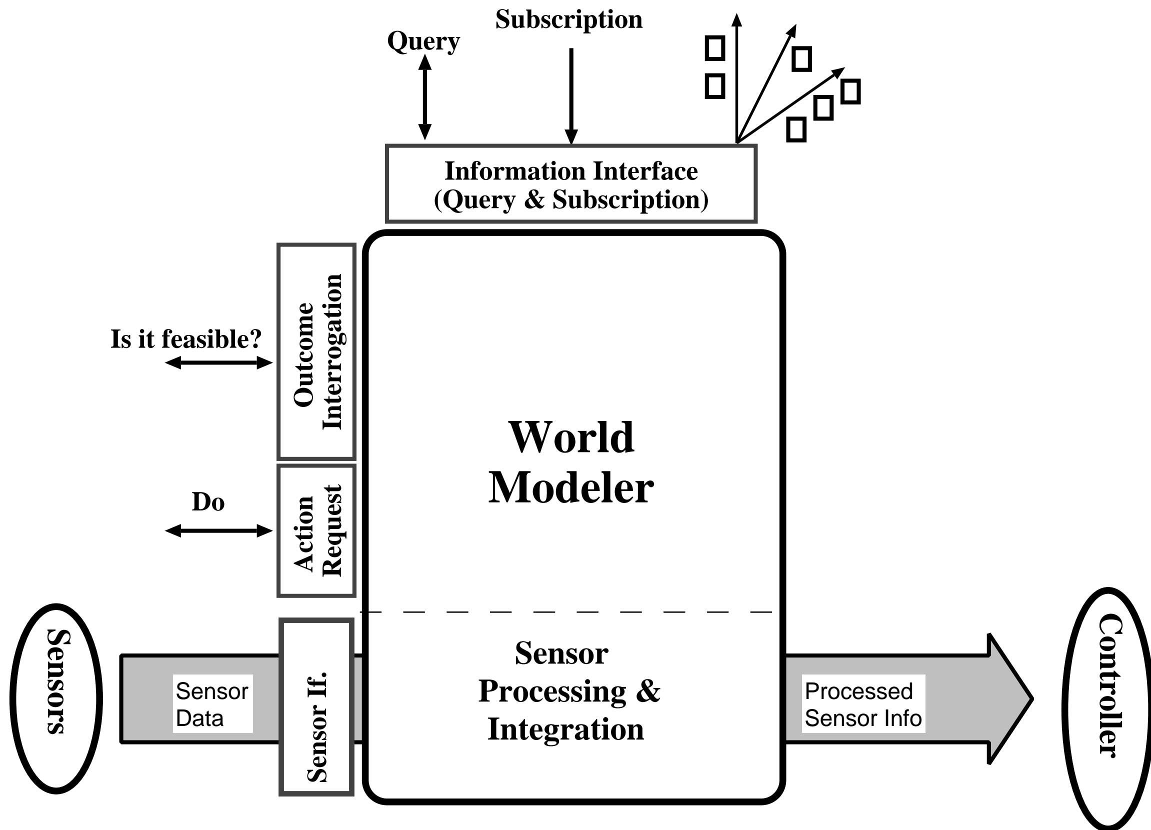


Figure 5.1: **Roles of the World Modeling Subsystem**

The World Modeler serves a variety of purposes: It provides status and state information to any subsystem in a query or subscription mode, it answers questions about the feasibility of different actions, it can be commanded (driven) so that its state changes as a result of system actions, it processes (raw) sensor information (sensor integration/fusion), and transforms this information into a form suitable for use by the remaining subsystems.

mode about the state and properties of different system parts. Second, it must allow interrogation on the *feasibility of different actions*. Third, it must accept *direction* from the strategic-control layer so that the model can be kept up-to-date and consistent with the strategic actions. Fourth, it must embed *sensor-processing* within the dataflow of the control-loop to process and integrate sensor signals into a form usable by the controller. The approach to world modeling taken in this thesis combines a physical description and model of the system with a repository of sensory and state information, where minimal processing is performed, and then only to transform the information into a form which is either more intuitive, generic, or flexible.

Rationale: The motivation for limiting the degree of sensor processing performed by the world modeler stems from the need to keep it versatile. The reason being that the ultimate representation of the world model information within each subsystem (Planner, Controller, GUI) is likely to be quite different. For instance, different planning techniques already use completely different representations of the workspace and objects (discrete bitmaps, quad-trees, spheroids, distance maps). Similarly for the GUI, objects may be rendered in a variety of fashions and details. Hence, the most appropriate object representation format (boundary representations, CSG[4], generalized cylinders) will vary among different graphical interfaces. Therefore, rather than attempting to develop an all-encompassing world-modeling subsystem, which will be necessarily tied to every piece of the system; we advocate the alternative of minimally representing what is known and sensed about the physical system, making this information available, and letting the subsystems further develop specific models for their own use.

In principle, the World Modeler (WM) could have been distributed among different computers. The characteristics of the system advise, however, to keep it centralized and colocated with the control subsystem. In this manner, the control subsystem (which requires the information at high bandwidth and with minimal delay) can access information locally. Consistency problems are also avoided.

5.2 Literature Review: World Modeling for Robotics

The World Model provides a framework where domain-specific knowledge may be used to process sensor information and provide a uniform, simpler interface to the remaining subsystems in need of that information. It also provides a structure for reasoning about actions and consequences (a requirement strategic control and planning). As a result most experimental robotic systems of significant complexity that either incorporate planning, strategic control, or use sophisticated sensors incorporate some form of world modeling. For example the HILARE-2 robot [116], MOBOT-III [63], and AMBLER [85], all incorporate sophisticated world-modeling subsystems.

Solid modeling is often an important aspect of World Modeling systems. This topic has been extensively addressed by mechanical CAD packages [78]. In the field of robotics, several authors have developed dedicated world modeling systems emphasizing the solid modeling aspect. These are mostly useful for graphical rendering, simulation, collision detection and fine motion planning.

[4]Constructive Solid Geometry.

For instance Mirolo and Pagello [111] describe a modeling approach based on CSG[5], with generalized cylinder primitives, that also incorporates surface representation for graphical rendering. Commercially available robotic-programming-and-simulation packages such as SILMA's CimStation [37], DENEB's IGRIP [16] and Tecnomatrix's RoboCAD [197] also incorporate sophisticated solid modeling. These systems, however, are primarily used to simulate and program robots off-line and do not integrate sensor information, nor interact with the robots during operation.

In the context of distributed robotic-system architectures, Oliveira, Camacho and Ramos [47] describe an architecture where different agents cooperate to achieve assembly-type tasks. They identify a separate "model + world descriptor" agent that provides information on the properties and state of the different objects in the workcell. However, this system is used only to aid in the generation of plans and programs. These programs are then down loaded to the workcell for execution. Their system is not used as part of the feedback loop.

World-modeling systems designed specifically for robotic workcells have also been developed. Ravani [143] developed a general-purpose system (RWORLD) for robot programming and simulation that models the manipulators and objects in the workcell. RWORLD uses four primitives to describe the world: rigid objects, devices (mechanisms with more than one DOF, e.g.: robot manipulators, end-effectors, and conveyor belts), transformation frames, and sensors. The description of rigid bodies includes mass, inertia, features, center of mass, and geometrical description (CAD based). Manipulator devices are described in terms of their rigid links, joints, Tool Center Point (TCP), and acceptable commands. Models of the joint include limits (displacement, velocity, torque) and geometric shape. Link models include rigid-body properties. The system can model geometrical, spatial, kinematic, and dynamic aspects. RWORLD also supports aggregation of objects (e.g. when assemblies are made or when the robot grasps an object or an end-effector). RWORLD can compute the kinematics and dynamics of the manipulator devices from the model description. This system appears to have been used only in simulation. A similar approach emphasizing a unix-shell like interactive interface for modeling robotic systems called R-shell has been developed by Vuskovic et al. [191].

The RCS specification. One of the most detailed specifications of the role and architecture of world-modeling subsystems for robotic applications is the one provided by NIST's RCS and NASREM reference models [139, 6, 7, 4]. This model has been used in several applications [5, 98].

[5] Constructive Solid Geometry. An approach that models solids as composed of primitive solid shapes that are combined using boolean (set) operations. Its main drawback is that surface information, necessary for graphical rendering, is hard to extract.

RCS advocates a vertical hierarchy of World-Model "components" where each layer operates in a periodic cycle at a rate that diminishes by roughly an order of magnitude with each higher level in the hierarchy. Each layer is restricted to communicate with the ones above and below in the hierarchy (it also communicates horizontally with peer modules in the planning and control hierarchies). While the strict hierarchy works well for lower levels in which the physical system and sensors are modeled, the higher levels (planners, strategic controllers) are often too unstructured to be restricted into accessing a single world-modeling layer. For instance, a strict hierarchy with no ramifications does not allow alternative implementations of similar functionality (i.e. there is only a single World Modeler at each level). Also, the periodic requirement is not natural for the higher layers, which are naturally event-driven. Nonetheless, there are several general tenets in the RCS model which have been adopted by the system developed in this thesis: (1) need for hypothetical[6] as well as factual queries[7], (2) need for periodic updates, (3) need for a lower layer embedded within the dataflow of the control-loop.

Sensor Integration. The need to integrate information from multiple sensors is always present in systems operating in not-completely-structured environments. Much theoretical work in this area is devoted to the definition of optimality criteria and the mathematical methods that can be used to combine information in an optimal fashion. A variety of such strategies have been presented in [100, 39, 57], including the classical statistical methods (Kalman filtering, Dempster-Shafer, Durrant-Whyte) [43], and others such as Error Functions [186]. However, as noted by Thomopoulos [184], these rigorous signal processing methods, require the existence of a precise mathematical model that describes the generation of the data (including noise and certainty characteristics). As these mathematical models are not always available, more ad-hoc methods such as deciding, averaging, and guiding are also in use [56].

Moreover, the use of formal generic approaches alone is difficult in practice because the sensors may be hard to characterize in the ways required by these methods (e.g. noise, error, or certainty may be difficult to characterize or even define for certain sensors). In addition, some of these properties may depend on the task, not just the sensor. Also, it is often difficult to incorporate in the generic approaches important practical aspects such as: bandwidth of the various sensors (which may be very different), failure characteristics (sensor integration scheme must be fail-safe and degrade gracefully), asynchronous task-dependent information, and domain-specific procedural

[6] Queries about the feasibility and/or outcome of actions.
[7] Queries about the current system state and characteristics.

knowledge. As a result, many systems that operate in the "field" use a combination of general and ad-hoc approaches.

5.3 Architecture of the World Modeler

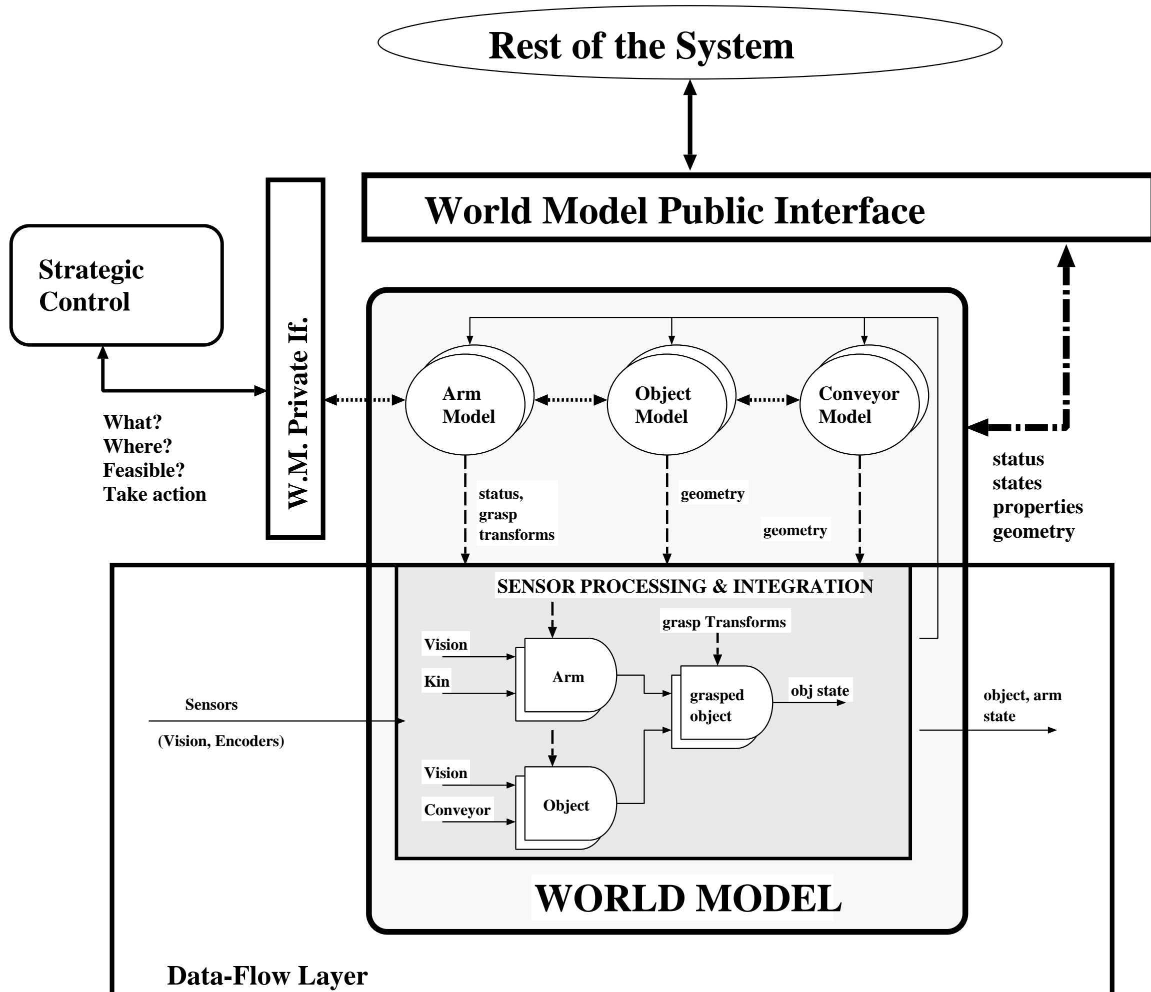


Figure 5.2: **Architecture of the World Modeling Subsystem**

The world-model subsystem contains a dataflow layer and an object-based layer. The dataflow layer processes and integrates information from multiple sensors. The object-based layer contains a software object for each physical entity (manipulator, conveyor, part), keeps status and state information and can respond to queries about the feasibility of different actions.

The implementation of the world modeler must be capable of supporting the roles described in Section 5.1; as well as providing the information required by the external interface described in Chapter 3. The external interface requires the World Modeler to provide three types of information:

Configuration information. This is static information describing the immutable characteristics of the system: Rigid-body shapes and mass properties, arm kinematic parameters, visual fiducials, and base position and orientation for the fixed workcell arms all belong to this category. This information is typically stored in configuration files. The role of the WM is to access and cache this information and make it available through its interface to the remaining subsystems. In this manner, the WM hides the details of configuration-file parsing and description formats.

Status (logical) information. The workcell transitions through a series of discrete states. These states represent distinct logical conditions that are meaningful irrespective of the implementation of the strategic workcell controller (i.e. they are not just "states" of a particular finite-state machine diagram or petri-net that controls the robot at the strategic level). These "abstract" states represent the activity of the workcell to the external world. Hence, the use of the word *status* to avoid confusion. Examples of status information include whether the manipulators are involved in motions, grasping objects, and which objects are being grasped. The WM interface uses a "bit + string vector" to represent the status of each of the robot arms in the workcell. Some of the entries in this vector were listed in the "status" entry of Table 3.2, Chapter 3. The World Modeler maintains this status and provides mechanisms for accessing and modifying if from the strategic-control layer.

Sensor-derived information. Sensor information may be processed for a variety of reasons: pure signal processing (filtering, enhancing etc.), transformation into a more useful format for kinematic/dynamics, and integration/fusion with other sources. The prototype robotic workcell presents examples of all these motivations. These examples a described in detail in Section 5.4.

To fulfill the above requirements, the world modeler subsystem has been designed integrating two layers with complementary computational models: object-based[8] and dataflow as illustrated in Figure 5.2.

[8]This layer is object-based, not object oriented because no inheritance hierarchy has been defined. This has not been a problem because we are using fairly simple models. At the time the world model was developed, there were no object-oriented language (e.g. C++) support for our real-time system. This situation has changed and the WM could benefit from using a true object-oriented approach.

5.3.1 The Object-Based Layer

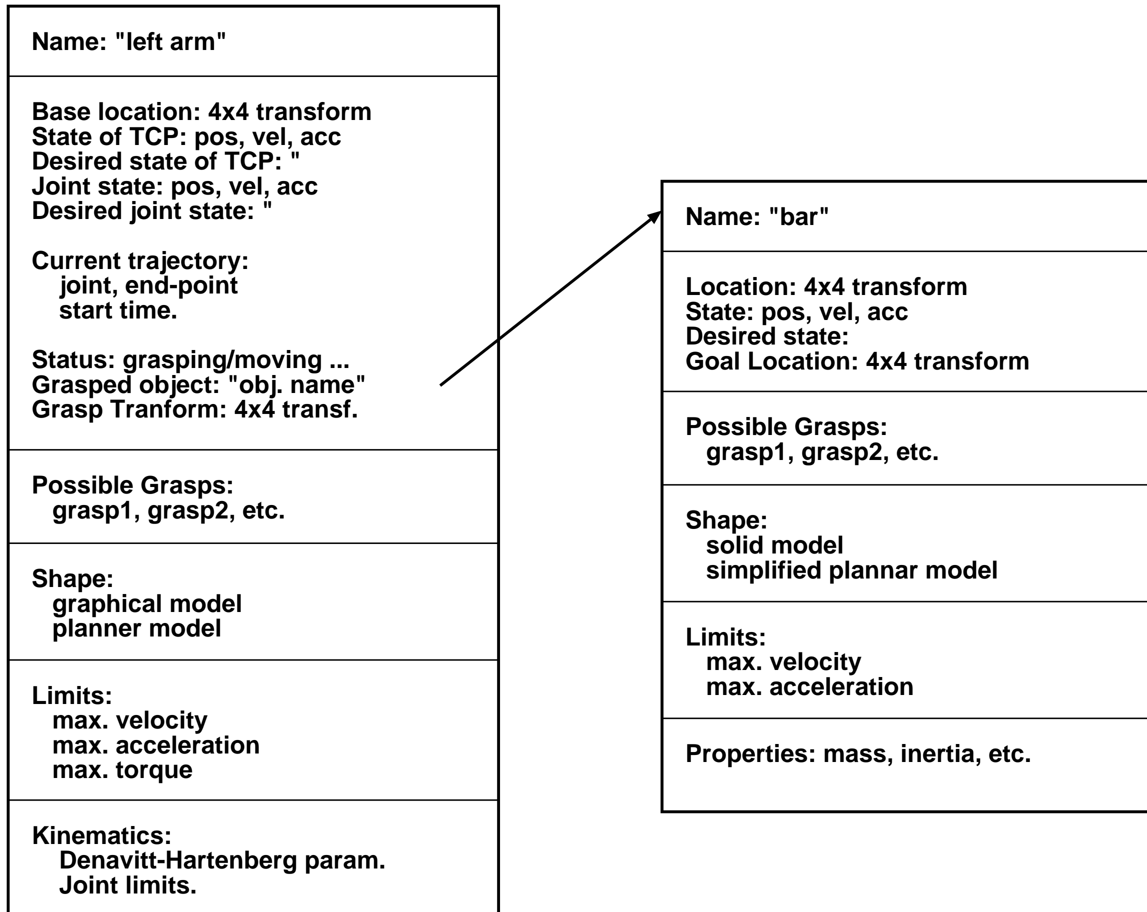


Figure 5.3: **Arms and rigid-objects are represented using software objects.**

This layer is built from three types of (software) objects: (1) arm-objects (manipulators), (2) part-objects (rigid objects that can be manipulated or need to be avoided), and (3) conveyor-objects (which can carry objects on top). There is one software object for each corresponding physical entity. For example, there are two arm-objects (one per arm), eight part-objects, and one conveyor-object. These software objects contain state information about the physical objects they model (see Figure 5.3), and communicate to the external world by exchanging messages through two interfaces: The public world-model interface described in Chapter 3 and a private interface described in Table 5.1. The private interface is used by the strategic-control system (see Section 6.7) to build and modify the model.

The private interface serves several functions:

- Create models for new physical entities.
- Interrogation of the world model regarding the feasibility of actions.
- Direct and inform the world model of actions taken by the strategic control so that the model is kept up to date.
- Query about the state and properties of the different system parts.

5.3.2 The Sensor Processing/Integration Layer

The sensor-processing layer is embedded within the dataflow of the control subsystem. Sensor integration is model-based and it is directed by the object-based layer. The results of the sensor processing/integration are immediately available to the the object-based layer. Details on this layer are presented in Section 5.4.

5.3.3 Characteristics of the Architecture

The architecture of the world modeler is unique in that it presents three radically different interfaces to the remaining system: (1) high-bandwidth dataflow interface, (2) (private) multiplicity-1 master read/write interface, and (3) (public) multiplicity-N, read-only interface. As shown in Figure 5.4, this approach allows the WM subsystem to be used efficiently within a hierarchy and has several advantages over a purely hierarchical approach:

- Interaction can occur with any other subsystem in the hierarchy, not just among subsystems at the same level, or at levels immediately above and below.
- Replication and redundancy can be supported. Using the public read-only interface, multiple subsystems that logically occupy the same position in a hierarchy can coexist. This allows, for example, multiple GUIs or planners to use information from the WM. Moreover, this approach leaves the system open for expansion: new subsystems needing world modeling information do not require a dedicated position and interface within the hierarchy.
- Fewer interfaces have to be defined. Compare this with defining interfaces at each level in the hierarchy.

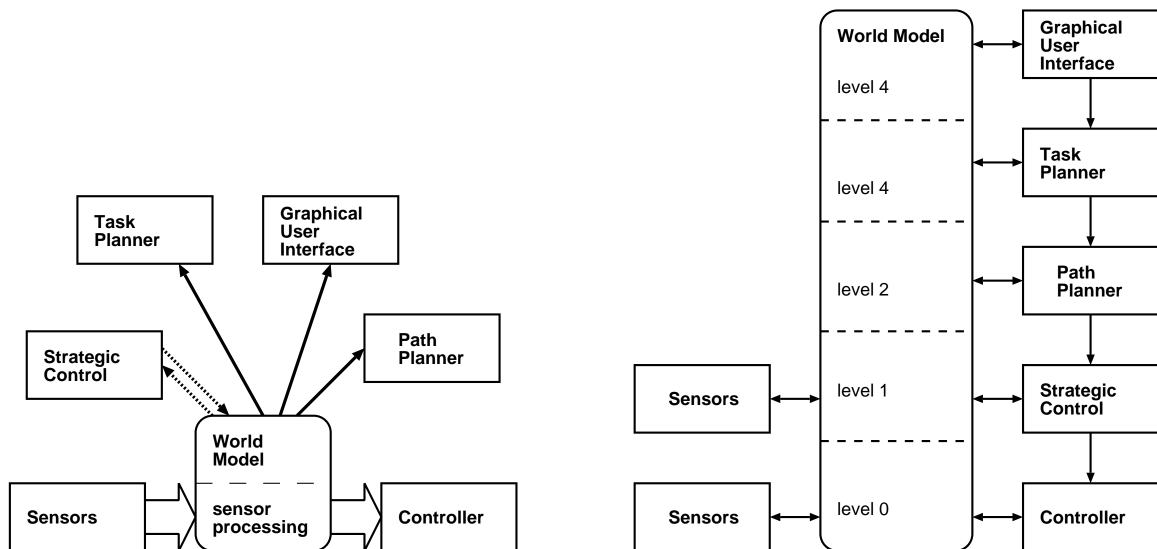


Figure 5.4: **Comparison of WM architecture with purely hierarchical WM**

The architecture of the WM (left) uses a dataflow interface to process sensor data right in the control loop, a private read/write interface, and a public read-only interface. The read/write interface can only be connected to a single subsystem which will be responsible for building and modifying the world modeler. The read-only interface can have any multiplicity and be connected to any number of subsystems. The hierarchical approach (right) uses dedicated interfaces between each pair of subsystems.

- High-bandwidth feedback control can be achieved using the processed sensor data because the processing is sitting within the dataflow of the controller and is driven by the *same* periodic control loop. This same data is made available at lower bandwidths to other subsystems through the other interfaces.

The hierarchical approach, however, offers advantages too: It is more modular and allows each subsystem to have a fully tailored interface. Nonetheless, the fact that the WM interface is anonymous and customizable (using the parametric approach described in Chapter 3); achieves most of the benefits of dedicated interfaces, without the drawback of interface explosion.

A sensor integration architecture must accommodate two functions: **(1) low-level sensor-fusion strategies** (such as Kalman filtering, Shafer-Dempster, Durrant-Whyte) to process and combine the stream of information and **(2) high level rule-based strategies** that can adapt the lower-level strategies to different situations, and provide for error detection and recovery[9]. The above architecture provides both mechanisms: The sensor-processing/fusion layer is embedded within the dataflow, and can be controlled by the object-based layer. The private interface allows

[9]For instance when a sensor malfunctions the low-level sensor-fusion strategy may need to be radically changed. Similarly, when an object is grasped, the information on the arm location can be used to derive the object location.

an external strategic or rule-based subsystem to interact with the World Modeler. Using the private interface, the object properties can be accessed and set, and hypothetical scenarios can be evaluated. In addition, the public interface allows unrestricted read-only access to the information produced by the World Modeler.

Although the architecture of the World Modeler subsystem has been tailored to the needs of the workcell, the following characteristics should be generally applicable to other intelligent robotic systems:

- Need for embedding sensor processing within the dataflow. This is justified because the control loops need high bandwidth access to the sensory information processed by the world model.
- Need for an interface to control the World Modeler. Changing environments, states and error conditions required rule-based strategies to direct the sensor-integration process.
- Use of an object-oriented layer above the dataflow. The use of objects to represent different physical entities (manipulators, parts, conveyors, etc.) allows the model to be easily built from simple pieces, and the system to be expanded to include more entities as need arises.

5.4 Sensor Integration

The terms sensor integration and sensor fusion refer to the use of more than one sensor to infer a particular property of the environment. Sensor *fusion* usually implies some form of analytical combination of the sensor signals (e.g. averaging), whereas sensor *integration* is used when different sensors are performing different tasks (for example guide each other or measure different properties). Integration sometimes is used as a more generic term that includes any mode of combining information from multiple sensors.

The need for sensor integration typically arises because no single sensor can provide all the required information with the necessary accuracy and bandwidth. By using multiple, complementary, sensors ambiguities can be eliminated. By using multiple sensors to measure the same property, error can be reduced.

In the dual-arm manufacturing workcell there are examples of each of the above needs:

- The position of the end effector in the X-Y plane can be obtained from kinematics (using the joint encoders and the rigid-link kinematics of the manipulators) and vision (tracking an LED

mounted on the elbow link). Kinematics has higher bandwidth, less noise, and better accuracy. However, vision is the same sensor used to determine object positions. Therefore, the vision sensor provides better measurements of the relative distance between the manipulator and the objects, and would be the preferred sensor whenever interacting with objects in the workspace (e.g. tracking and picking them from the conveyor).

- The position and orientation of an object can be determined by the vision system. Once the object is grasped, its position and orientation can also be determined from kinematics (i.e. given the grasp transform and the location of the manipulator end-effector). These measurements need to be combined for several reasons: First the vision often loses track of the object when it is grasped (the arm(s) grasping the object obstruct the view from the overhead camera). Second, the vision system is two dimensional and cannot measure the object height. Third, the kinematic measurements are available at higher bandwidth allowing higher performance control. Fourth, whenever possible, vision data must be used because vision provides better relative measurements of distances to other objects and measures object location directly[10]. Sensor-integration is complicated by the fact that it must perform reliably under many circumstances: one or two-handed grasp, with or without visual tracking.
- The location and velocity of an object on a conveyor can be determined by the vision system. However, the object is usually lost from vision during the final track and grasp operation because the arm obstructs the camera view. Additional information on the conveyor displacement and velocity is available from an encoder mounted on the conveyor-motor shaft. This relative data must be combined with the available visual information to allow successful picking of objects from the moving conveyor.

The next three sections describe the approach used to solve these sensor-integration issues.

5.4.1 Integration of Kinematics and Vision for Robot-Arm Position

The objective is to generate an estimate of the arm configuration by merging the information provided by the visual tracking of the elbow-link-mounted LEDs with the information provided by arm-joint encoders. The encoders have the least noise and highest repeatability. From the encoder resolution described in Table 2.3 and the link-lengths (see Table 2.1) it can be seen that the X-Y location of the end-effector as derived from arm kinematics has a resolution of $60\mu m$. This

[10]The kinematic approach relies on a good knowledge of the grasp transforms, and these may slip.

information is also available at the sample rate of the arm controller (100 Hz). For comparison, the vision system provides data at 60 Hz (with a 1/30 sec. delay), has a resolution of about 0.8mm $\times$ 0.2mm[11]. Lens distortion contributes further to vision error. The vision system is calibrated using a 3$^{\rm rd}$ order polynomial fit, but even after calibration, the absolute error may be as large as 4mm in some parts of the workspace (see Appendix A).

Despite the higher resolution and bandwidth of the kinematic measurements, to interact with the workspace objects, capture them from the conveyor, and avoid inter-object collisions, relative measurements are critical. Moreover, vision provides an absolute measurement whereas the encoders have offsets that must be calibrated during each power-on.

Approach. Given the characteristics of the two sensors, what we desire is the resolution, bandwidth, and noise characteristics of the kinematic-based measurement with the absolute values provided by the vision measurement. In other words, the high frequency behaviour of the kinematics and the low frequency behaviour of the vision.

The key observation is that small difference between the endpoint coordinates measured from kinematics and vision can be mapped into an equivalent joint-angle offset. A first order approximation to this mapping is given by the manipulator jacobian: $dx = J(\theta)d\theta$. These small differences (typically 3-4 mm, see Appendix A) will result in small angular offsets (under 0.01 rad) which have negligible effect on the value of the Jacobian, Mass Matrix, or any other quantity related to the manipulator dynamics. Therefore, we can–*without side-effects*–bring the kinematic and vision measurements to full agreement by adding corrective offsets to the encoder-measured joint angles. These offsets may be adaptively changed[12] as the manipulator moves though the workspace using the equation $d\theta = J^{-1}(\theta)dx$. Note that this approach also takes care of the initial offsets due to the power-up configuration of the arms (the encoders measure relative angles from the initial power-up angles)[13]. The architecture used to implement this approach is shown in Figure 5.5.

Performance. The performance of the sensor integration scheme can be seen in Figure 5.6. In this figure, the arm has been commanded on a straight-line slew and the vision and kinematic estimates for the x-coordinate of the endpoint are compared. Initially (steady state) the vision and kinematic

[11] The resolution of the vision system is 0.1 pixels along the X axis and 0.02 pixels along the Y axis [154]. In interlaced mode, a pixel corresponds to an area of 8mm $\times$ 9mm.

[12] The vision-based correction is is first run through a low-pass butterworth filter with 3 Hz bandwidth so it only affects the low-frequency value of the position signal.

[13] This capability was an important motivation for adapting the angular offsets instead of adding corrective terms in cartesian space directly.

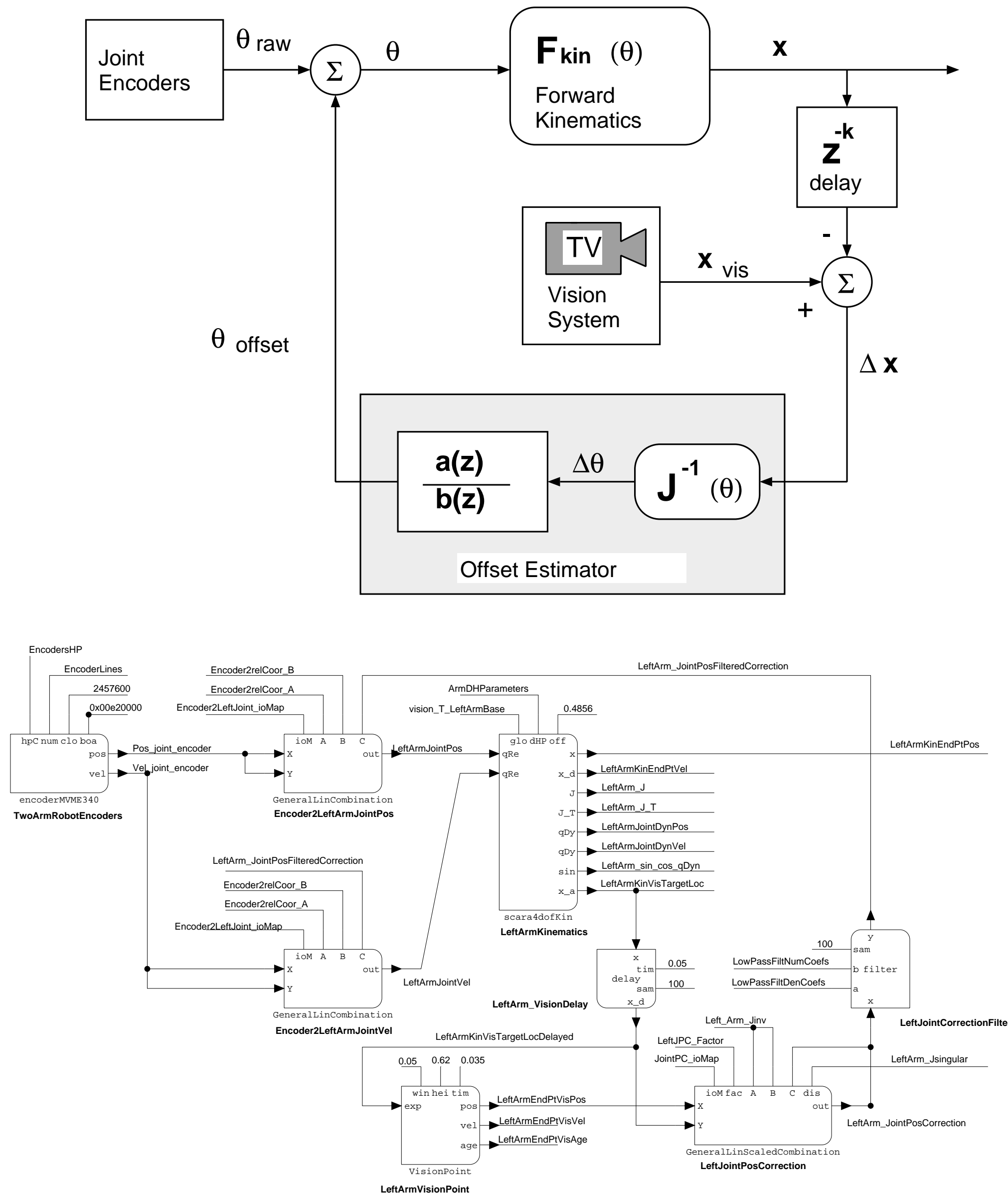


Figure 5.5: **On-Line estimator of initial angular offsets**

The (shoulder and elbow) joint angles produced by incremental joint encoders need to be combined with the initial angular offsets that correspond to the arm power-up location. In addition, these offsets are used to compensate for the (configuration-dependent, small) differences between kinematics and vision coordinates. The offset estimator maps the Cartesian error between kinematic and vision coordinates through the inverse Jacobian to obtain a corresponding joint-space error. This error is run through a filter (essentially an integrator followed by a low pass filter) to produce an estimate of the angular offsets. This process is illustrated in the top diagram. The diagram on the bottom shows the ControlShell implementation using generic software components.

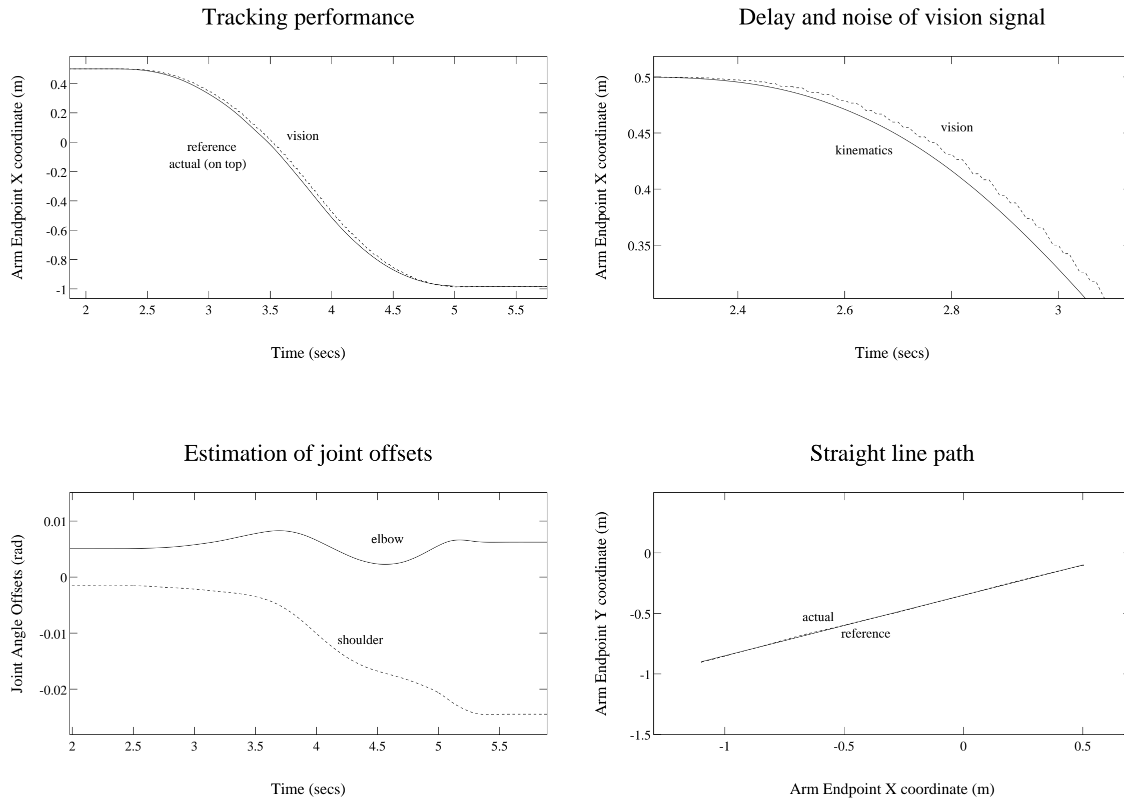


Figure 5.6: **Performance of On-Line angular-offset estimator**

The top-left plot shows both kinematic and vision coordinates during a straight-line slew (bottom-right plot). The vision signal is not suitable for high-performance control: it is quite noisy and has a delay of about 50 ms due to the processing overhead (detail in top-right plot). The on-line estimator adapts the values of the shoulder and elbow angular offsets (bottom-left plot) so that at the end of the trajectory the kinematic and vision coordinates match again.

measurements match (the adaptation has made it happen). During the slew, the vision measurements are delayed and noisy; the adaptation is active but too slow (due to the low-pass filter) to make a difference until the arm stops at a new location. At the new location, the offsets are adapted until the vision and kinematic measurements match again. The speed of the adaptation can be seen in the step response of Figure 5.7. This plot also illustrates the adaptation to the power-up configuration which has approximately 1 second time constant.

Model construction

nature	*action*
arm	Create and add a model for a manipulator arm. Provide its name. The configuration files are read to create the model, read the kinematics, DH parameters, base location in the workspace etc.
object	Create and add a model for an object (a manipulable part or a pure obstacle). Looks up the configuration-database to get all the information on the object (geometry, grasps, visual fiducials etc.).
conveyor	Create and add a conveyor model. Provide its name. The configuration database is used to obtain the conveyor's shape, height, and visual-fiducials.

Model feasibility inquiries

subject	*checks*
extrapolation	Extrapolate future object/arm state.
reachability	Can the arm reach a given position?
graspability	Would the arm succeed in grasping an object?
safety	Check an arm trajectory for collisions.
release height	What would be the height of the object if it were released now?

Model control

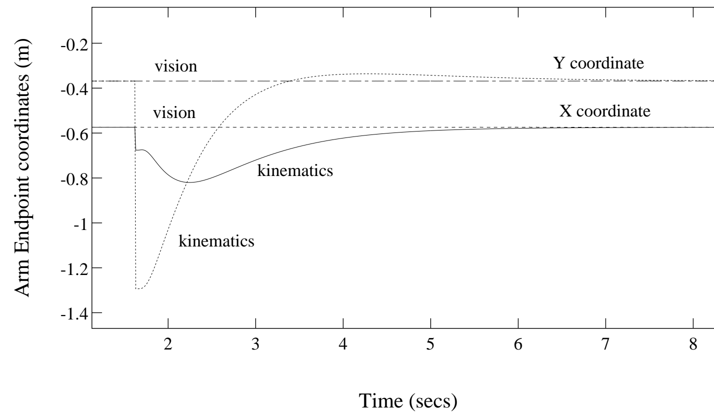
subject	*control action*
arm status	Modify grasping, catching, moving, tracking, picking, or lifting status
misc. info	Set arm/object trajectories and start time
kinematics	Transform via points using Forward Kinematics or Inverse Kinematics

Model query

topic	*available information*
status	Get status on motion, picking. or grasping. Get object being grasped.
distances	Compute distance from arm to a location relative to the current object position.
goals	Get object goal destination, get current trajectory and time when it started.

Table 5.1: **World-Model private interface**

In addition to the **public** *world model interface described in Table 3.2, the* **private** *world-model interface provides a mechanism for the strategic layer to incrementally build the system model, access and modify the world model status, request services, and inquire about the feasibility of prospective actions.*



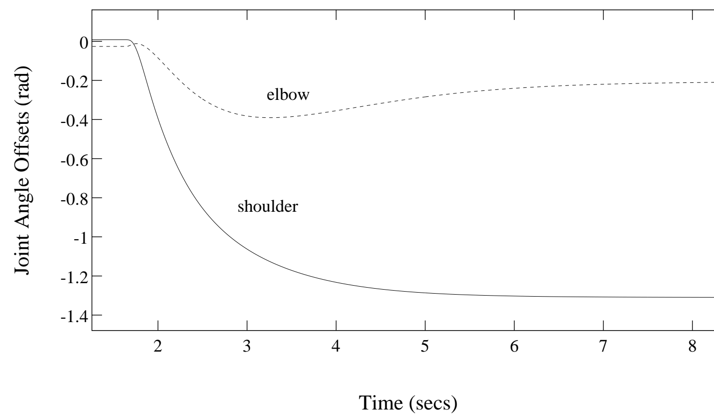


Figure 5.7: **Adaptation to power-up configuration**

When the incremental encoders are powered-up, it becomes necessary to identify the angular offsets that correspond to the location of the arms. In this experiment the arms are held at an unusual location when the encoders are powered-up. As a result, the initial kinematic location is very different from the (true) vision location. As the angular offsets are identified (bottom plot) the kinematic coordinates converge to the vision coordinates (top plot).

5.4.2 Robot State and Vision for Grasped-Object Position

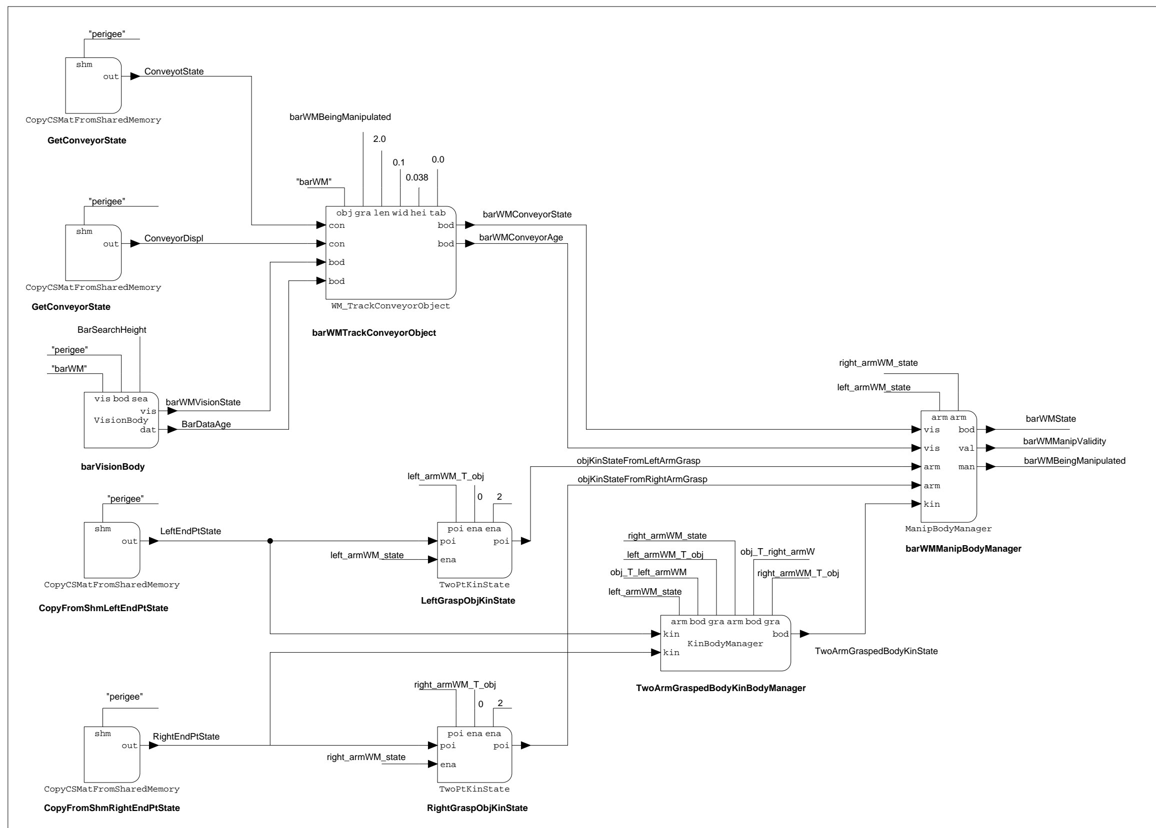


Figure 5.8: **Sensor Fusion for the position of grasped objects**

This dataflow diagram details the sensor fusion scheme. The grasp transforms are measured using the vision system after the grasp has been consolidated. If the object is not tracked by the vision, the transform is set to the desired grasp transform. The `KinBodyManager` *component uses rigid body kinematics from each existing grasp to derive the kinematic estimate of the object position (an average of all the grasps is used). The* `ManipModymanager` *component merges the information with that of the vision.*

The overhead vision system used to track objects cannot measure object height. In fact, object height must be provided to the vision tracking algorithm so that the objects can be accurately identified. In addition, the vision system often loses track of the objects once grasped because the arm obstructs the LEDs from the camera. If the grasp transforms are known, they can be used in combination with the known arm locations to provide a kinematic estimate of the object location. This kinematic estimate can be merged with the vision information to provide a measurement that incorporates object height and is robust to loss of visual tracking. The sensor integration scheme fulfills two concurrent roles: First it *guides* the vision sensor (by providing the height of the object).

Second it *replaces* the vision sensor whenever it loses track of the object. Notice that it would also be possible to *enhance* the estimate by merging its measurements with a less noisy and higher bandwidth estimate from the kinematics. The current implementation does not exploit this last fact. Instead, the integration of kinematic and vision estimates uses a simple heuristic: If the object is grasped by both arms, the kinematic measurement is used. In the case of a single-arm grasp, the vision is used if available (for all but height), else kinematics are used. In any case, the kinematically-derived object height is used to guide the vision system in its tracking.

This strategy does have some failure modes, but the failure is graceful. For example if the grasp slips, the corresponding change in the grasp transforms results in an error in both the object and the reference-arm positions. However, the arms are under impedance control and this error will only result in certain forces/tensions being applied through the object[14]. The grasp may also be completely lost from one (or both) manipulators. In this situation, the controller still commands each manipulator to follow the trajectory as if the object was still grasped and a corrective action is only taken when the failure is discovered by the system (e.g. by seeing the object at a different location). These examples illustrate the need for these fairly low-level sensor-integration schemes to be combined with higher-level strategies (such as the strategy which watches for an object whose vision coordinates are far from the kinematic coordinates to determine whether the grasp has failed) to achieve robust operation.

5.4.3 Conveyor Displacement and Vision for Object Position

For the workcell to operate properly, it must accurately predict and/or sense the motion of the objects on the conveyor. As previously mentioned, the object is often lost during the capture maneuver because the arm obstructs the camera view of the object's LEDs. This problem is solved by adding LEDs to the conveyor (so its location can be determined) and an incremental encoder to the motor driving the conveyor belt (so that belt displacement can be measured)[15]. The World Modeler uses a dedicated sensor-integration software component for each conveyor/object pair. The position and size of the conveyor and various objects is used by the component to determine whether a specific object is on top of that conveyor. If an object is on the conveyor, vision information is used whenever available, otherwise, the object's position and velocity are estimated from the last vision update and the conveyor measurements. This estimation uses the measured conveyor orientation (determined by the vision system) and the relative displacement of the conveyor belt from the time

[14]The magnitude of these forces will depend on the arm impedances estimation errors.

[15]The procedure described here can be applied to any number of conveyors present in the workcell.

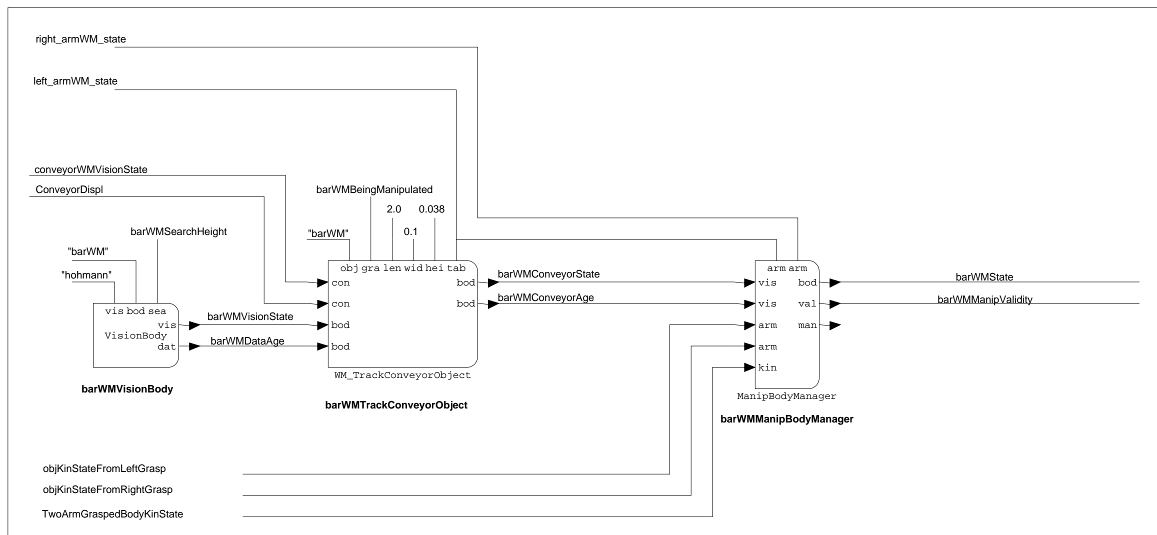


Figure 5.9: **Sensor integration scheme for objects on a conveyor**

A dedicated component exists for each object/conveyor pair. The component uses the vision-sensed position of the conveyor and object to determine whether the object is on the conveyor by modelling the area the conveyor occupies and figuring out whether the center of gravity of the object rests on this area. The vision measurements are used whenever available. When vision tracking is lost, the position of the object is extrapolated from the last visible position using the relative conveyor displacement and orientation.

visual tracking was lost. Object velocity is simply deduced from conveyor speed and orientation. This scheme is illustrated in Figure 5.9.

Figures 5.10 and 5.11 illustrate the performance of this algorithm to track objects on a conveyor. In all cases the accumulated error is smaller than the accuracy required for a successful capture (about 1 cm).

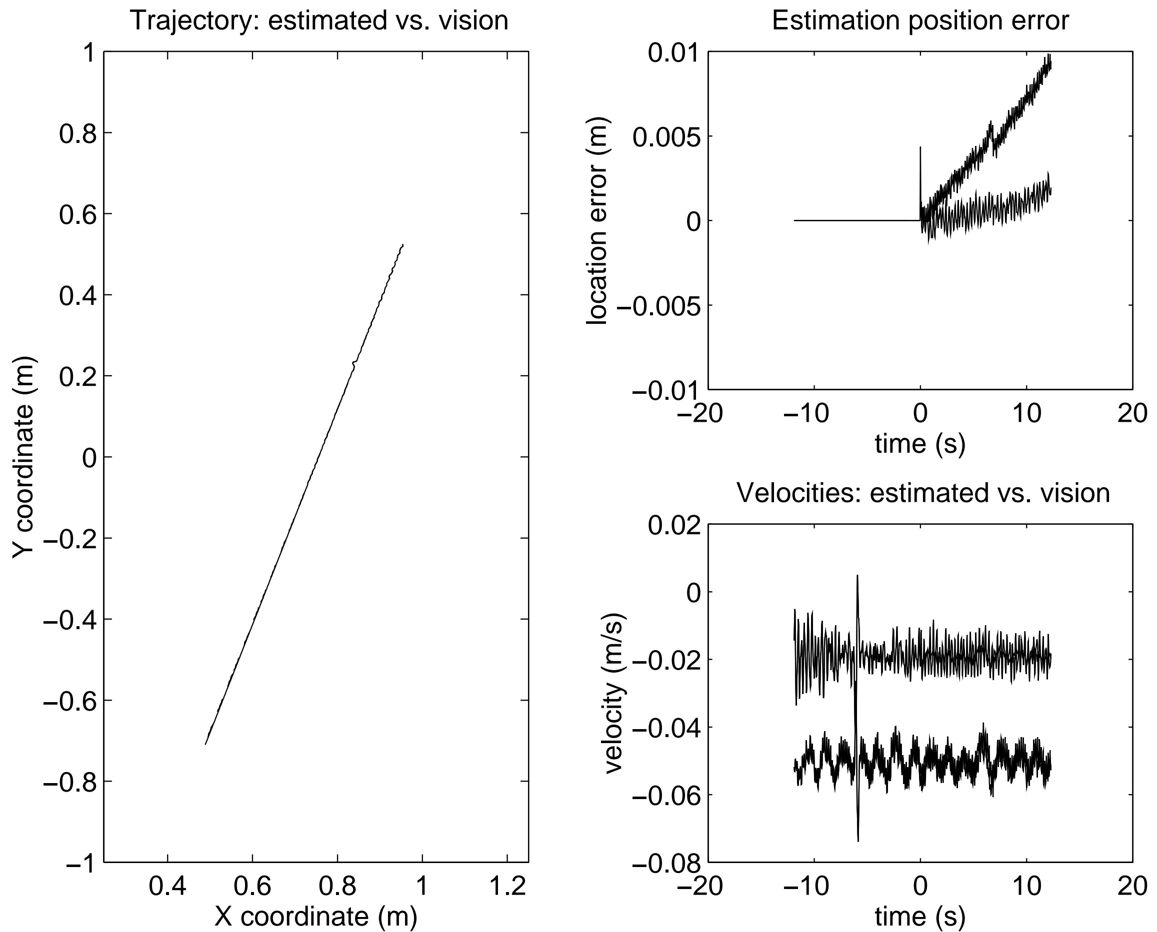


Figure 5.10: **Tracking performance of objects on the conveyor at 5 cm/sec**

The above plots illustrate the performance of the sensor-integration algorithm for the nominal conveyor speed of 5 cm/s. At time t=0, the sensor-integration software was misled into believing that the vision had lost track of the object. In this manner, the difference between the actual vision state and the one derived from the conveyor sensors can be compared. The position discrepancy is on the order of 1 cm even after 0.5 m travel without vision information, this is the limit on the gripper tolerance for a successful capture. The plots on the right illustrate positions and velocities for the X and Y coordinates.

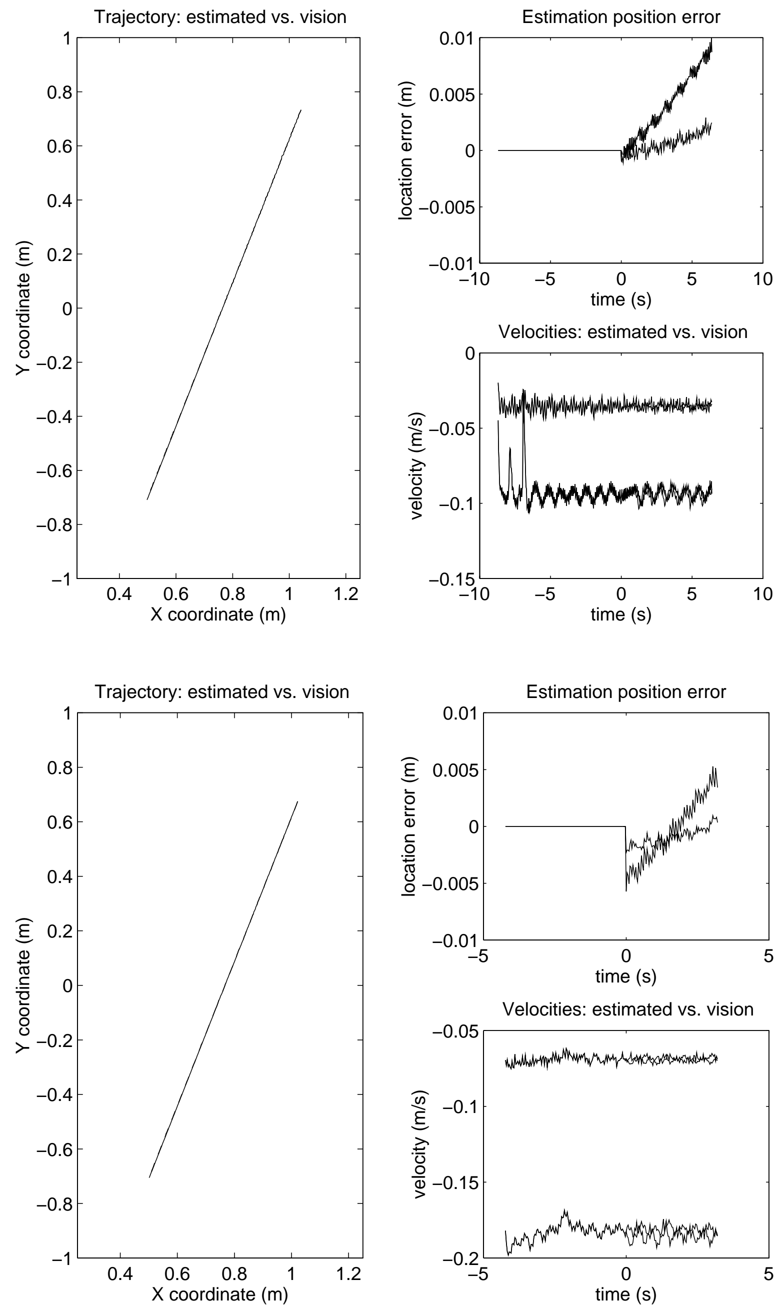


Figure 5.11: **Tracking performance of objects on the conveyor at 10 and 20 cm/s**

The above plots illustrate the performance of the sensor-integration algorithm for faster conveyor speeds: 10 cm/s and 20 cm/s. The approach used is the same explained in Figure 5.10: the sensor-integration algorithm believes the vision has lost track of the object at time t=0. As in Figure 5.10, the position discrepancy is on the order of 1 cm after 0.5 m displacement. This is mostly due to the error in the (vision-measured) orientation of the conveyor.

5.5 Summary

This chapter has described the World Model subsystem. The world model plays a variety of active and passive roles: **(a) repository** of information, **(b) consultant** regarding feasibility of different actions, **(c) disseminator** of information in response to subscriptions and **(d) transformer** of sensor information (conditioning, sensor integration/fusion). The architecture adopted is composed of an object-layer and a dataflow layer. The dataflow layer is responsible for sensor processing and is connected directly to the sensor and control subsystem. The object-based layer provides the remaining functionality through the use of software objects that correspond to the different entities (manipulators, conveyors, parts) in the workcell.

The sensor-integration layer integrates vision and kinematic data for the manipulators and parts. It also merges the conveyor displacement with information from the vision system to allow robust part acquisition despite the potential loss of visual tracking. Experimental results show excellent performance.

This approach is modular, generic, and scalable. Additional manipulators, parts and conveyors can be added with ease. Incorporating a dataflow layer within the world model brings the benefits of model-based sensor integration directly into the control loop. The feasibility checks and action interfaces allow direct interaction with a strategic-control layer that can inform the model of changes in the activity of the system.

Chapter 6

Hierarchical Control System

This chapter describes the hierarchical control subsystem (controller). The ultimate objective of this research was not to advance the edge of performance of the control system but rather to develop a complete balanced system so that overall performance requirements of the experiment were met. While it is important to exercise rigorous control methodology when developing the control system, having a very-high-performance controller would be of little benefit unless the rest of the system were able to exploit the extra performance.

6.1 Control System for the Manufacturing Workcell

The control subsystem receives strategic-level commands through the Robot Interface described in Chapter 3, and controls the manipulators in response to those commands. To this end, the control subsystem must not only provide stable control of the manipulators, but must also interact with the world modeler and use a certain level of "intelligence" to transition between control modes, perform all the autonomous steps and coordination required to pick up objects from the moving conveyor, and detect/report/recover from failure conditions.

Safety is commonly a concern in robotic systems. In the Manufacturing Workcell, the size, inertia and peak velocities (on the order of 1 m/s) of the manipulators make safety mechanisms imperative to protect both humans and the equipment and its environment. Passive mechanisms such as an enclosed workspace and the ubiquitous "kill switch" can be effective to protect the human operators when used consistently. However, ensuring the safety of the manipulators themselves, even in the presence of failures in the planner/controller, requires special provisions in the design of the control subsystem.

6.2 Literature Review: Hierarchical Robotic Control Systems

A number of successful experiments at the Aerospace Robotics Laboratory have used a hierarchical approach to control dual-arm robotic systems. This approach was pioneered in the laboratory by Schneider [154, 157]. Several extensions were made in subsequent experiments: Ullman [189] modified it for free-flying robotic systems, Meer [105, 106] extended it to handle manipulation of flexible objects, and Pfeffer [131, 132] added an extra lowest-level layer to handle manipulators with joint-flexibility and non-ideal actuators. The lowest levels of the dual-arm workcell control system are heavily based on Pfeffer's methodology.

This chapter spans a wide variety of topics: low-level control issues, cooperative manipulation, and strategic programming of robot behavior/actions. Each of these issues has been the focus of substantial research in the robotics community. A detailed review of this vast field is beyond the scope of this thesis. Rather, the summary provided here will be restricted to reviewing related hierarchical approaches. Specific references on each one of the layers are included within the corresponding sections of this chapter.

Hierarchical control systems for robotics have been proposed by many authors [194, 4, 154, 24]. Some of the most detailed are RCS [139, 6, 7] and NASREM [4, 5] from NIST. These approaches have already been discussed in 3.1.

Classical robot-control hierarchies are programmed using a specialized robot language [22]. Sequential languages are not well suited to control highly asynchronous event-driven systems [157]. While some languages provide asynchronous extensions such as LM's "guarded-commands" [86], many authors have opted for the use of more declarative or synchronous languages [58] such as Esterel [172], or constructs such as Statecharts [59] and finite-state machines [157, 75, 141, 161]. For instance, the ROTEX experiment [64] used finite-state machine transitions to control the different stages of a remote ORU replacement.

Habib et al. [55] present a control system for the Yamabico-9 mobile robot composed of strategic and motor layers. The strategic layer is programmed using ROBOL-0, which is essentially a language in which to program a FSM whose states ("action modes" in their terminology) represent modes in the operation of the system. State transitions occur when pre-programmed conditions (which depend in external stimuli and sensed values) are met. Transitions trigger execution of a corresponding transition function. This model is similar to the one provided by ControlShell.4.x except ControlShell's allows the next-state to depend on the return values of the transition functions.

The KAMRO workcell [1] uses petri-nets to sequence through the stages of pre-computed assembly plans. Coste-Manière, Espiau, Simon and Rutten [35, 36] propose the use of a "task-level" language containing a rich set of event-handling constructs which allow the program to wait for conditions to occur, loop while certain conditions hold, or even watch in parallel for sets of conditions and bind reactions to them. The language provides synchronization mechanisms between parallel applications. Programs in this "task-level" language are compiled into Esterel and ultimately executed as FSM programs. On the other hand, ControlShell.4.x's Finite-State Machines are much higher level than those generated by Esterel, and programming them directly with the aid of available graphical tools can be as simple as (and often more intuitive than) writing "task-level" programs.

6.3 Control System Hierarchy

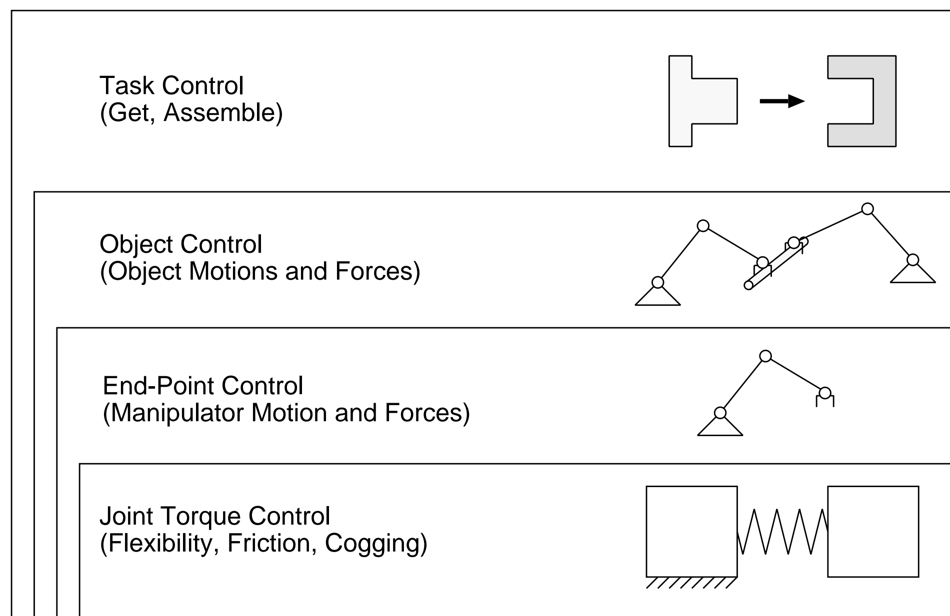


Figure 6.1: **Hierarchical Control Subsystem**

We use a four layer hierarchy to control the two-armed robot. At the lower joint *level, we use joint-torque sensors to compensate for the non-idealities of the motor (cogging, non-linearity) and the joint dynamics induced by the joint flexibility. The next layer, the* arm *level control, can now assume ideal actuation (i.e. that the motors deliver the desired torque to the link itself) and use a Computed Torque approach to compensate for the non-linear arm dynamics. The third* object *layer is concerned with object behavior and assumes that the arms are virtual multi-dimensional actuators that apply forces and torques to the object. The top layer implements elementary tasks such as object acquisition and release, insertions etc.*

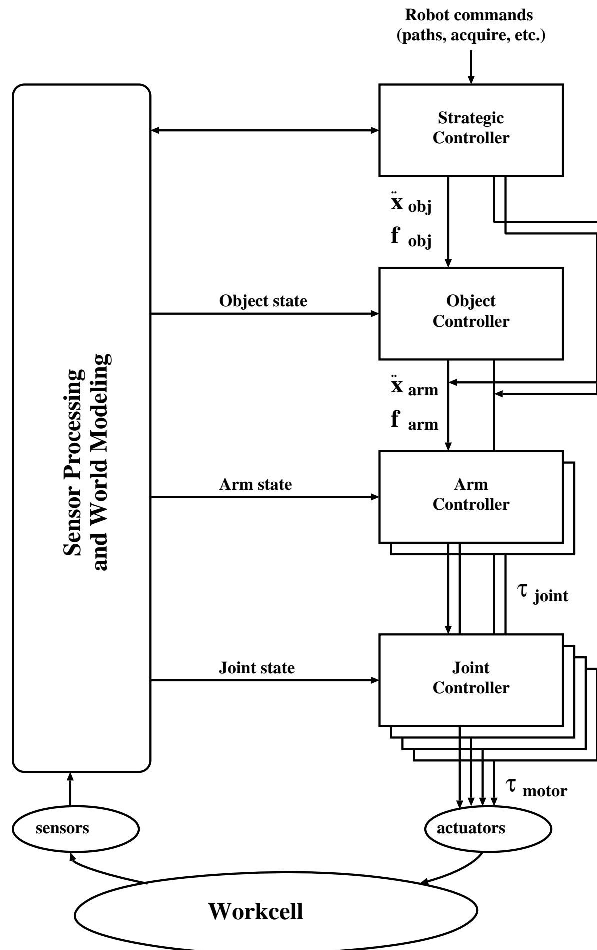


Figure 6.2: **Hierarchical Control Dataflow**

The strategic level processes robot commands and generates object and arm trajectories. If an object is being manipulated, the object controller generates arm references (positions, velocities, accelerations, and applied forces) based on the object reference trajectory, or else the arm trajectories are used as references for the arm controller. The arm controller uses these references to generate reference joint torques. The joint-controller commands motor torques so that the reference torque is delivered to each link. Each one of these controllers closes a feedback loop.

The control system for the two-arm robotic workcell uses the four-layer hierarchy shown in Figure 6.1. The idea behind the layered approach is to simplify the control design by decomposing it into functional layers. Each layer closes a control loop transforming the response of the system from the one presented by the layer below to the one provided to the layer above. These layers and their interfaces are selected so that each layer presents a more idealized system to the layer above, transforming the actual system dynamics into some more-ideal pre-selected set of dynamics. This

approach lends itself naturally to a multi-processor implementation. In the workcell experiment, each layer was executed on a different processor communicating through shared memory. The closed-loop bandwidth of the innermost (joint-control) loop is about 40Hz (Section 6.4), the next (arm-control) loop has a closed-loop bandwidth of 3.3 Hz while the outer (object-control) loop has a closed-loop bandwidth of 1.1 Hz. The theoretical basis for such layered frequency-separation approach is based on the application of singular perturbation theory to manipulator dynamics [83, 81]. Figure 6.2 illustrates the dataflow of this control hierarchy, and Table 6.1 summarizes the interfaces between the different layers.

Aside from simplifying the control design, the layered approach has the advantage of permitting individual layers to be modified without affecting the remaining layers (provided the interface does not change). Therefore, individual layers can be changed to implement different controllers, control policies, or even compensate for different plant/object dynamics. For example, if the joint flexibility in the workcell arms changed or was eliminated, only the joint-control layer would be affected. Similarly, if an object had some dynamics (e.g. a flexible or deformable object), only the object layer would be affected [106].

.

6.4 Joint-Control Layer

The purpose of the Joint-Control layer is to compensate for joint flexibility and non-ideal motor characteristics. To this end, the manipulators were designed and built with integral joint-torque sensors on both shoulder and elbow joints [131]. The concept of using fast joint-torque loops to modify joint dynamics so that they exhibit (viewed from the layers above) near-ideal torque response, has been understood for quite some time [96, 129, 133]. In fact, some commercial manipulators, such as the Robotics Research arm, the Zebra Zero arm , and the newer Barrett Arm incorporate such joint-torque loops.

In order to close a local torque loop around each joint, it is first necessary to identify the motor-drivetrain-joint subsystem that exists between the commanded torque input and the measured (delivered) torque output. This is a non-linear system (primarily due to motor friction and cogging). During his research, Pfeffer explored the use of cogging compensation to linearize the system prior to identification and control-design, and determined this extra step was unnecessary because once the linear torque loop was closed, the cogging compensation did little to add to its performance.

Layer	*Responsibility*	*Interface to layer below*
Strategic Control	Implement strategic commands: follow trajectories, pick up moving objects, move objects	Robot interface described in Chapter 3
Object Control	Command arms so that the object behaves like a virtual impedance on each of its DOF	Object forces and accelerations
Arm Control	Change arm end-effector dynamics so that it behaves like a virtual Cartesian-space impedance	Endpoint force and acceleration
Joint-Torque	Compensate for joint flexibility and non-ideal motor characteristics (cogging, friction). Make actuators appear to be ideal torque sources	Torques applied to each rigid link at the joint

Table 6.1: **Summary of control layers and their interfaces.**

The control system for the manufacturing workcell consists of four layers. Each layer performs a well defined function and modifies the apparent system dynamics to simplify the task of the layer above.

Although this result may not hold in general, it was certainly the case for the Manufacturing Workcell.

Identification of each of the four motor-drivetrain-joint subsystems (shoulder and elbow joints on each of the two arms) was performed by collecting frequency-response data with the aid of a Schlumberger Solartron model 1254 frequency response analyzer. This instrument generates a periodic analog signal[1] to drive the system, and measures the steady-state (amplitude and phase) response of the system on as many as four analog signal outputs. The analyzer measures the amplitude and phase at the fundamental frequency (unless instructed otherwise). The frequency responses were taken with a real-time computer running a weak position-control loop on both joints in order to keep the manipulator close to its nominal position.

In principle, the behavior of the flexible subsystem could change drastically as a function of arm configuration and payload. However, Pfeffer designed the actuators and (low) gear ratios in such

[1] The Solartron is capable of generating periodic signals of several forms. For the identification, sinusoidal inputs were used exclusively.

way that the effective link inertias are much greater than the actuator inertias, effectively decoupling each actuator-joint subsystem. Pfeffer's derivation is presented in [131], Chapter 4.

For the identification of the shoulder subsystem, the elbow was regulated at 90° relative shoulder angle (position of maximum decoupling). For the identification of the elbow subsystem, the shoulder was mechanically kept fixed. Since the system is not linear, the frequency response varies depending on the amplitude of the driving sinusoid. Only frequency responses where the peak-to peak value is under 4.0 Nm are of interest, because this torque is never exceeded during operation (this is the limit that was chosen for safety considerations). Frequency sweeps with peak torques of 0.25 Nm, 0.5 Nm, 1.0 Nm, 2.0 Nm and 4.0 Nm were performed. The results were very similar for all except the smaller (0.25 Nm and 0.5 Nm) peak values. (This was to be expected since the cogging torque for the shoulder and elbow motors has been determined to be on the order of 0.5 Nm [131]). The system identification was performed on the frequency response to the 1.0 Nm peak-torque excitation, which appeared most representative of the joint subsystem response over the intended torque range.

Results of the identification and model fitting are shown in Figure 6.3 for the right shoulder subsystem. A second-order model was fitted to the frequency response. The parameter-fitting technique used was developed by Pfeffer [130]. This fit identifies the poles and zeros of the second-order model, $\mathbf{p}$, that minimize a cost function, $J(\mathbf{p})$, representing the mismatch between the magnitude of the experimental frequency response $|H_e(w)|$ and that of the model $|H_m(w, \mathbf{p})|$.

$$J(\mathbf{p}) = \sum_{k}^{\text{frequencies}} \left[\log(|H_e(w_k)|) - \log(|H_m(w_k, \mathbf{p})|)\right]^2$$

The use of only the magnitude of the frequency response, and the selection of this particular cost-function, have been justified by previous researchers [168, 130]. Once a model has been fit to the response of each joint subsystem, it is transformed into state-space form, discretized (using the zero-order-hold approximation) for the controller sampling rate (360 Hz), and transformed to balanced-realization form. These transformations produce a model of each subsystem in the discrete state-space form:

$$\begin{aligned} \mathbf{x}[k+1] &= \mathbf{\Phi}\,\mathbf{x}[k] + \mathbf{\Gamma}\,u[k] \\ y[k] &= \mathbf{H}\,\mathbf{x}[k] \end{aligned}$$

A predictive estimator and a state-space controller were designed for each subsystem using the LQE/LQG optimal estimator/controller formalism [50]. This approach results in the following control equations:

$$\bar{\mathbf{x}}[k+1] = \mathbf{\Phi}\,\bar{\mathbf{x}}[k] + \mathbf{\Gamma}\,u[k] + \mathbf{L_p}\,(y[k] - \mathbf{H}\,[k]\bar{\mathbf{x}}[k]) \quad (6.1)$$

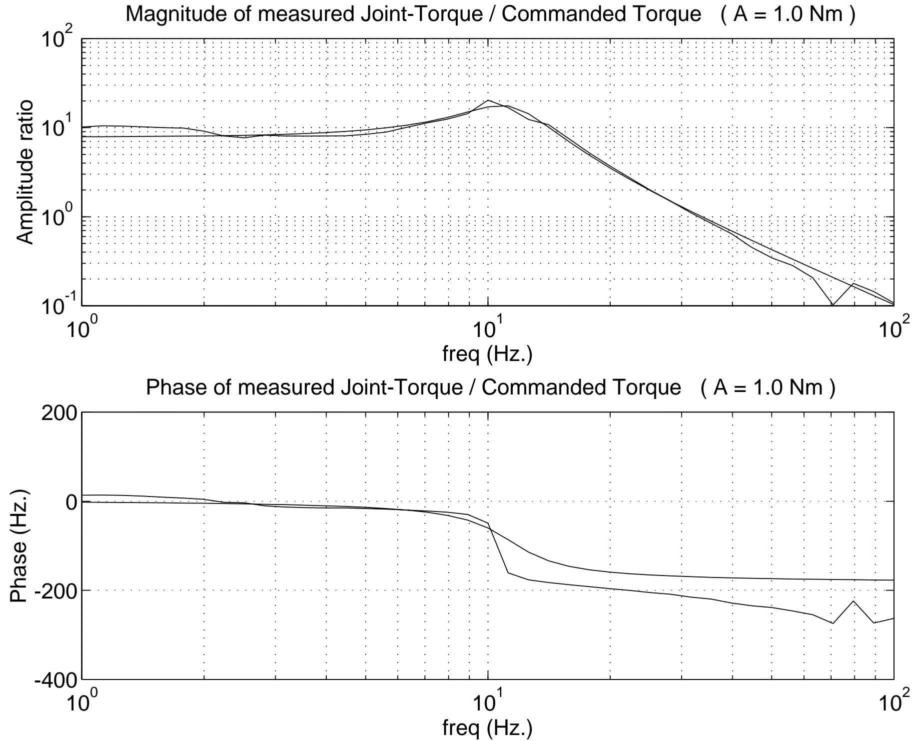


Figure 6.3: **Experimental frequency response for the motor-drive-train-link plant of the right arm.**

The frequency response was obtained using sine sweeps. A second-order model has been fit to the amplitude data. This model has proved to be an adequate approximation to the true plant dynamics in the frequency range of interest. The second-order model has a DC gain of 7.85, a resonant frequency of 11.4 Hz, and a damping ratio of 0.226

$$u[k+1] \quad = \quad \mathbf{K}\left(\mathbf{N_x}\, u_r - \bar{\mathbf{x}}[k+1]\right) + \mathbf{N_u}\, u_r \tag{6.2}$$

The estimator and control parameters are summarized in Appendix B (tables B.2 and B.1). Figure 6.4 shows the Z-plane location of the the poles introduced by the controller and estimator as well as the closed-loop system roots for the right-shoulder subsystem. The experimental step-responses shown in Figure 6.6 confirm the expected rise-time, although the system has considerably more overshoot. This control-system is directly implemented as a set of ControlShell.4.x components, as illustrated in Figure 6.5.

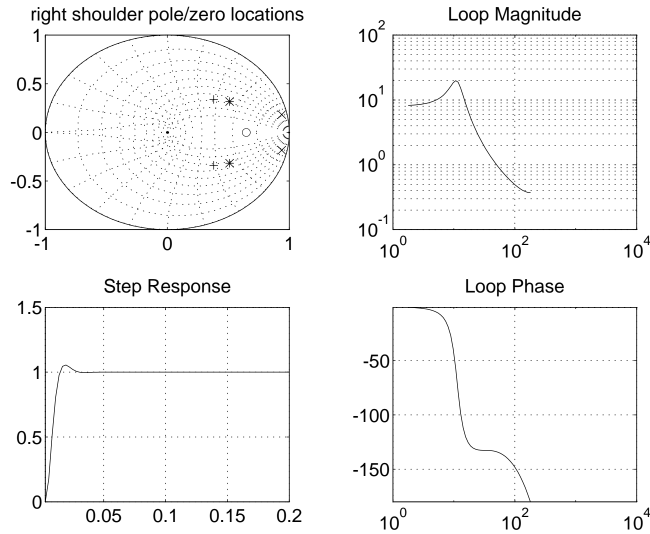


Figure 6.4: **Closed-loop roots for the motor-drive-train-link plant and controller of the right shoulder.**

The controller was designed using the LQE/LQG state-space technique for a sample rate of 360 Hz. The location of the closed-loop poles represent an adequate compromise between performance and robustness. This controller has a gain margin of 2.7 and a phase margin of 46 degrees.

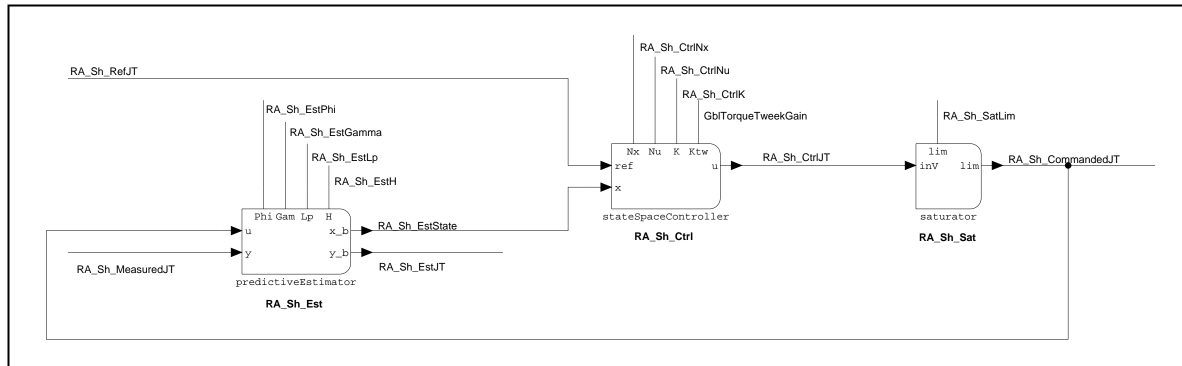


Figure 6.5: **Joint-control layer dataflow.**

Shown are the components required to control the shoulder and elbow motors of the right arm. Two main ControlShell.4.x components are used to control the torque at the robot joints. A state space controller component (`stateSpaceController`*) implements a state-space control law:* $u = \mathbf{K}\,(\mathbf{N_x}\, r - \mathbf{x}) + \mathbf{N_u}\, r$*, where* r *is the reference torque (ref) and* u *is the commanded motor current. A state space predictive estimator (*`predictiveEstimator`*) estimates the system state:* $\mathbf{x}_{k+1} = \mathbf{x}_k + u + \mathbf{L_p}\,(y - \mathbf{H}\,\mathbf{x}_k)$*. The* `saturator` *component is used to enforce software limits on the actuator torques.*

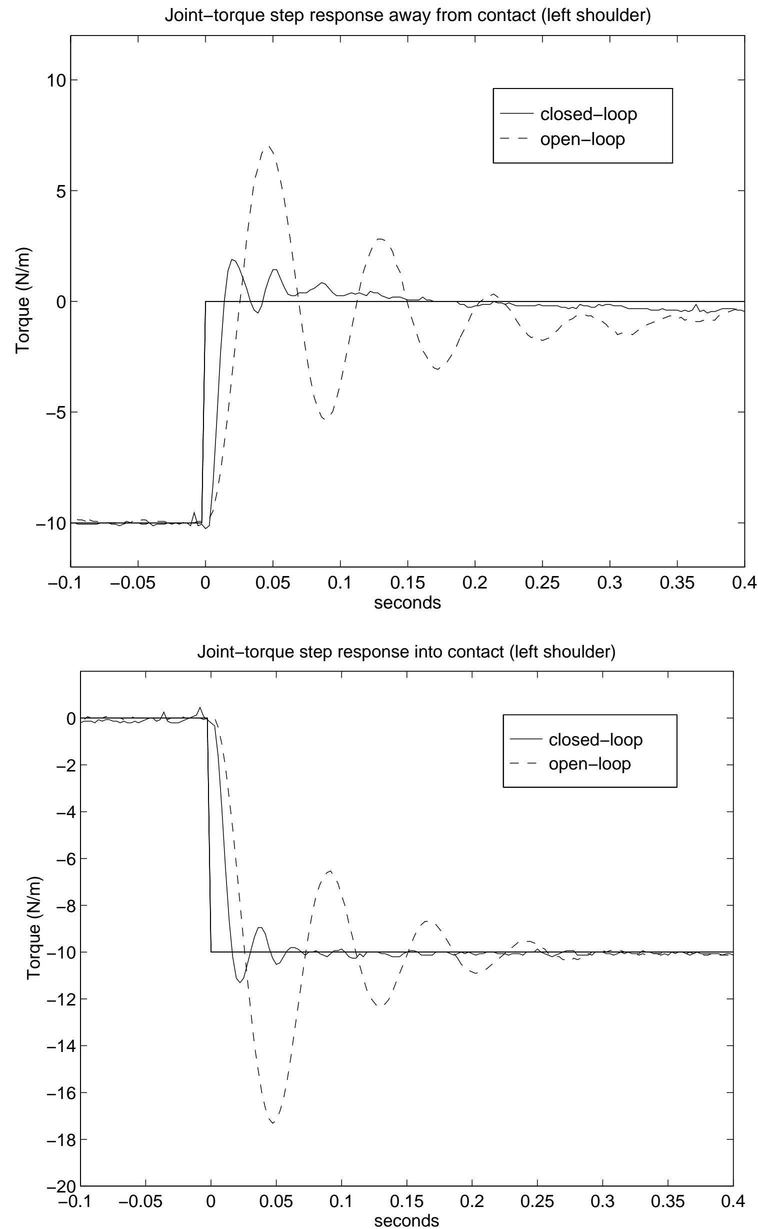


Figure 6.6: **Joint-torque step response of left shoulder subsystem.**

This Figure compares the open-loop and closed-loop responses of the right shoulder subsystem. The closed-loop system has a bandwidth of 40 Hz and shows good steady-state tracking and fast, well-damped transient response.

6.5 Arm-Control Layer

The control achieved at the joint-level allows the arm-control above it to treat the manipulator as if it had ideal motors and the commanded torques were delivered perfectly to the rigid links.

Ultimately any arm controller must generate joint-torque commands based on the manipulator's desired state; however, different controllers differ in two related but somewhat decoupled issues:

- The error law or control policy: selection of the space in which errors are computed and specification of reference behavior when reacting to external disturbances. Typical choices include position and velocity errors in Cartesian or joint space, impedance relationships and hybrid position/force control schemes.
- The implementation of the policy. The type of controller used to enforce the error law. Typical choices are inverse dynamics (also referred to as computed-torque) methods, kinematic controllers and independent-joint-space controllers.

Inverse dynamics controllers, which are the highest-performance control schemes, require a good kinematic and dynamic model of the manipulator. The following sections describe the control scheme used for the workcell manipulators.

6.5.1 Arm Controller

The arm controller used in the Manufacturing Workcell combines two controllers: a high-performance inverse-dynamics controller used for manipulator configurations away from kinematic singularities, and a low-performance (yet robust) joint-space PID controller used near singularities. These two controllers are blended in a transition region.

The high-performance controller implements an impedance relationship at an operational point located at the center of the tool (gripper), and uses an inverse dynamics (computed-torque) approach to enforce this relationship. This particular choice of control law (policy) and implementation is usually referred as "arm impedance control" and was introduced by Hogan [65]. The advantages of arm-impedance control in the context of flexible drive-train robots and object manipulation have been described in previous research [131, 154].

The structure of the arm-impedance controller is described below for the simpler case of a SCARA manipulator. The general formulation can be found in [157, 131]. Table 6.2 summarizes the meaning of the different symbols used in the description.

Symbol	*Dimension*	*Meaning*
$\mathbf{M_{imp}}$	4×4	Desired virtual mass (diagonal matrix).
$\mathbf{x}, \dot{\mathbf{x}}, \ddot{\mathbf{x}}$	4×1	Actual Cartesian position, velocity and acceleration of the operational point.
$\mathbf{x_{ref}}, \dot{\mathbf{x}}_{\mathbf{ref}}, \ddot{\mathbf{x}}_{\mathbf{ref}}$	4×1	Reference Cartesian position, velocity and acceleration of the operational point.
$\mathbf{K_p}, \mathbf{K_v}$	4×4	Desired virtual stiffness and damping (diagonal).
$\mathbf{f_{ext}}$	4×1	External force acting on the operational point.
$\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}}$	4×1	Vector of generalized coordinates and time derivatives.
$\mathbf{M}(\mathbf{q})$	4×4	Configuration-dependent mass matrix.
$\mathbf{B}(\mathbf{q})$	4×3	Configuration-dependent matrix of Coriolis term.
$\mathbf{C}(\mathbf{q})$	4×4	Configuration-dependent matrix of centripetal terms.
$\mathbf{g}(\mathbf{q})$	4×1	Configuration-dependent gravity vector.
$\mathbf{J}(\mathbf{q})$	4×1	Jacobian relating joint-rates to operational-point velocities.
$\boldsymbol{\tau}$	4×1	Torque vector.

Table 6.2: **Notation for arm-impedance controller derivation.**

The above sizes apply to a 4-DOF SCARA manipulator. The cartesian vector $\mathbf{x}$ *contains the position and rotation about the vertical axis of the operational point.*

First, the error law specifies that the manipulator operational point (normally the end-effector or tool) must obey the following impedance relationship:

$$\mathbf{M_{imp}}\,\ddot{\mathbf{x}} = \mathbf{M_{imp}}\,\ddot{\mathbf{x}}_{\mathbf{ref}} + \mathbf{K_p}\,(\mathbf{x_{ref}} - \mathbf{x}) + \mathbf{K_v}\,(\mathbf{x_{ref}} - \mathbf{x}) + \mathbf{f_{ext}} \tag{6.3}$$

Second, the actual equation of motions of the manipulator are:

$$\mathbf{M}(\mathbf{q})\,\ddot{\mathbf{q}} + \mathbf{B}(\mathbf{q})\,[\mathbf{q},\mathbf{q}] + \mathbf{C}(\mathbf{q})\,[\mathbf{q}^2] + \mathbf{g}(\mathbf{q}) + \mathbf{J^t}(\mathbf{q})\,\mathbf{f_{ext}} = \boldsymbol{\tau} \tag{6.4}$$

The cartesian coordinates of the operational-point are related to the configuration-space coordinates through the manipulator Jacobian as:

$$\mathbf{x} = \mathbf{J}(\mathbf{q})\,\mathbf{q} \quad \Rightarrow \quad \ddot{\mathbf{x}} = \mathbf{J}(\mathbf{q})\,\ddot{\mathbf{q}} + \mathbf{J}(\mathbf{q})\,\mathbf{q} \Rightarrow \quad \ddot{\mathbf{q}} = \mathbf{J}(\mathbf{q})^{-1}\,(\ddot{\mathbf{x}} - \mathbf{J}(\mathbf{q})\,\mathbf{q}) \tag{6.5}$$

The impedance relationship (6.3) can be enforced if actuator torques are chosen in (6.4) such that the resulting acceleration $\ddot{\mathbf{q}}$ (equation (6.5)) results in operational-point accelerations $\ddot{\mathbf{x}}$ that verify the impedance relationship. This scheme is illustrated in Figure 6.7.

The kinematics and equations of motion for this type of SCARA manipulators have been derived by previous researchers [66]. These equations are included for completeness in Appendix D.

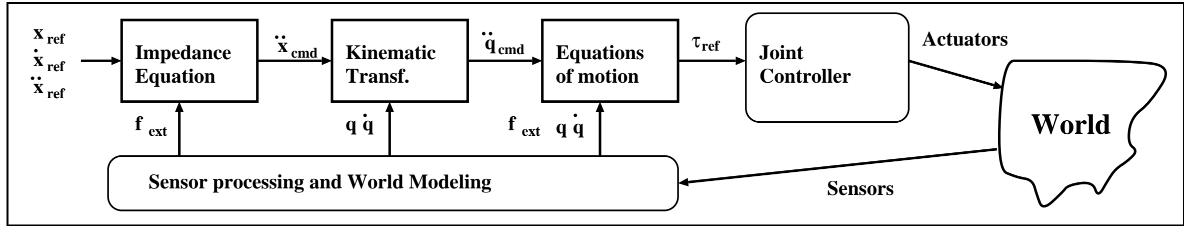


Figure 6.7: **Block diagram for the computed-torque impedance controller.**

The arm controller enforces an impedance relationship at the arm operational point (OP). Given the cartesian error between the reference (desired) and actual states for the OP, the impedance relationship specifies the commanded acceleration $\ddot{\mathbf{x}}_{\mathbf{com}}$ *for the OP (i.e. the acceleration that will satisfy the impedance relationship). This acceleration is transformed into the equivalent joint-accelerations* $\ddot{\mathbf{q}}_{\mathbf{cmd}}$ *using the manipulator kinematics. Given* $\ddot{\mathbf{q}}_{\mathbf{cmd}}$ *, the manipulator dynamics are used to determine the reference joint torques* $\boldsymbol{\tau}_{\mathbf{ref}}$ *that will achieve this joint-acceleration. These joint torques become the reference command for the joint-control layer below.*

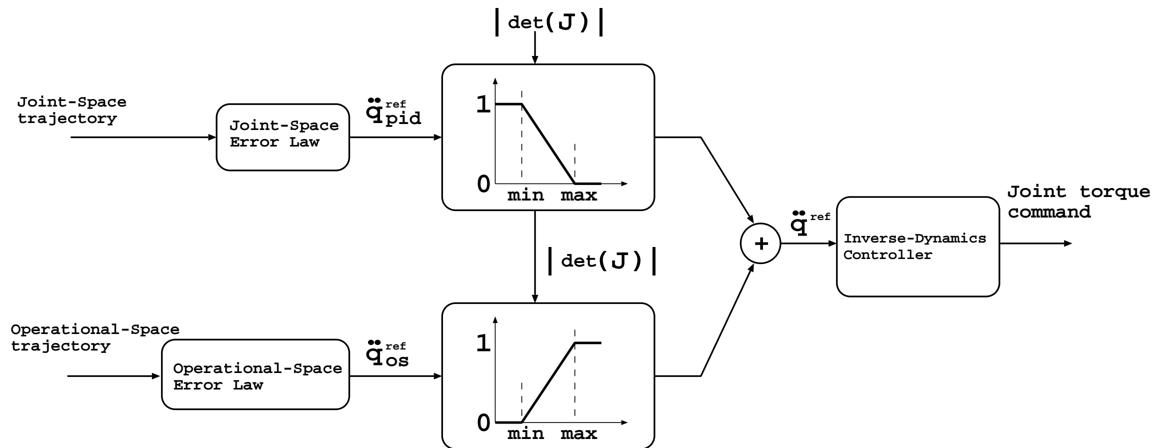


Figure 6.8: **Merging a cartesian-space computed-torque controller with a joint-space PID controller.**

The strategic module is required to specify both the cartesian-space and the corresponding joint-space trajectories for any arm motion. These trajectories are run in parallel through a cartesian-space error law (see Figure 6.7) and a joint-space error law. The commanded joint accelerations $\ddot{\mathbf{q}}$ *from the two controllers are combined to generate the actual joint-acceleration command* $\ddot{\mathbf{q}}^{cmd}$ *for the inverse dynamics controller. The combination is a weighted sum with weights that depend on how close the manipulator is to a singularity as measured by the value of the determinant. The transition interval corresponds to relative elbow angles* $|q_{el}|$ *between 0.1 and 0.2 radian.*

The disadvantage of this inverse-dynamics controller is that it cannot be used to transition the arm through a kinematic singularity (which occurs whenever the elbow is fully extended). At the kinematic singularity, the Jacobian matrix becomes singular, and its inverse cannot be computed as required in equation (6.5). The inverse-dynamics controller was modified so it would

be stable even at the singularity by using a Jacobian pseudo-inverse in equation (6.5) (i.e. whenever the determinant of the Jacobian is below a threshold, the pseudo-inverse is used instead of the Jacobian inverse). Although this modified controller works well in regulation, it cannot be used to follow reference trajectories that cross the singularity, because these trajectories cannot be specified exclusively in operational-space. In essence, the workcell manipulators are redundant: There are two arm configurations ("elbow in" and "elbow out") that result on the same cartesian location of the operational point. Two different (joint-space) trajectories that start from the same initial state, go through the singularity, and come out with opposite elbow configurations are indistinguishable to the operational-space controller.

It is important to be able to cross the singularity under control to take full advantage of the arm workspace. For this reason a hybrid approach was developed: Commanding an arm trajectory requires both the operational-space trajectory and the corresponding joint-space trajectory to be simultaneously specified. The controller monitors the proximity of the arm to a singularity (by evaluating the determinant of the Jacobian matrix). Away from the singularity the inverse-dynamics controller is used. Near the singularity, a pure joint-PID controller is used. In the transition region, a weighted combination of the reference commands from both controllers is used. In either case, the commanded joint-space accelerations are fed to the inverse-dynamics controller as illustrated in Figure 6.8.

The parameters used for both controllers are summarized in Table C.1 (Appendix C). A word on the selection of impedances and virtual masses: Note that for $\mathbf{f_{ext}} = 0$ only the ratios $\mathbf{K_p} / \mathbf{M_{imp}}$ and $\mathbf{K_v} / \mathbf{M_{imp}}$ are relevant. In view of this, $\mathbf{K_p}$ was chosen (through an iterative process) to achieve small steady-state tracking errors and $\mathbf{K_v}$ adjusted to keep the system slightly under-damped (for faster performance). The selected gains correspond to a closed-loop bandwidth of $f_{cl} = 3.26Hz$, and a damping ratio of $\zeta_{cl} = 0.6$. When the arm is in contact with the environment, the force/torque sensors measure $\mathbf{f_{ext}}$ and the operational point is commanded to respond with the same acceleration as a mass of value $\mathbf{M_{imp}}$. For this reason, larger $\mathbf{M_{imp}}$ results in more stable, yet "slower" contact behavior. The value finally chosen was a compromise for the required tasks. Gains for joint-space error law were selected such that they produced commanded accelerations of similar magnitude as the operational-space error law for small joint errors at the singularity boundary. In other words, $\mathbf{K_p^{pid}} = \mathbf{J(q)}\ \mathbf{K_p} / \mathbf{M_{imp}}$ and $\mathbf{K_v^{pid}} = \mathbf{J(q)}\ \mathbf{K_v} / \mathbf{M_{imp}}$ at this boundary.

6.5.2 Arm Controller Performance

Figure 6.9 illustrates the implementation of the arm-level controller in terms of reusable software components. The full inverse-dynamics computed-torque controller for both arms was run at 100 Hz on a dedicated processor board[2].

A straight-line slew that pushes the edge of the controller's tracking performance is illustrated in Figure 6.10. This slew moves the arm across its workspace (maintaining the same configuration) in 2.5 seconds, with extremely good tracking. This performance could not be achieved by the joint-based PID controller, as was demonstrated by previous research [131]. Figure C.1 in Appendix C illustrates the step response of the arm-control layer.

Figures 6.11 and 6.12, illustrate control system performance when the arm traverses a kinematic singularity. The hybrid controller is able to control the arm motion smoothly through the transition.

This thesis does not make the claim that the approach followed here to achieve stable performance through the singularity is valid in general. Merging a singularity-free (yet low-performance) controller with a high-performance controller in such a way that the singularity-free controller is used in proximity to the singularity is simple and intuitively appealing (in fact is an approach commonly followed by so called fuzzy controllers). However, there are a multitude of issues to be addressed more formally for this approach to be rigorous. It is not even clear under which circumstances a weighted (configuration-dependent) average of two controllers results in a stable controller. This investigation could be the topic of continuing research.

6.6 Object-Control Layer

The purpose of the object-control layer is to allow the workcell to manipulate objects either single-handedly or cooperatively using both manipulators. In this section, the term *cooperative* control is used in contrast with *coordinated* control. In either case an object reference trajectory is specified; but in coordinated control, that trajectory is transformed (through the grasp kinematics) into reference arm trajectories, and each individual arm is controlled from its own trajectory (regardless of what the other arm, or the object are doing). In cooperative control no individual-arm trajectories are computed. Rather, the arm commands are computed directly from the object trajectory and error. This closes another control loop on the object state. Figure 6.13 contrasts these approaches. True cooperative object control often requires force sensing at the arm end-effectors so that the interaction forces can be accurately controlled.

[2]A VME-based single-processor computer containing a 33 MHz m68040 processor.

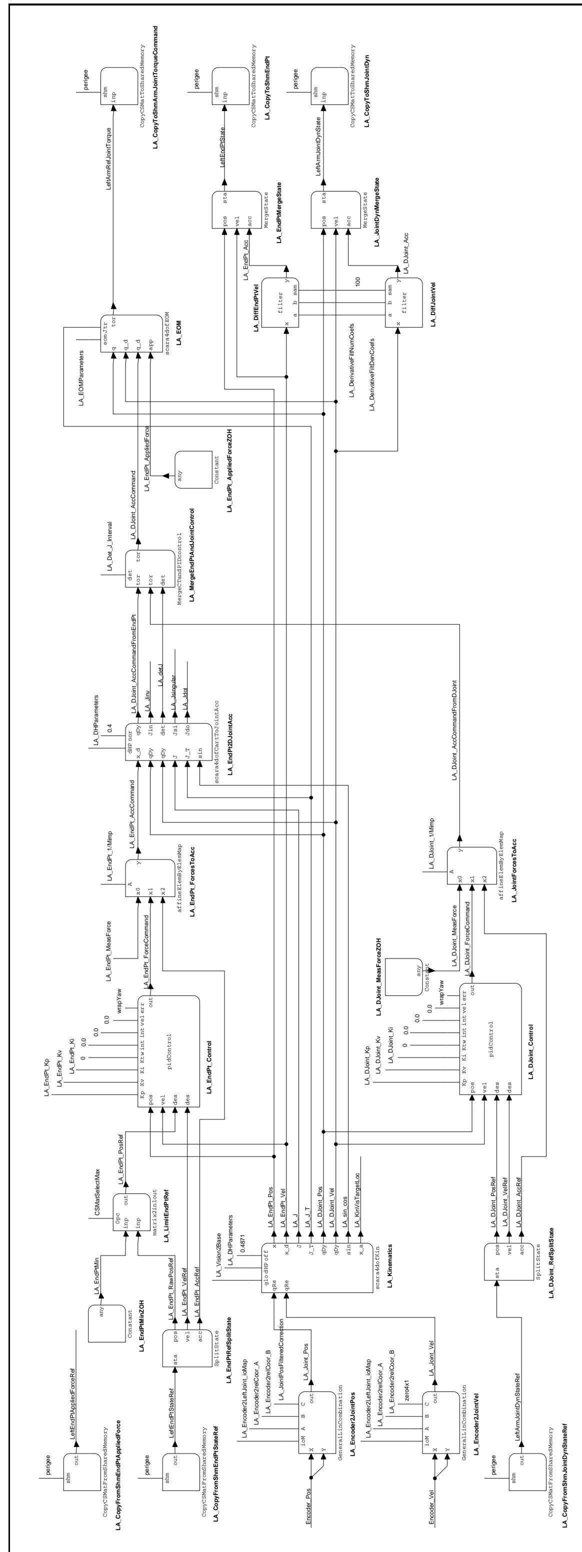


Figure 6.9: **Arm-control layer dataflow.**

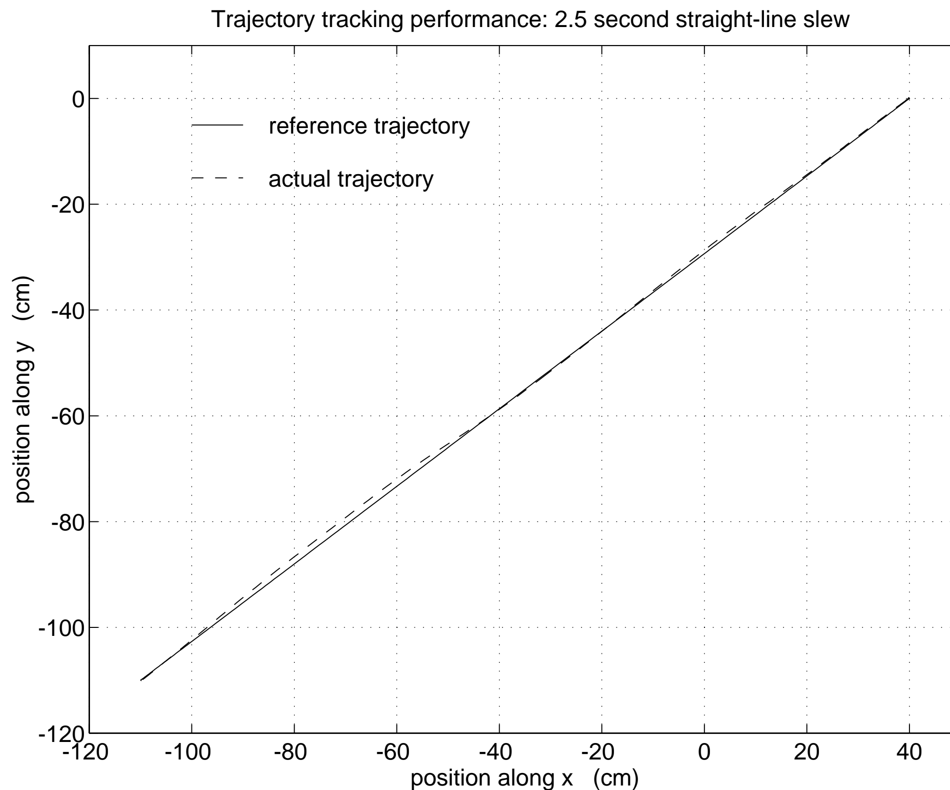


Figure 6.10: **Experimental tracking performance: straight-line path**

Illustration of the tracking response of the right arm. The reference is a fifth-order polynomial trajectory for the arm endpoint, commanding it to follow a 1.75 m straight-line path in 2.5 sec. This trajectory requires accelerations of up to 1.2 *m/s*^{2}. *The maximum tracking error is* 1.4 *cm.*

There have been many approaches to cooperative object control [151, 79, 60, 205, 187, 142]. This experiment uses the *Object Impedance Control* (OIC) approach originally developed by Schneider [154, 155]. This approach draws from Hogan's impedance control concept [65] and the work of Nakamura [114]. OIC was originally developed for fixed manipulators handling a rigid object. This work was later extended to manipulators on a mobile base [190, 189, 42] and manipulation of objects with internal dynamics [105, 106]. These researchers have demonstrated the utility of the OIC approach to cooperative-object manipulation. Detailed derivation of the OIC approach can be found in [155, 131]. This section will simply state the main concepts and relevant equations.

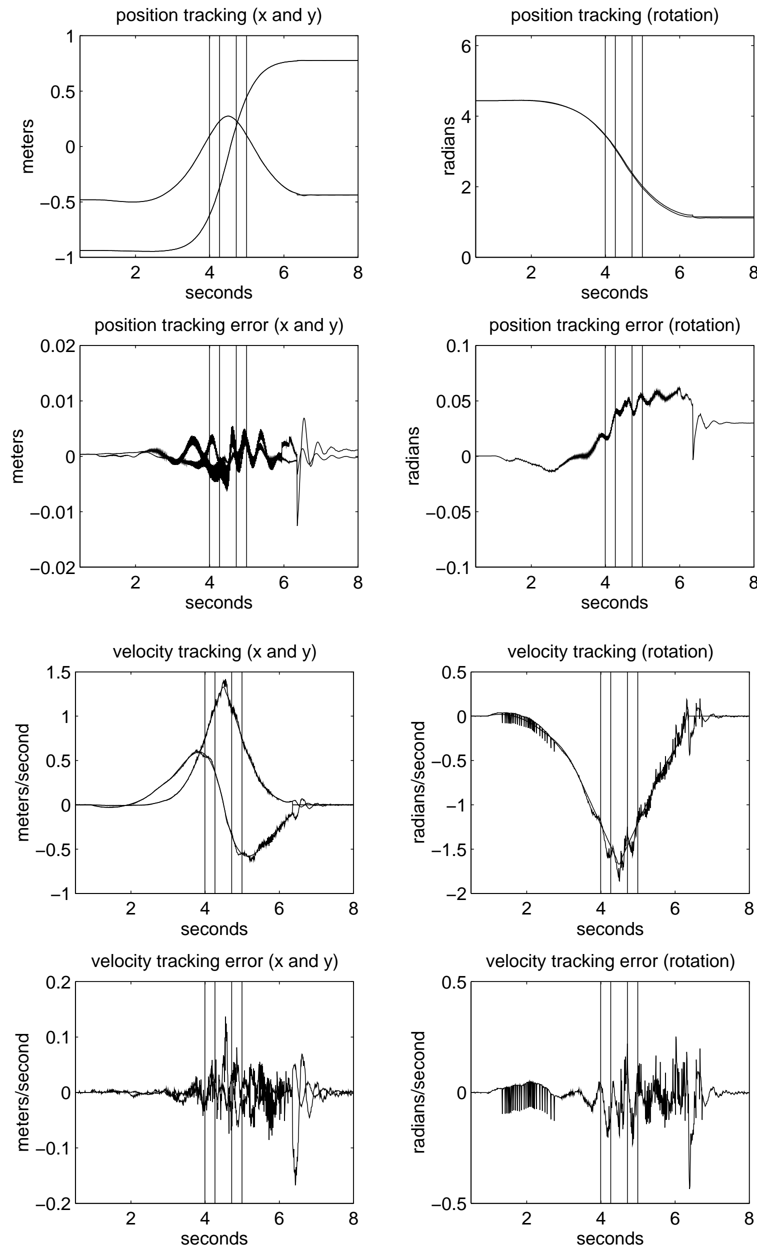


Figure 6.11: **Experimental tracking performance: path through kinematic singularity.**

Tracking performance on X, Y, yaw position and velocities. The above plots illustrate reference and actual trajectories (top line), the tracking error (second line), the reference and actual velocities (third line), and velocity tracking error (fourth line). The vertical lines indicate the region where the arm is at a kinematic singularity. In the center band between the innermost vertical lines, the arm is under pure joint-based control, in the adjacent bands, cartesian and joint control commands are averaged (see Figure 6.8), away from the singularity pure cartesian control is used.

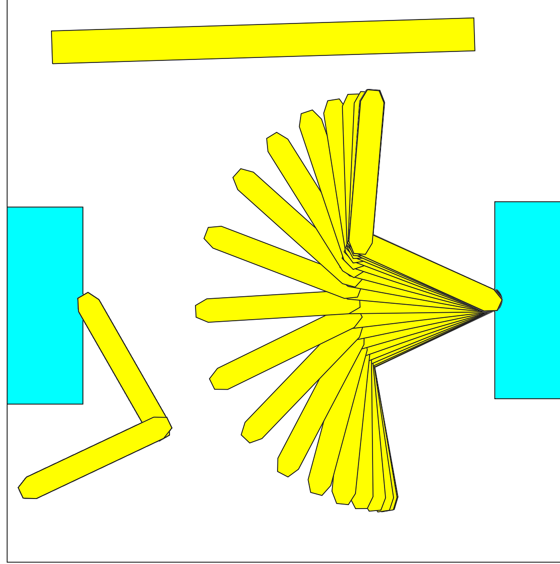


Figure 6.12: **Animation of trajectory through kinematic singularity.**

This figure animates the data collected during the trajectory shown in Figure 6.11.

The OIC is a model reference controller which enforces an impedance relationship on the state (position, velocity, and acceleration) of a certain point in the object (the object's Remote Center of Compliance or RCC)[3]:

$$\mathbf{M_{imp}}\,\ddot{\mathbf{x}} = \mathbf{M_{imp}}\,\ddot{\mathbf{x}}_{\mathbf{ref}} + \mathbf{K_p}\,(\mathbf{x_{ref}} - \mathbf{x}) + \mathbf{K_v}\,(\mathbf{x_{ref}} - \mathbf{x}) + \mathbf{f_{ext}} \tag{6.6}$$

The above equation can be interpreted as representing the equations of motion of a *virtual* object that is affected both by the forces applied to the object $\mathbf{f_{ext}}$ and a virtual force that makes it behave as if it were attached to the reference trajectory by a spring and dash-pot on each degree of freedom.

The dataflow of the OIC is illustrated in Figure 6.14. First the RCC and a reference "state" for the RCC, $(\mathbf{x_{ref}}\,, \mathbf{x_{ref}}, \ddot{\mathbf{x}}_{\mathbf{ref}})$ are specified. Given the actual object state $(\mathbf{x}, \mathbf{x})$, and the rigid-body transformation to the RCC, the impedance relationship of equation (6.6) can be used to determine $\ddot{\mathbf{x}}$ (the acceleration that the RCC should have to satisfy the impedance relationship). The acceleration of the RCC ($\ddot{\mathbf{x}}$) can now be transformed back to obtain the required object acceleration $\ddot{\mathbf{x}}_{\mathbf{com}}$. Since the actual EOM of the body and the grasp transforms are known, the required manipulator-end-effector forces and accelerations can be computed. These required end-effector

[3]This point does not have to be physically in the body. It is sufficient that it remains fixed in any body-fixed reference frame. That is, it is related through a rigid-body transformation to the object's position.

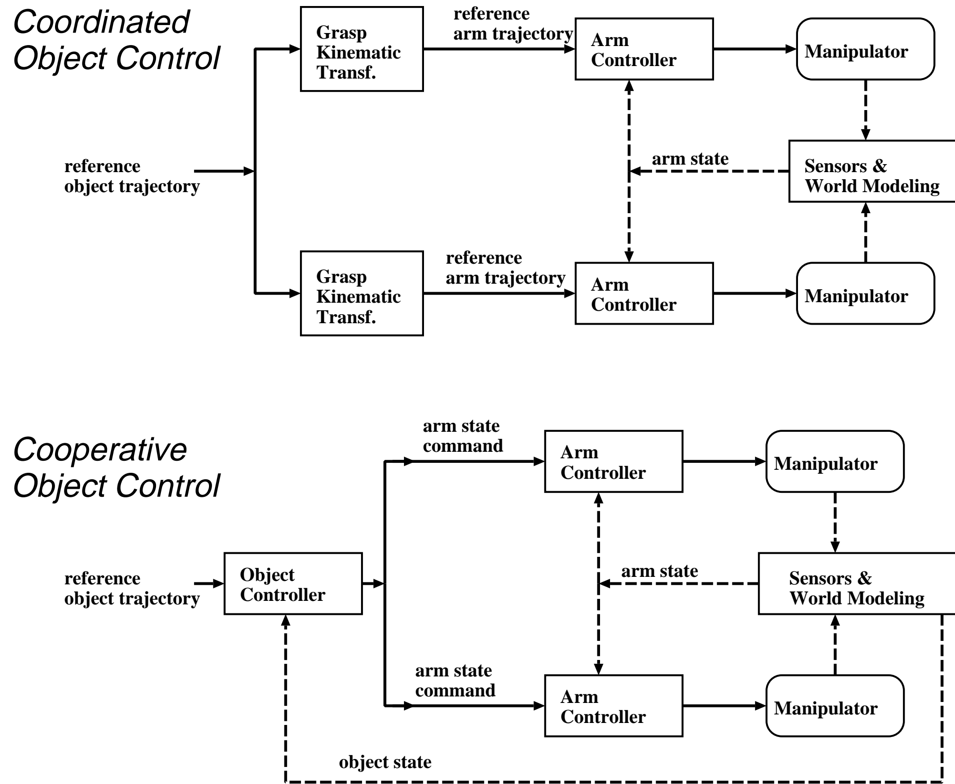


Figure 6.13: **Comparison of cooperative object control with coordinated object control**

In coordinated object control the reference trajectory for the object is merely transformed using the grasp kinematics into reference trajectories for each arm. Each arm is then controlled independently from its own trajectory. Cooperative object control generates no arm trajectories directly. Instead, arm commands are generated from the reference object trajectory and the error between the trajectory and the actual object state. Cooperative object control thus closes an additional, important control loop on the object state.

accelerations/torques become commands for the arm-level controller. In case there is redundancy solving the required manipulator forces on the object, the redundancy can be used to control the internal forces the object is subject to.

The above OIC formulation is restricted to situations where the manipulator arms holding the object are not at a singular configuration. This restriction is enforced by the strategic-control layer.

A nice feature of the OIC approach is that the EOM of the object decouple the arms so that, assuming each arm can compute locally its own required reference end-effector state and applied forces[4], each arm would only need to know about its own dynamics to control itself. This

[4]This computation transforms the required object acceleration through the known grasp transforms and distributes the required force/torque on the object among the two arms.

Figure 6.14: **Dataflow of the object-impedance controller**

Implementation of the object-control layer using ControlShell.4.x components. This diagram illustrates the control loop for the "barWM" object. The reference trajectory `barWMDesState` *can originate from a planned via-point trajectory (*`VIP_Generate` *component), a locally generated intercept (*`fifthOrderStateTrajectory`*), or a tracking (*`WM_TrackConveyorObject`*) trajectory. The* `ObjImpControl` *components combine the reference state, actual state, and (estimated) external force on the object to generate reference states and applied forces for each one of the manipulators. The* `ObjImpControl` *component uses the specific grasp transforms, mass properties, and virtual object behavior set by the strategic-layer (using the private world-model interface) to implement the desired impedance relationship.*

feature makes the algorithm easily parallelizable and extensible to multiple independent robots that cooperate in the manipulation. The work of Dickson [42] addresses these issues.

Figure 6.15 illustrates the tracking performance of the OIC for a cooperative object motion in free space animated in Figure 6.16. Comparisons with other object-level controllers and contact experiments have been documented extensively by Schneider [155, 154] and Pfeffer [131].

6.7 Strategic-Control Layer

The underlying object and arm-control layers allow the strategic layer to command reference object and arm trajectories. The strategic layer performs the following five functions: (1) It generates all the required trajectories, (2) it keeps track of which objects are being grasped by which manipulator(s), (3) it sequences through the different steps required to achieve individual robot commands (defined

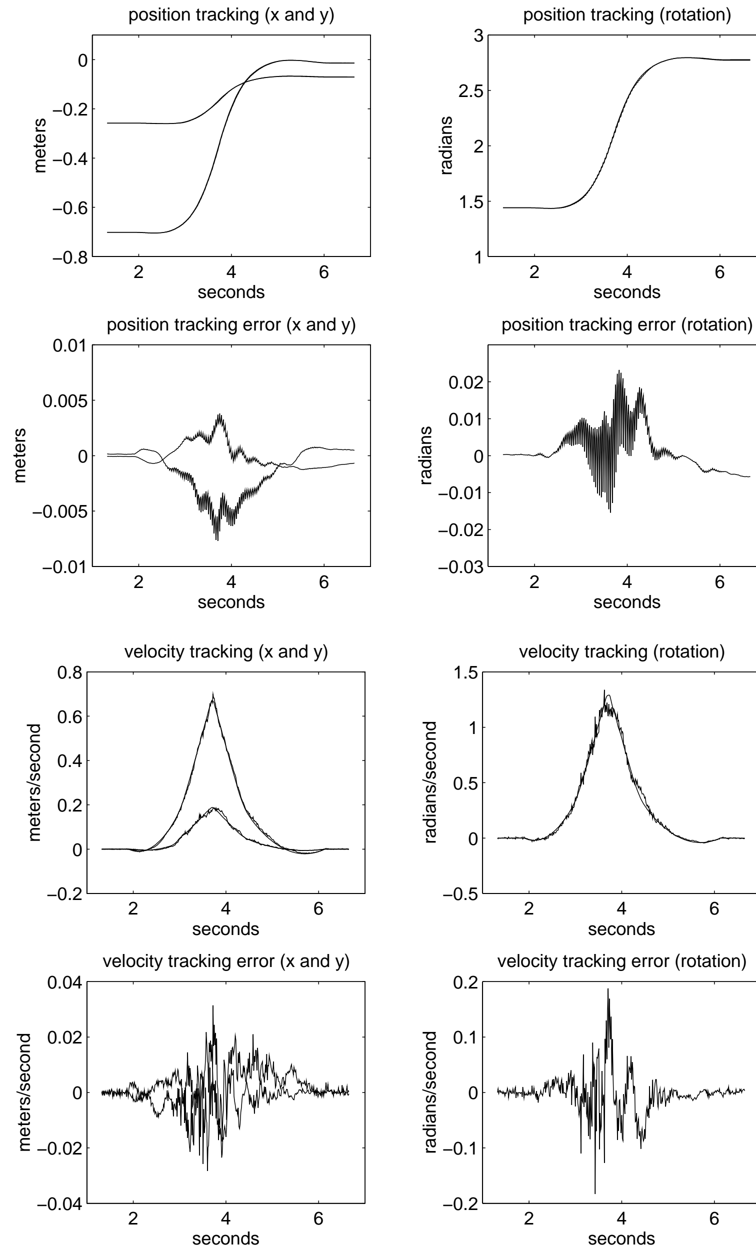


Figure 6.15: **Experimental tracking performance of the object-impedance controller**

Tracking performance on X, Y, and yaw position and velocities. The above plots illustrate reference and actual trajectories (top row), the tracking error (second row), the reference and actual velocities (third row), and velocity-tracking error (fourth row), for the object trajectory shown in Figure 6.16.

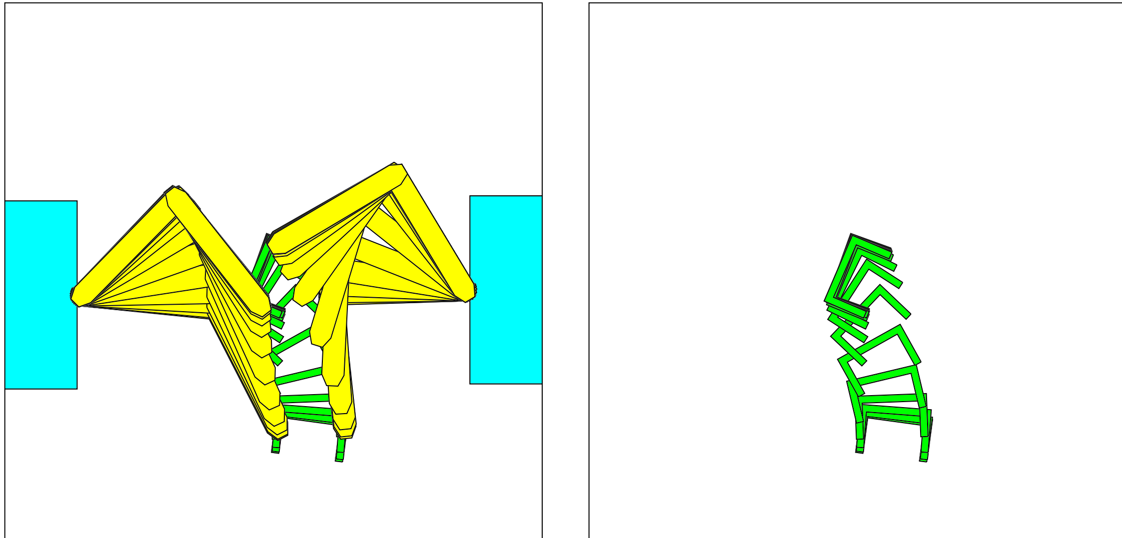


Figure 6.16: **Animation of trajectory followed by the object under cooperative control**

This figure animates the data collected during the trajectory shown in Figure 6.15.

in Chapter 3), (4) it monitors the progress and safety of the system, and (5) it takes corrective actions when failures occur.

6.7.1 The Finite-State Machine Programming Model

The strategic control of the workcell uses the Finite-State Machine Engine available within the ControlShell.4.x programming system. This approach to strategic robot-control was introduced by Schneider [154, 157] and later expanded by Ullman [189]. In their work, they developed strategic controllers based on a finite-state-machine programming model (FSM) allowing individual arm motions, dual-arm capture of a single moving object, and cooperative manipulation of an object. The dual-arm workcell must perform the aforementioned tasks. Additionally, the dual-arm workcell needs to handle concurrent manipulation of multiple objects (e.g. one object per arm), mixed arm and object motion (e.g. an arm is manipulating an object while the other is moving), and in general be extensible to any number of arms/objects. These needs represent a significant departure from the previous work in that they require multiple FSMs to be active concurrently so that different parts of the total system (e.g. each arm) can operate independently. In other words, some operations of the robotic workcell require independent, yet coordinated, strategic control of each arm.

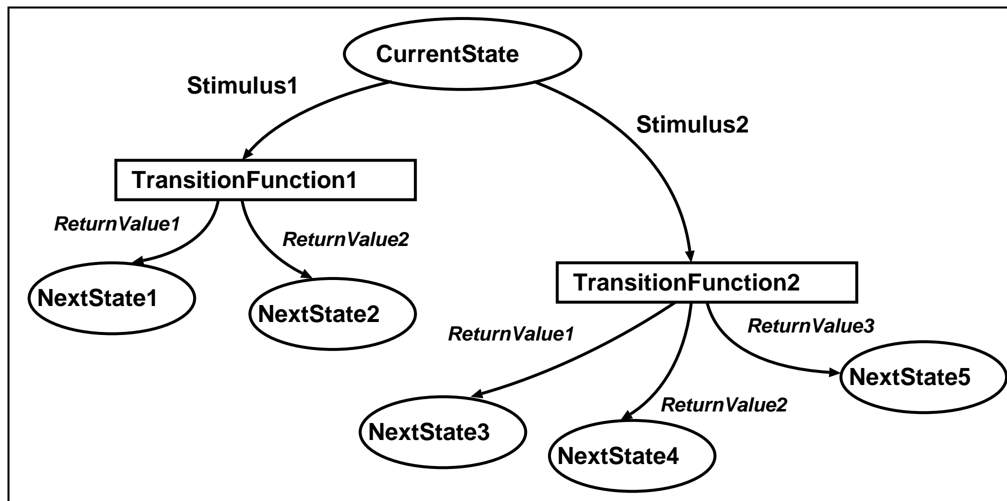


Figure 6.17: **ControlShell.4.x's Finite-State Machine model**

The arrival of a matching stimulus triggers the execution of the corresponding transition function whose return value determines the next state.

ControlShell.4.x's FSM has been described in detail and compared with robot-programming techniques in [154, 157]. Here ControlShell.4.x's FSM model is described briefly so that the subsequent state diagrams are meaningful. ControlShell.4.x's FSM is an extension of the traditional FSM model: A *state* represents the internal configuration of the system and encodes the information needed to determine its future behavior. A *state transition* occurs as a response to an external *stimulus*. For each state and stimulus pair, there is an associated *transition function*. The arrival of valid stimulus triggers the execution of the corresponding transition function whose return value determines the next state. Stimuli that do not have an associated transition-routine are simply discarded (or handled by a pre-specified *catch-all* state). A stimulus is simply a message sent to the FSM and can be generated from anywhere in the system. In ControlShell.4.x, a FSM program is built using a graphical editor where the state, stimulus, and transition functions/return values are specified. The automatically-generated code is then linked to user-written code that implements the transition functions. ControlShell.4.x's FSM facility contains other useful constructs to facilitate building reactive systems. The interested reader may find further details in [158, 159, 144].

6.7.2 Strategic Control of the Workcell Using FSMs

The strategic control of the workcell uses several concurrently-active FSMs as illustrated in Figure 6.18. The architecture consists of a Workcell Daemon and several FSMs (one per individual

arm). These FSMs communicate by sending stimuli to each other. The Workcell Daemon receives robot commands through the Robot Interface (Chapter 3), and sends stimulus to one (or more) of the individual-arm FSMs. The individual FSMs control the arms and synchronize with each other while informing the World Modeler of the significant events. The strategic layer affects the object and arm control layers by modifying their dataflow (usually achieved by changing configurations, that is, the active components in the system) and the parameters used by the different components. In this manner, the strategic level can change control modes, generate/start/stop trajectories, and modify controller parameters (e.g. grasp transforms, impedance gains).

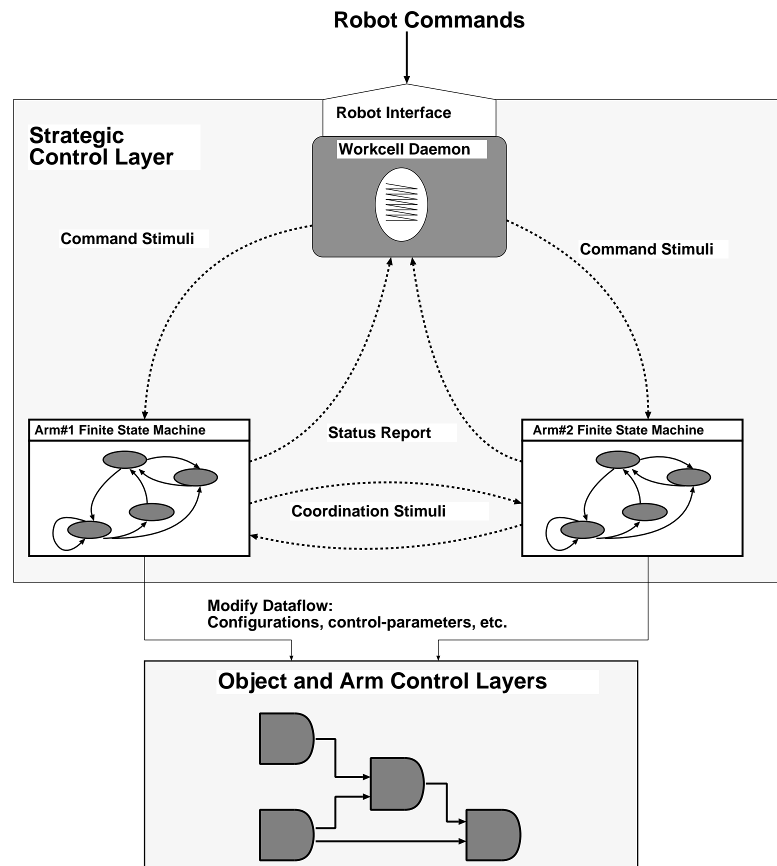


Figure 6.18: **Architecture of the strategic-control layer**

The strategic-control of the workcell uses a Workcell Daemon and a set of State-machine programs (one per manipulator in the workcell). The Workcell Daemon interfaces to external subsystems though the Robot Interface. Commands arriving through this interface are processed and sent as stimulus to one (or more) of the arm-FSM. This approach scales to more than two arms by adding an extra arm-FSM for each new arm.

Extensions to ControlShell.4.x's FSM Model

In order to make the approach easily expandable to any number of manipulators in the workcell, we have pursued the concept of a FSM program as defining an abstract data type (ADT) (as opposed to an instance of the type) from which multiple FSM-instances can be created. This view interprets a FSM as defining a behavior specification in terms of how to respond (which transition functions to call) to different stimuli. An FSM-instance binds a specification to private data that can be accessed by the transition-functions (similar to what happens when a method is invoked in an object in OO programming languages).

This view of FSM-programs as ADTs allows the strategic controller of the workcell to scale easily to any number of manipulators. New manipulators simply instance fresh copies of the arm-FSM ADT; each instance binding to different private data containing the specific information for that manipulator (kinematics, location, state etc.), which is obtained from the manipulator's name by querying the World Modeler. This approach is quite powerful but it does not come for free: The Arm-FSM and transition-functions must be carefully designed so that the different FSM-instances can interact and cooperate as required, even though the specific names (or even the number) of the arm-FSM instances are not known in advance. An example will help clarify this point: when picking up a part from the conveyor with two arms, both arms must synchronize so that they close the grippers simultaneously, and once the object has been grasped, cooperate to move the object. For this to work, the individual arm-FSMs must query the world modeler at run-time to find out with which other arm-FSM they need to synchronize. A simple extension to ControlShell.4.x's code-generation process provided this functionality.

The next two sections describe the different control modes (or configurations) the workcell can operate in, introduces the workcell daemon and the individual arm FSMs in detail, and illustrates how the FSM strategic programs utilize the different control modes to capture and manipulate objects in the workspace.

Reference Trajectories

In the absence of events (external or internal), different strategic-level FSMs remain in the same state, and the operation of the workcell is determined by the dataflow layer. Thus, workcell operation is fully characterized by two complementary dataflow aspects: trajectory-generation (the source of reference states) and the controller used. Loosely speaking, trajectory generation specifies the desired behavior of the system (trajectories for the arms and object to follow) while the controller

is responsible for commanding the actuators so that the system tracks the desired behavior. In the previous sections the different controllers have been described in detail. This section focuses on the possible sources of reference trajectories.

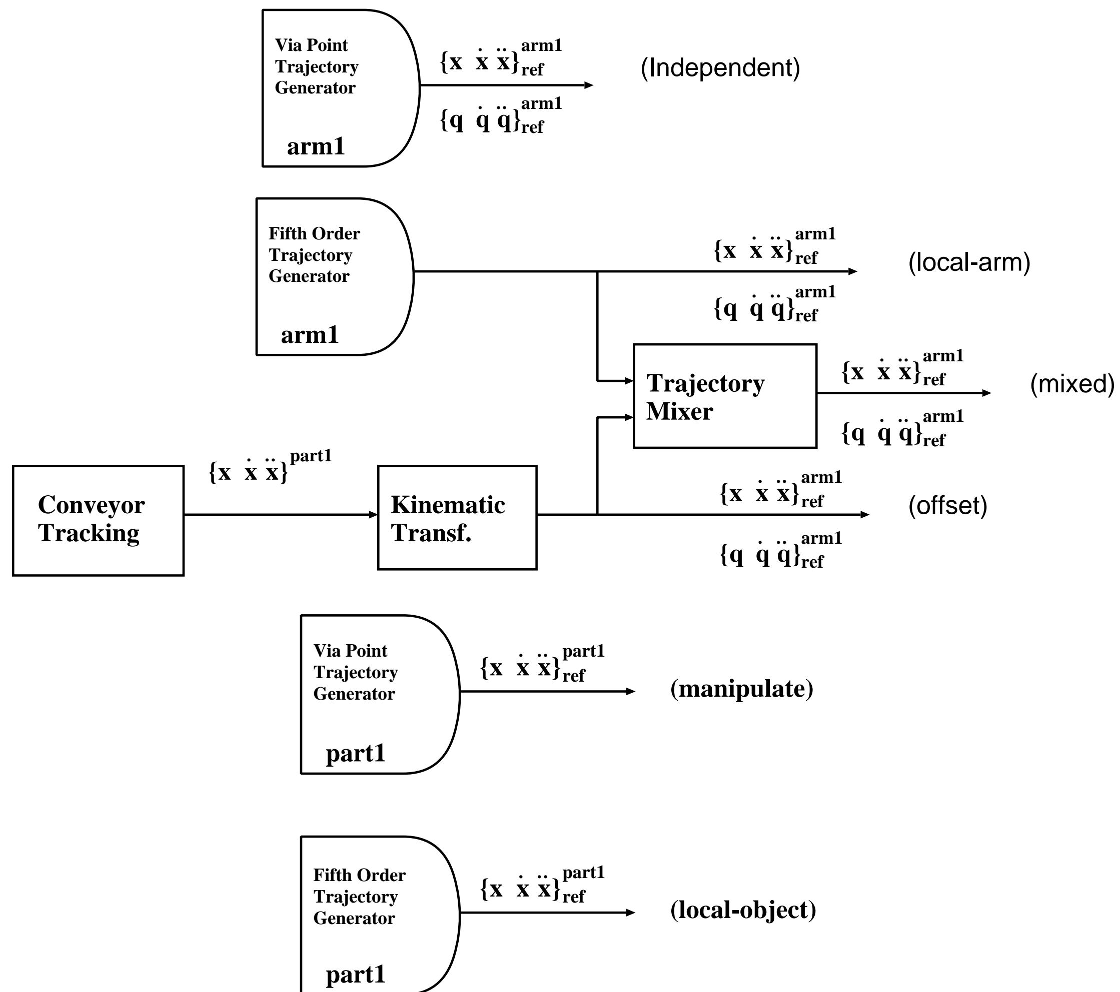


Figure 6.19: **Sources of arm and object trajectories**

For each arm, there are four possible sources of arm reference states: a via-point trajectory, a two-point fifth-order trajectory, a kinematic transformation from the state of an object, and a mix of the previous two. The sources of object reference states are either a via-point trajectory or a fifth-order trajectory.

Figure 6.19 illustrates the different sources of reference trajectory states: **Via-Point trajectories** are splined trajectories that traverse a sequence of via points (a.k.a. way points or through points). Via-Point trajectories are used by the planner (or any other subsystem) to command motions along a specified path. The on-line generation of such trajectories is a research topic in itself and is described

in detail in Chapter 7. **Fifth-Order trajectories** result from interpolated fifth-order polynomials that match pre-specified initial and final states at specified times. Generation of fifth-order trajectories from the boundary conditions is a straight-forward computation. Fifth-Order trajectories are useful to generate short intercept motions that match expected object states so that moving objects can be captured. **Trajectory mixes** serve two purposes: (1) they allow different degrees of freedom to be controlled from different source trajectories[5] and (2) (by using continuously varying mixing coefficients) allow smooth transitions between different trajectory sources.

Workcell Configurations

A *configuration* can be loosely defined as a state of the dataflow. The previous sections have described the different control modes for both arms and objects as well as the different trajectory sources. For the purposes of the strategic-control layer, a configuration (or operating mode) is defined by specifying both the control mode, and the source trajectories for each of the manipulators and grasped objects. Table 6.3 summarizes the possible configurations of each arm:

Configuration	*Object Traj. Source*	*Arm Traj. Source*	*Control Mode*
independent	None	Via Points	arm-impedance
local-arm	None	fifth order	arm-impedance
offset	None	slaved to object	arm-impedance
mixed	None	Mixed	arm-impedance
local-object	fifth order	slaved to object	object-impedance
manipulate	Via-point	slaved to object	object-impedance

Table 6.3: **Modes of Operation**

These are the main operating modes for each arm in the workcell. These modes are used by the FSM in the strategic-control layer to change the behavior of the system as a reaction to events and external commands.

Independent Mode. In this mode, the arm is controlled with the hybrid controller described in Section 6.5.1, either regulating to a fixed location or following a Via-Point trajectory. This mode is employed to move the arms, except in the case when the arm is about to grasp an object.

[5] This is important when an object is being captured since, the x,y position and orientation of the tool must match that of the object while the z value must be independently controlled to pick-up the object.

Local-Arm Mode. In this mode, the arm follows a locally-generated fifth-order trajectory. This mode is useful for small arm motions such as ones in the final stages of intercepting a moving object. During an interception, the final point in the fifth-order trajectory is the expected state of the object grasp location at the computed intercept time. This mode is only used once the arm is near the object's grasp position.

Offset Mode. To track an object, the reference state of the arms is obtained directly from the measured position of the object's grasp location. The desired grasp state is obtained by transforming the current object state using the desired grasp transform. This corresponds to the use of the information right out of the kinematic transformation module in Figure 6.19. This mode is used regardless of whether the object is static or in motion. Maintaining the information on the correct object state is the responsibility of the the World Modeller. This mode can be used to track an object from above (by specifying a grasp transform with some vertical offset) and to keep the arm at a constant position with respect to an object while waiting for the grasp to consolidate or for other arms to grasp.

Mixed Mode. To descend and pick an object, the reference arm state is generated by mixing two reference trajectories: The object's grasp location is used as a reference for all but the Z degree-of-freedom. A locally-generated fifth-order trajectory for the Z degree-of-freedom moves the arm onto the object.

Local-Object Mode. In this mode, the object and the arms grasping it are under object-impedance control, the reference state for the object comes from a locally-generated fifth-order trajectory. This mode is useful for moving the object short distances such as when lifting the object after a grasp, moving it down to release it, or (in the future) fine assembly maneuvers.

Manipulate Mode. In this mode, the object and grasping arms are also under object-impedance control. The reference object state is fed from a via-point trajectory. This mode is useful for moving objects around the workspace when the specific path followed by the object is important (in practice always, unless the motion is very short). The via points are normally specified by the planner subsystem. Note that the case when several arms are manipulating a common object is handled by the object-impedance controller that knows about this fact[6] and generates reference states for all grasping arms.

[6]More precisely, it is programmed with this fact by the strategic layer.

The Workcell Daemon

The Workcell Daemon is responsible for receiving robot-commands through the Robot Interface (see Chapter 3), process the commands, and coordinate the individual-arm FSMs. In order to be scalable to any number of arms in the workcell, the Workcell Daemon does not model what the individual arms are doing in detail. Instead, whenever a command arrives, it queries the World Modeler to decide if the command is acceptable given the the current state, and if so, where the command should be forwarded. For example, a command to move an object will be rejected unless that specific object is being grasped, and if so, it will setup and start the corresponding object trajectory. A command to move an arm will be simply forwarded to the specific arm FSM (which in turn may either move the arm or reject it altogether).

Individual-Arm FSMs

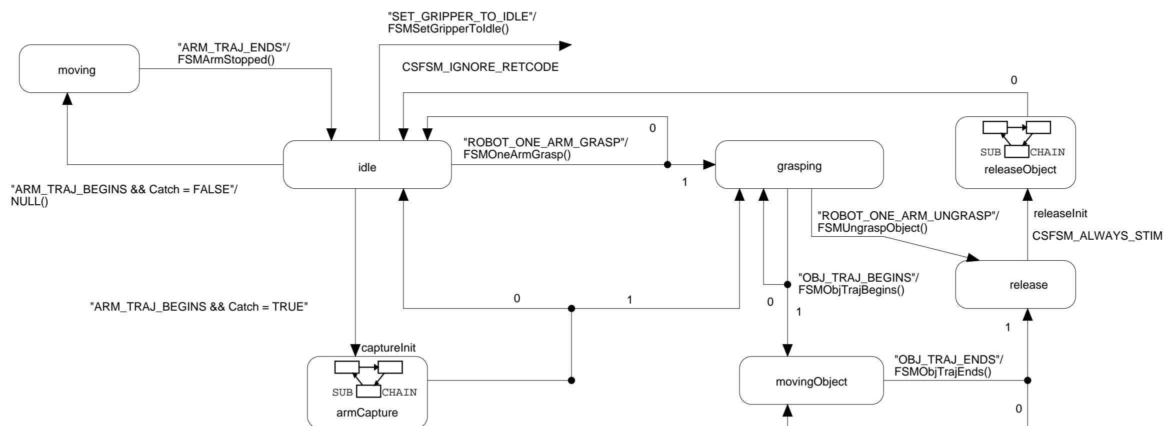


Figure 6.20: **Transition Graph for the Arm Finite-State Machines**

Two Arm Finite-State Machines (FSMs) are used for the strategic control of the workcell. These FSMs have identical functionality but are bound each to the private data of the corresponding arm. The two FSMs allow the independent strategic control of each arm. These FSMs communicate with each other to allow synchronization during coordinated arm maneuvers (such as picking an object with two arms).

Figure 6.20 illustrates the FSM of each individual arm. As previously explained, this FSM diagram represents conceptually an abstract-data type from which we instance several FSMs by binding the transition diagram with its own private data. The arm FSM receives stimuli from the Workcell Daemon, the other FSM, and several event-generating components in the dataflow[7]. If a valid

[7]For example the trajectory-generation components generate events every time a trajectory is started or completed.

stimulus is received, it executes the corresponding transition routine that in turn modifies the dataflow and may send other stimuli. The FSM utilizes *subchains* (fine-state-machine subroutines) to perform capture and release operations. These subchains are illustrated in Figures 6.21 and 6.22.

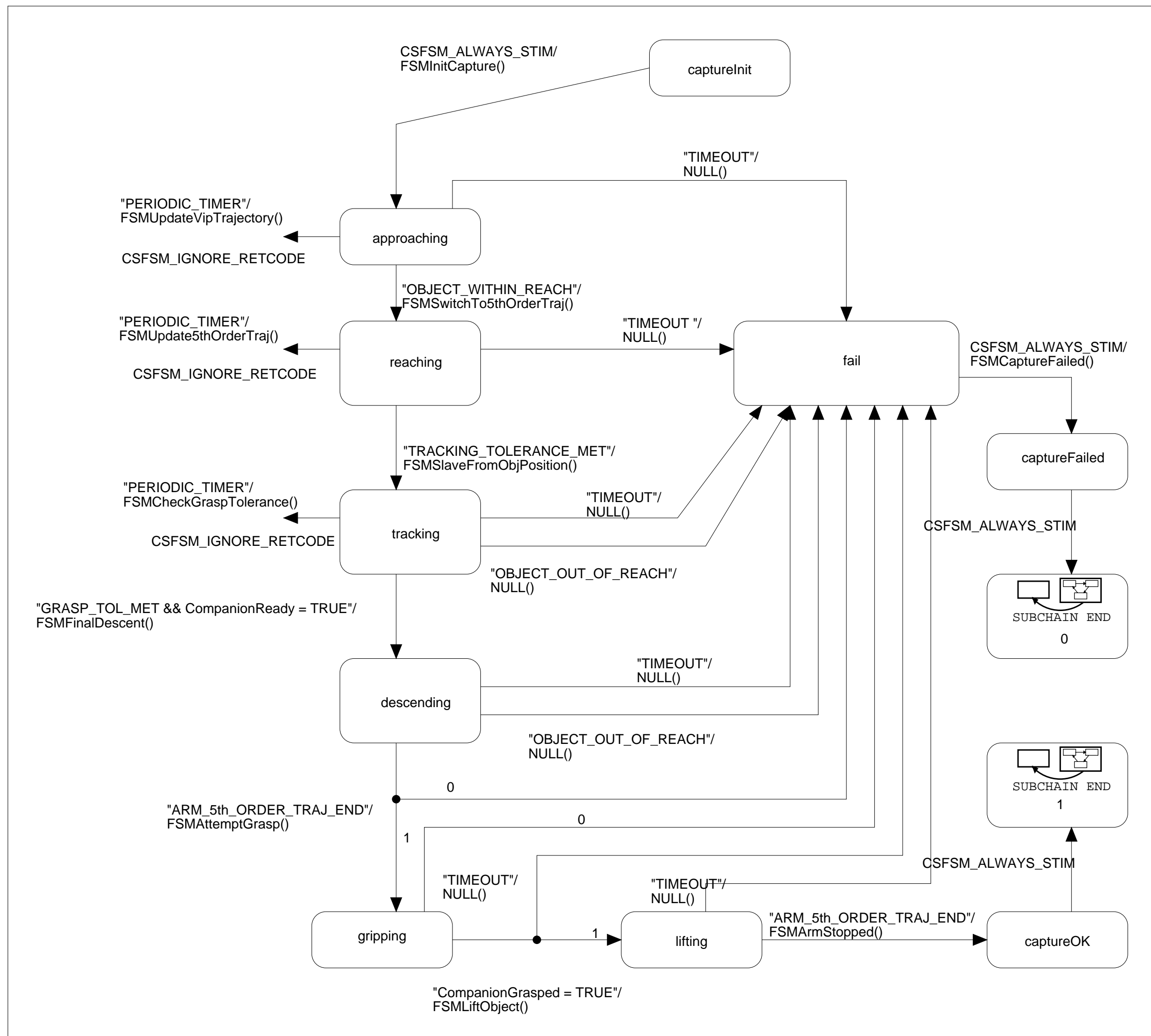


Figure 6.21: **Transition Graph for Capture Subchain**

Capturing an object from a moving conveyor requires a series of steps and control-mode transitions. The Capture subchain sequences each arm through these steps. Transitions are triggered by events such as the completion of a trajectory, a timeout, or the detection of an object within the grasping range.

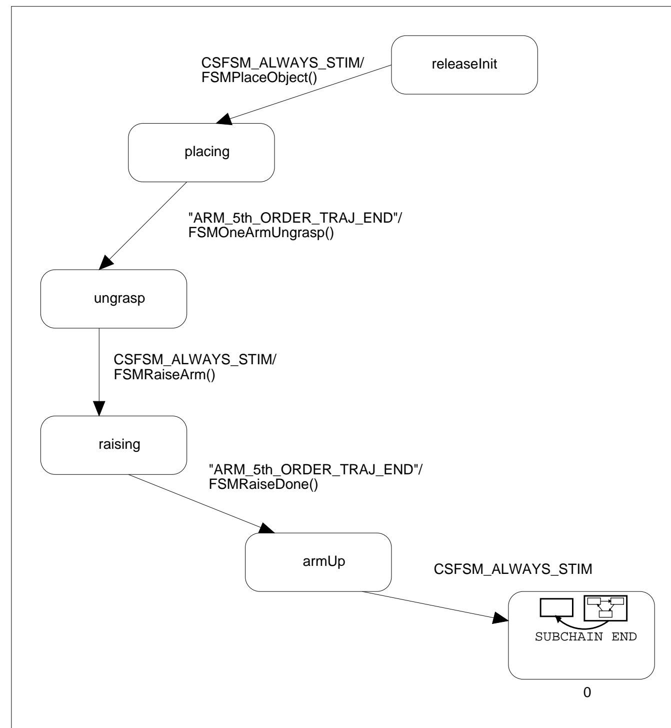


Figure 6.22: **Transition Graph for Release Subchain**

The use of a FSM subprogram to release an object allows delicate placement of the object at a pre-specified location under local, closed-loop control.

6.7.3 Picking an Object from the Conveyor Belt

This section describes a concrete example of the interaction of the Workcell Daemon, the arm FSMs, and the dataflow layer during the task of picking up an object from the conveyor.

The maneuver is triggered by the arrival of a `ROBOT_ONE_ARM_MOVE_N_CATCH` strategic-command through the Robot Interface. This command includes, among other things, the name of the arm to be used, an approach trajectory, the name of the object to pick and the grasp transform (from object position to grasp position). The Robot Daemon uses the name of the arm to query the World Modeler to ensure that the arm is available and in the `independent` configuration. Then it retrieves the the Via-Point trajectory generator component for that arm and starts the requested trajectory. The Robot Daemon also sends the `CaptureObject=YES` stimulus to the corresponding arm FSM. When the via-point trajectory starts, a “start” routine (installed by the Workcell daemon) automatically sends a `ARM_TRAJ_BEGINS` stimulus to the arm FSM which immediately enters the Capture subchain.

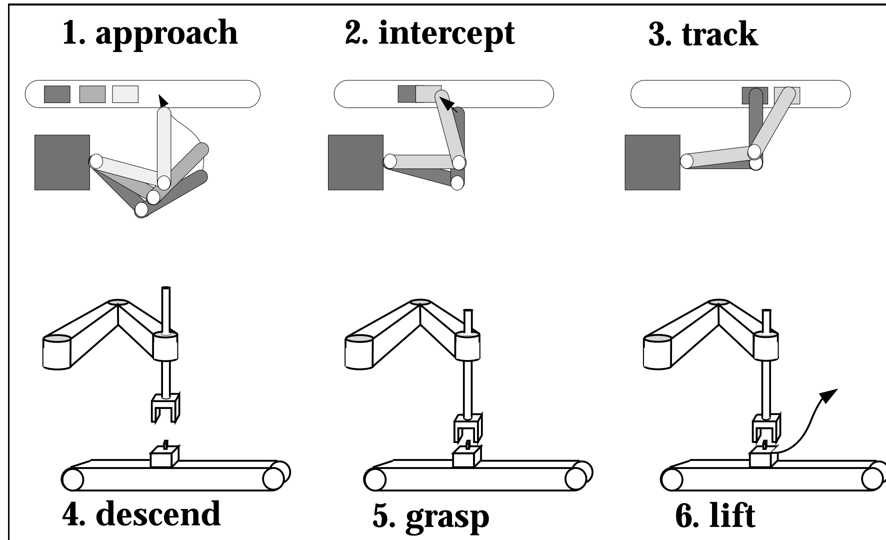


Figure 6.23: **Capture Strategy**

Six discrete stages are required for the capture of a moving object. During approach, the arm follows a pre-planned via-point trajectory (1). Triggered by the arm proximity to the object, the reach (intercept) stage uses a locally generated intercept trajectory to move the arm towards the object while matching the object's velocity (2). In the track stage the arm is regulated directly from the object's position and velocity (3). Three more stages: descend, grasp, and lift complete the capture.

The Capture subchain. The basic capture strategy is illustrated in Figure 6.23. To perform a capture, the Capture subchain transitions through a series of states as shown in Figure 6.21. The subchain immediately executes `FSMInitCapture()` which initializes a set of auxiliary "watchdogs" to send periodic stimuli and timeouts, and transitions to the `approaching` state. While in this state, the FSM receives `PERIODIC_TIMER` stimuli that trigger the `FSMUpdateVipTrajectory()` transition function that refines the last points of the trajectory, based on the most current estimate of the object's grasp position, at the intercept time. This function also checks whether the arm is within a certain tolerance of the current object grasp position, in which case an `OBJECT_WITHIN_REACH` stimulus is generated. The `FSMSwitchTo5thOrderTraj()` transition function changes the arm configuration to `Local-Arm mode` and starts a fifth-order intercept trajectory from the current arm state to the intercept state. This trajectory matches the expected positions, velocities, and accelerations at intercept time. While in the `intercepting` state, the trajectory is refined periodically (each time the `PERIODIC_TIMER` stimulus arrives). The periodic `FSMUpdate5thOrderTraj()` transition function also checks whether the arm is within a certain tolerance of the current grasp position. If so, the `TRACKING_TOLERANCE_MET` stimulus is sent. The corresponding `FSMSlaveFromObjPosition()` transition function switches the

arms to the `Offset Mode` configuration, the offset being the desired grasp transform with the addition of a small offset in height so that the arm looms directly above the object. The grasp tolerance is checked periodically by the `FSMCheckGraspTolerance()` transition function. When met, the `GRASP_TOL_MET` stimulus is sent. The `FSMFinalDescent()` transition function switches to the `Mixed Mode` configuration and sets the appropriate descent trajectory and mix coefficients so that the arm is tracking in the (x, y, θ) degrees of freedom while it descends on the object. When the trajectory ends, the installed notify routine informs the FSM with the `ARM_5th_ORDER_TRAJ_END` stimulus, and a grasp is attempted by `FSMAttemptGrasp()`. If successful, the arm transitions to the `gripping` state, the `FSMLiftObject()` function switches to the `Local-Object` configuration and initiates a lift trajectory. When the lift trajectory finishes, the `FSMArmStopped()` function switches the arm to `Manipulate Mode` (where it is slaved to the VIP-Trajectory generator for the object), and the Capture subchain returns leaving the arm FSM in the `grasping` state.

Note the numerous timeouts and error conditions that may arise during the capture maneuver. Auxiliary processes check for inter-arm collisions, kinematic singularities, and object reachability. If errors are detected, the corresponding stimulus is sent and the capture is aborted.

Also note that in the case where two arms must cooperate to capture the object, the corresponding FSM synchronize twice: once prior to the final descent and a second time before they lift the object. Each arm FSM is informed of the need for synchronization by the Workcell Daemon when the maneuver starts and they wait at the synchronization points for `CompanionReady=TRUE` messages from their peers.

6.7.4 Experimental Pick Operations

Figure 6.24 illustrates the value of several relevant signals during a single arm capture operation. Note the switch to the `intercept` state at time 4 sec. From there on the position, velocity, and acceleration (not shown) of arm tool matches that of the object[8]. Also note the switch to the `descend` state at 5.2 seconds. Note the velocities are matched during the time the grasp is consolidating ($t = 5.6\ s$ to $t = 6.6\ s$)[9]

[8]In fact the tool position, velocity, and acceleration matches that of the grasp location in the object rather than the object's. However in this case, since the object has a single grasp location, the object frame has been chosen to coincide with the grasp location.

[9]The grasp takes a long time to consolidate because the grippers are pneumatic and they are connected through a long, thin hose.

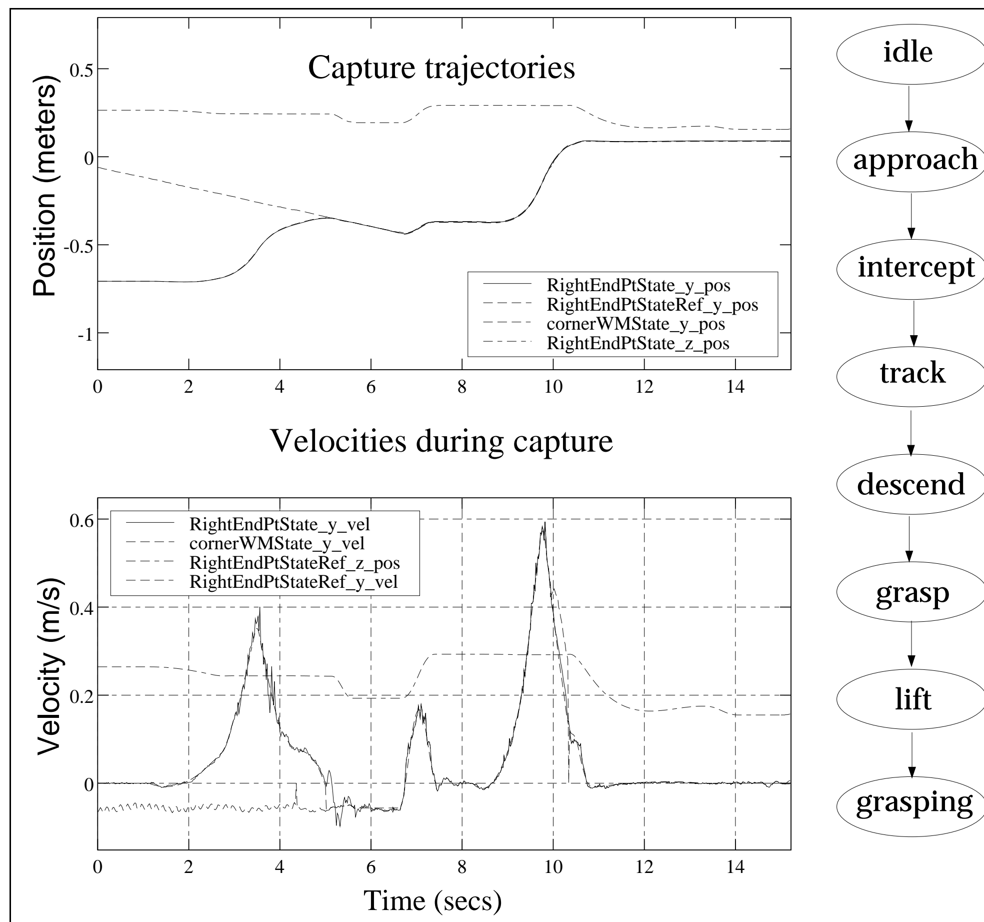


Figure 6.24: **Experimental single-arm capture of moving object**

The object is picked from the conveyor at $t = 6.5$ *secs. Trajectory generation switches from planner control to local intercept trajectory (*`intercept` *state) at* $t = 4$ *secs. Velocities are matched and remain matched during the descent and grasp phases (t = 5.2s to t = 6.8s) . At* $t = 9$ *secs, the object is moved along a planned path to its goal location.*

Figure 6.25 illustrates the case when two arms must cooperate to grasp and manipulate the object. Notice the intercept trajectories (approximately $t = 4\ s$ to $t = 4.8\ s$) bringing each arm to the corresponding grasp location in the object (symmetrically at $\pm$15cm about the object center). The arms wait for each other until they are both ready to descend ($t = 5\ s$). Also note that while the grasps are consolidating ($t = 5.5\ s$ to $t = 6.5\ s$), the state of both arms' grippers accurately track the position and velocity of the corresponding grasp locations within the object. At time $t = 6.5\ s$ both grasps are consolidated and the arms cooperate to lift the object.

The individual-arm FSMs allow the arms to be independently commanded. The arms can be used to capture two different objects *simultaneously* as illustrated in Figures 6.26, 6.27, and 6.28.

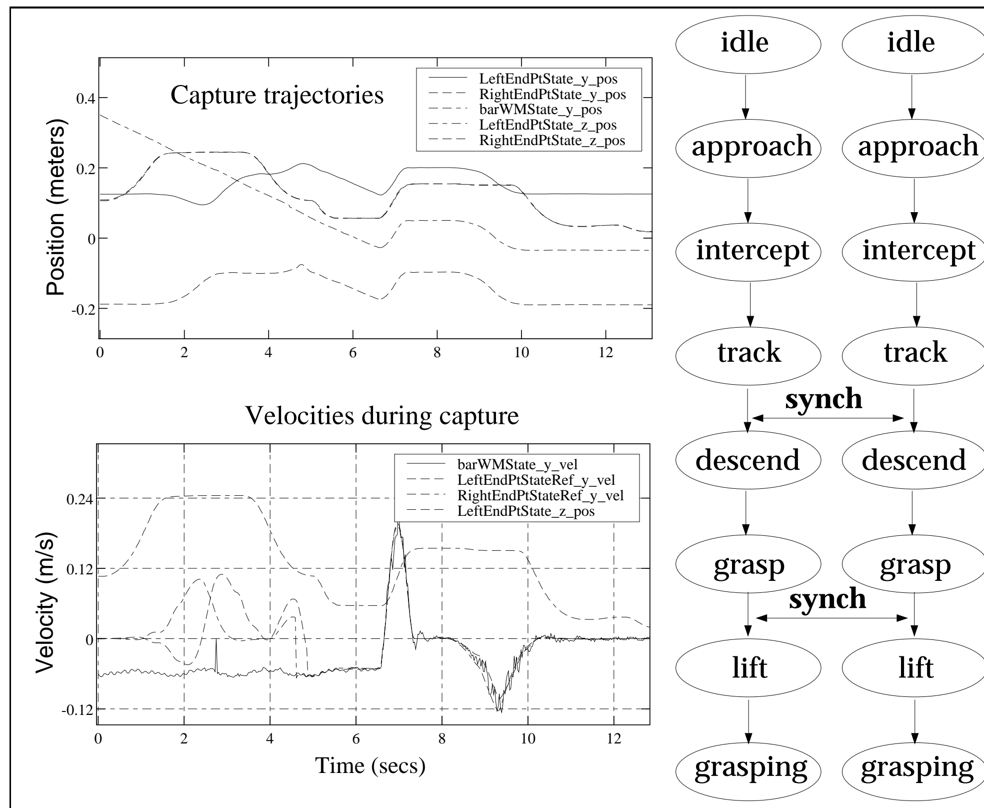


Figure 6.25: **Experimental dual-arm capture of moving object**

In response to a planner command, a single object is picked from the conveyor belt with two arms and subsequently delivered. The two arms synchronize at $t = 5$ secs, and then grasp and lift the object simultaneously. Again both arms match velocities with the object before the final descent and grasp is performed. The object is moved to the goal position at $t = 7$ seconds.

The sequence of pictures in Figure 6.29 illustrates the capture of two consecutive objects from the conveyor (using one hand per object). Figure 6.28 illustrates the use of both arms to *simultaneously* capture two different objects (one with each arm), while Figure 6.30 illustrates the use of two arms to *cooperatively* capture a single object.

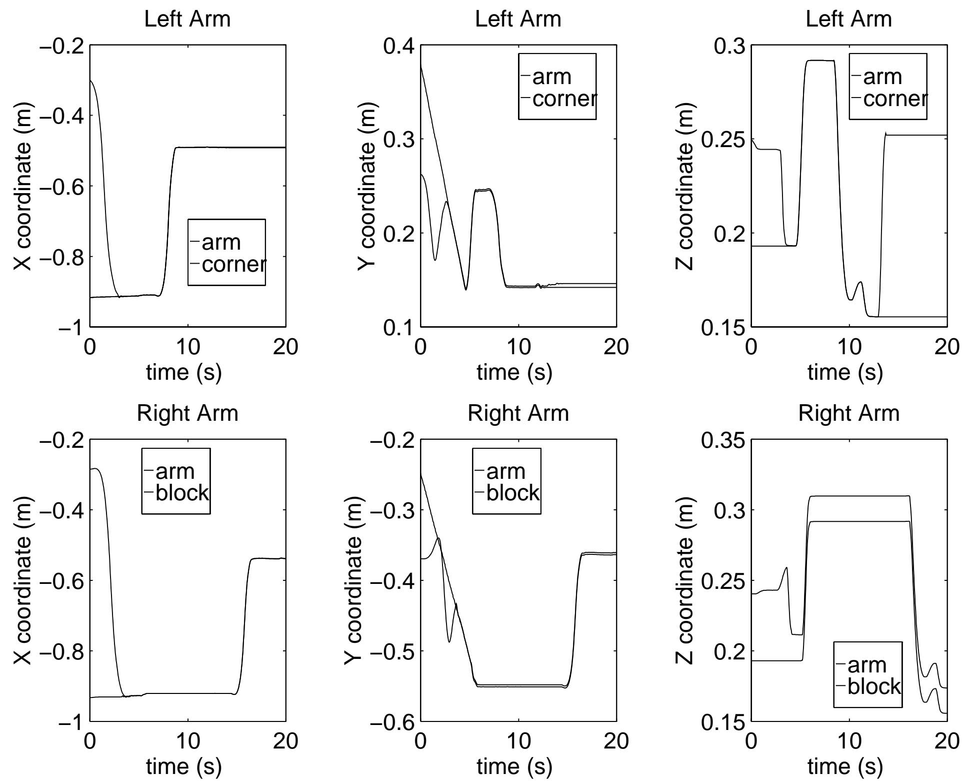


Figure 6.26: **Experimental simultaneous multiple-object capture**

These plots illustrate the simultaneous capture of two objects, each by a different arm. Each arm proceeds independently under the strategic control of the corresponding arm-FSM. The discrepancy between the Z coordinate of the right arm and the (grasped) "block" object is simply due to grasp location being offset in Z with respect to the body frame.

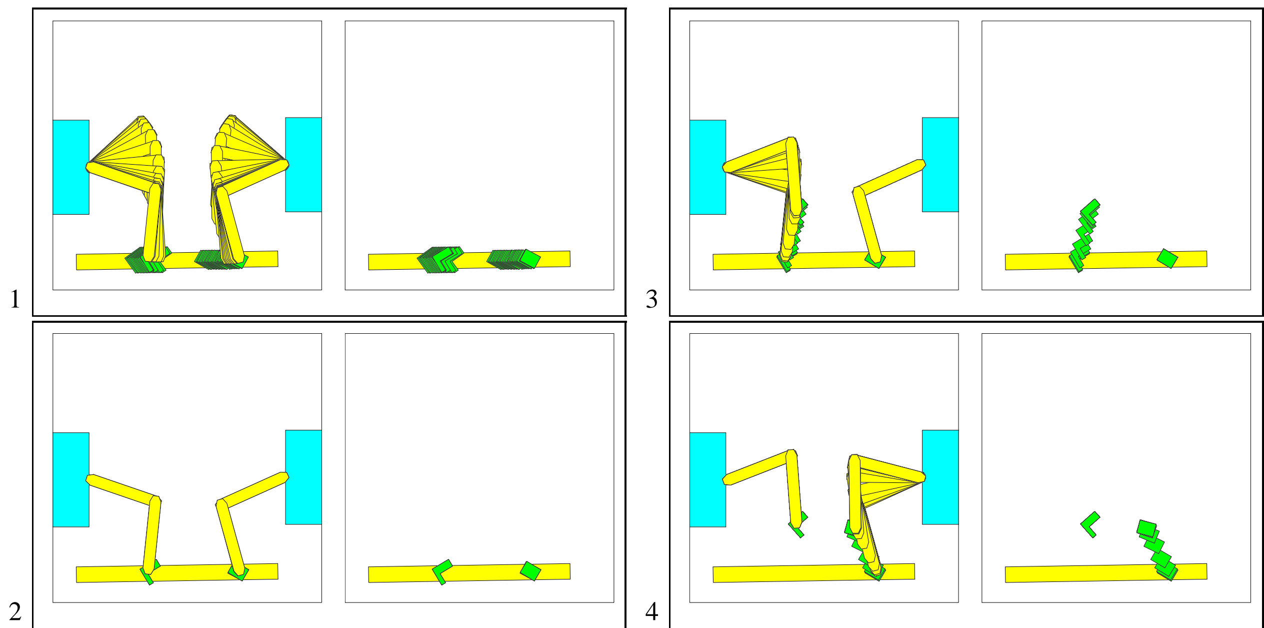


Figure 6.27: **Animation of experimental data of simultaneous multiple-object capture**

This figure animates the data presented in Figure 6.26.

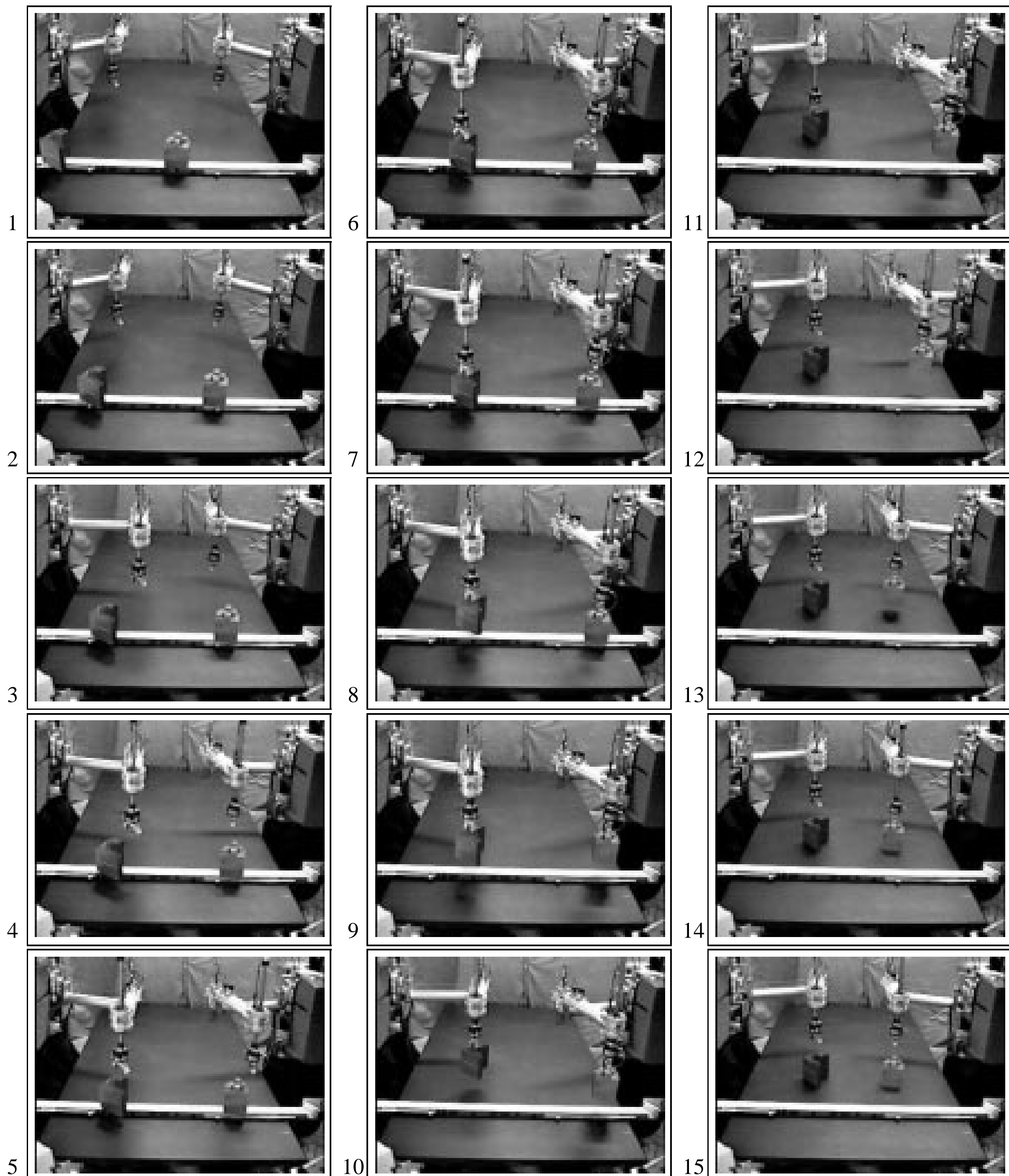


Figure 6.28: **Photographic sequence of system picking two objects from the conveyor simultaneously**

This sequence illustrates the use of two arms to simultaneously acquire two objects from the conveyor. This capability is enabled by the strategic control layer (by using a dedicated FSM program to supervise each arm), and can be extended to any number of arms in the workcell.

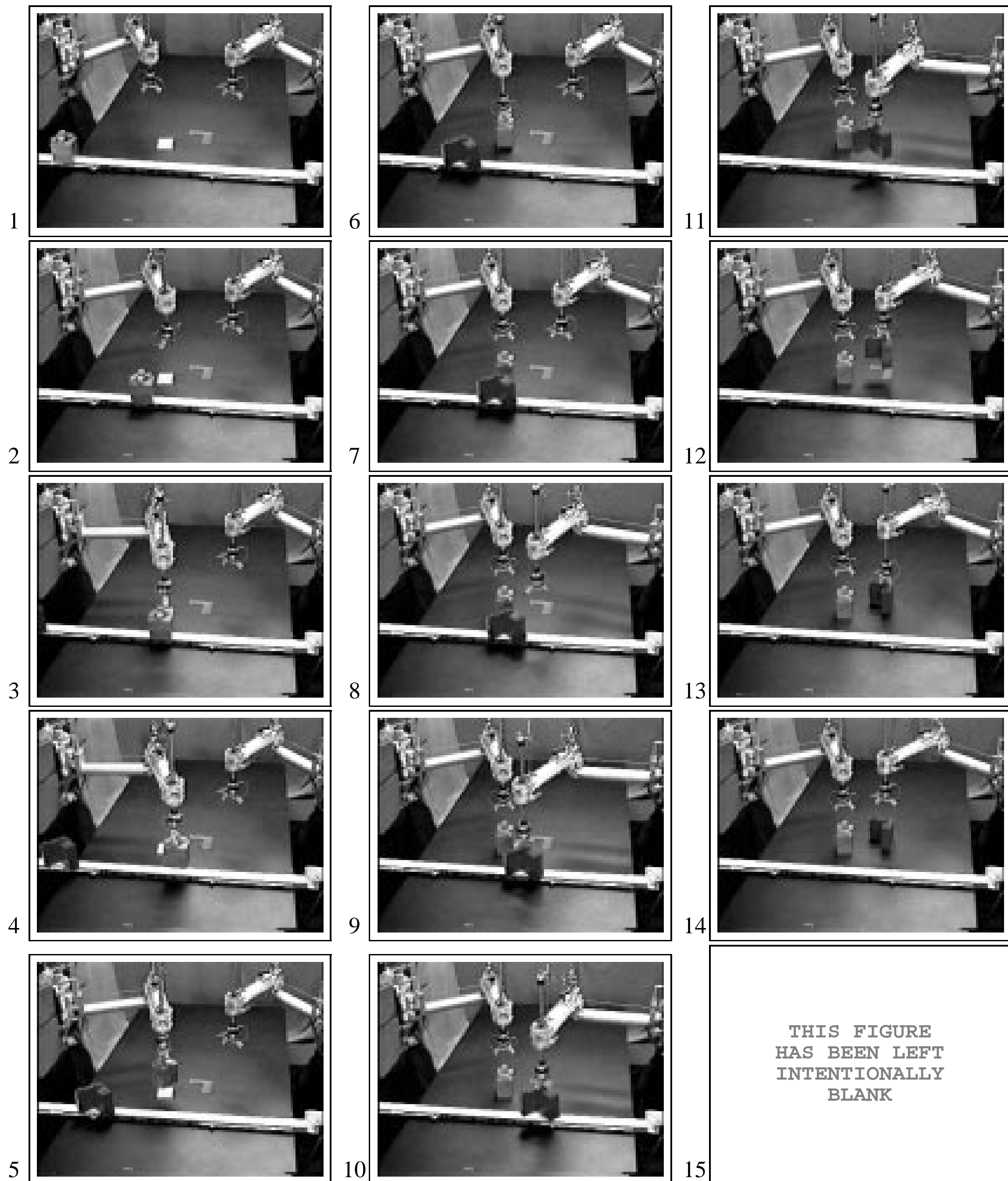


Figure 6.29: **Photographic sequence of system picking two objects from the conveyor**

This sequence of fourteen images has been obtained by digitizing video filmed during the operation of the system. The positions, velocities and accelerations of the manipulators are carefully controlled to match those of the object to be picked so that its motion is not disturbed suddenly. This delicate transition can be seen in the data (from a different capture) presented in Figure 6.24.

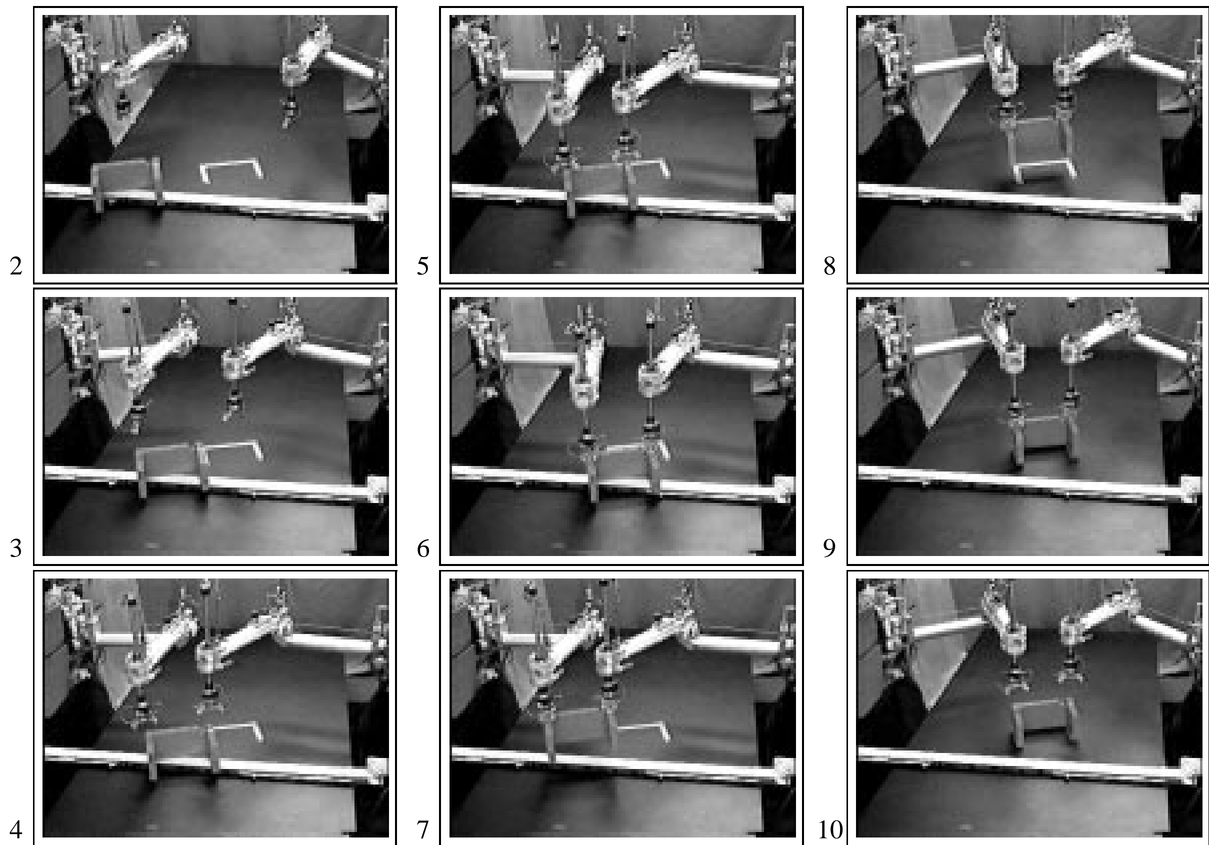


Figure 6.30: **Photographic sequence of system using two arms to pick the same object from the conveyor**

Sequence illustrating the use of two arms cooperatively to pick the same object from the conveyor. In addition to precise tracking of the respective grasp position by each arm, both arms need to synchronize in their grasp and lift operation.

6.8 Summary

This chapter has described the hierarchical-workcell control subsystem composed of four layers: Joint-Control, Arm-Control, Object-Control and Strategic Control. Each layer of the hierarchy addresses one control aspect and presents a simplified interface to the layer above. The joint-control layer uses joint-torque loops to compensate for joint flexibility, motor cogging, and other non ideal features. The joint-control layer allows the arm-control layer to treat the motors as ideal torque sources that apply their torque directly to the rigid links. The arm-layer uses an inverse dynamics (computed-torque) controller to implement an arm-endpoint-impedance controller that allows the object-control layer to command the arms by specifying applied endpoint forces and reference trajectories for the arm end effectors (joint trajectories must also be specified to go through kinematic singularities). The object-control layer uses an object-impedance controller to allow the strategic-control layer to control the motion of the object directly (by specifying object trajectories, impedances, and remote-center of compliance). Finally, the strategic-layer coordinates the motion of the two arms, reconfigures the dataflow, and modifies control parameters so that the workcell can be controlled by sending the commands specified in the Robot Interface (system-command interface of Chapter 3).

In the process of developing the hierarchical control system, several contributions and enhancements to previous approaches have been presented, including: (1) configuration-based merging of operational-space and joint-space controllers to achieve stable transition through kinematic singularities, (2) scalable-strategic control layer for the workcell, allowing additional manipulators to be easily added, and (3) development of strategic programs for the workcell enabling concurrent–as well as cooperative–arm motions (e.g. picking up multiple objects from the conveyor simultaneously, picking objects from the conveyors using two arms cooperatively).

Chapter 7

On-Line Trajectory Generation

This chapter presents a computationally-efficient algorithm to generate reference trajectories from a geometric description of the desired paths for the arm or objects in the workcell. This algorithm allows external subsystems to command motions in the workcell by specifying the path to be followed geometrically (e.g. by giving a sufficiently fine set of "through points"). These geometric paths will be converted into trajectories that follow the path as quickly as possible, within the dynamic constraints of the system (actuator torque limits, externally imposed object-acceleration limits etc.). Since the system must operate on-line, both the efficiency of the final trajectory, and that of the computational process itself must be placed on equal footing to evaluate system performance. Moreover, in a real-time environment, predictability of the computational delay of an algorithm is often more important than best-case or even average-case performance. The algorithm presented here is both computationally efficient and predictable. This is accomplished by introducing new proximate-optimal constraints (more restrictive than the dynamic constraints of the system) that may produce sub-optimal trajectories, but in return, the algorithm is guaranteed to execute in a time that is linear with respect to the length of the path. The linear scale factor depends on the specific dynamic equations of the system. Hence, this constant factor can be characterized á priori for a given system.

7.1 The Role of Trajectory Generation within the Workcell

The high-level command of a robotic workcell from a planning subsystem requires a format for specifying the movements of the arms and objects in the workcell. Many of the planners in the literature (and the ones used in this research) can take into account the kinematics of the system, but

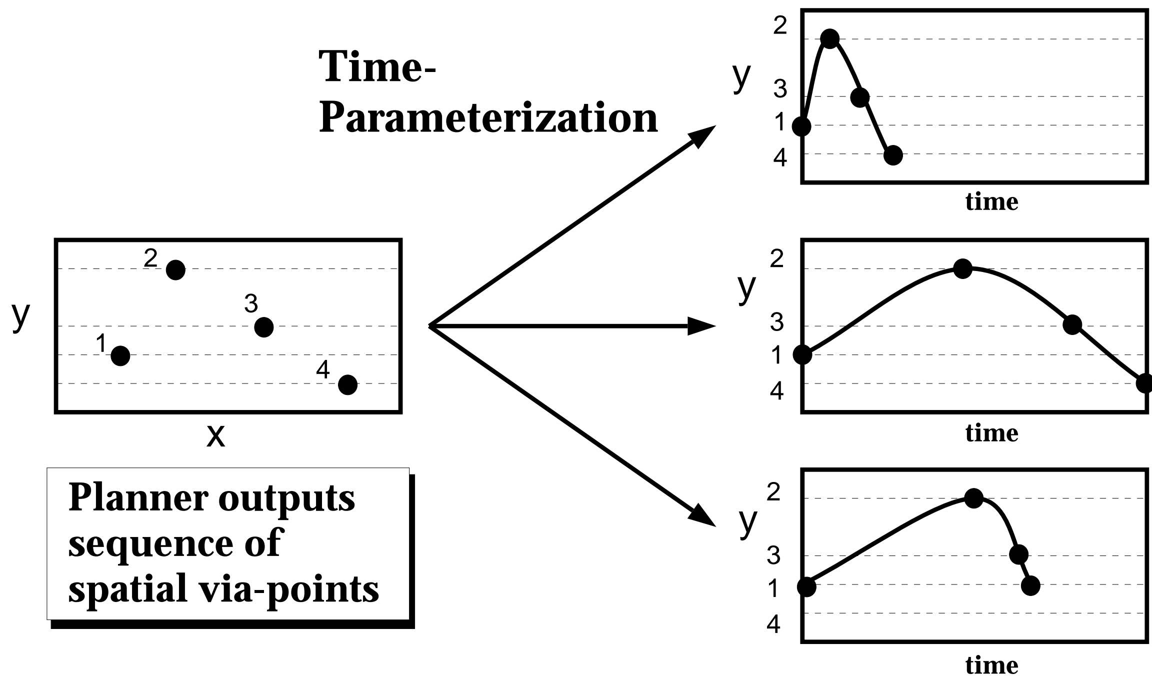


Figure 7.1: **The Time-Parameterization Process**

Time parameterization transforms a geometric path (given for example as a sequence of spatial via-points) into a reference trajectory that specifies positions, velocities, and accelerations as a function of time. Many possible trajectories can be created that traverse the same sequence of via-points. The three plots on the right illustrate three possible time-parameterizations traversing the same set of via-points. The time-parameterization process usually minimizes some performance index (such as total travel time) subject to the constraints of the system (e.g. velocities, accelerations, torque limits).

not its dynamics. The generated paths are *geometric* in nature. A geometric path specifies the path to be followed in space by giving, for instance, a sequence of via-points (a.k.a. through-points or way-points) in a space that fully describes the motion of the arms (or the manipulated objects). These via-points can, for example, represent joint-configurations, or for a non-redundant manipulator, positions of the operational point in space. In any case, the control system must ultimately be provided with a reference trajectory to track. This trajectory contains not only positions, but also velocities and accelerations as a function of time. The process of transforming geometric paths into trajectories is known as *time parameterization* of the geometric path, and is depicted pictorially in Figure 7.1.

The need for path specification and automatic trajectory generation is not restricted to the command of robotic workcells. Rather it is present in many other robotic system such as mobile robots and underwater vehicles, whenever complete paths need to be specified to the system[1].

Time parameterization of an á priori -obtained geometric path is not the only way to obtain a reference trajectory for the workcell control system. Several other approaches are discussed in section 7.2. However, it is one of the most popular due to its flexibility, intuitive appeal, conceptual simplicity, and speed.

The need for on-line operation of the workcell in a dynamic environment, combined with the fact that it must be commanded from a geometric planner, and operate in a not-completely-structured environment, introduces several requirements for the trajectory generation algorithm:

- The algorithm must be computationally efficient and be able to generate trajectories that fully exploit (but do not exceed) system capabilities. Since the goal is to perform on-line planning and execution of tasks, we are willing to trade off strict time optimality (speed of the final trajectory) in exchange for a computationally more efficient algorithm.

- The algorithm running time must be predictable for a given geometric path. This allows the different subsystems to anticipate and account for the computational delay in the time-parameterization process.

- The algorithm must provide a mechanism for "patching" ongoing trajectories, i.e., modifying parts of the path that lie ahead of the current trajectory. This allows the system to adapt to changes (such as a better estimate of the intercept position of a moving object) that may be detected after the trajectory was initiated.

In addition, there are several practical requirements imposed by the architecture and system interfaces, particularly because subsystems may operate remotely:

- The algorithm must accept geometric paths described as discrete sequences of via-points (through-points). Via-points sequences are a bare-bones, minimalist representation of the information that typical geometric planners are able to provide. Deciding on some ad-hoc continuous or even differentiable representation as an input to the algorithm would impose unnecessary constraints on the planning subsystem.

[1]For example, a tele-operated system has no need for time-parameterization because the reference state is generated directly from the motions of the tele-manipulation mechanism.

- The algorithm must produce a continuous description of the trajectory, from which position, velocities, and accelerations can be inferred at any sample rate. Using a continuous representation rather than a sequence of position, velocity, and acceleration states at some sample rate isolates the trajectory-generation algorithm from the control subsystem. In fact, the control subsystem may have sample rates which may be unknown to the subsystem performing trajectory-generation (or the controller may be be multi-rate). A continuous representation can also be described with fewer parameters (e.g. a few spline coefficients) rather than a much denser sequence of states at the controller sample rate. A compact description is advantageous when communicating with a controller over a link with limited bandwidth.
- The trajectory must have limited jerk. Again, the trajectory generator may be used for a variety of systems, on some of which jerk could excite flexibility or other system dynamics[2].

After providing a review of the related approaches in the next section, the rest of the chapter will describe a proximate-optimal trajectory-parameterization algorithm developed to address the above issues, and present experimental results to evaluate its computational efficiency (linear with respect to path length) and predictability. This algorithm is currently used for several other projects ranging from free-floating space robots [179] to under-water vehicles [104, 195], and has been made available to the robotics community as a ControlShell component [144].

7.2 Literature Review: Trajectory Generation

Current industrial robots do not employ any automated means of time-parameterizing geometric paths. Rather, many robot programming languages (such as VAL-II) provide facilities for low-level trajectory definition and tuning based on acceleration profiles. With this approach, the burden of tuning those parameters is left to the applications programer or systems integrator. Typically, trajectory tuning is performed manually (by trial and error) in an attempt to find the fastest trajectory that can be achieved with reasonable tracking errors (which are related to motor saturation and controller bandwidth). This approach is both difficult and time-consuming, and only applicable when the programming is done off line (i.e. the paths are known ahead of time and the robot will execute them over and over). Given that trajectory tuning is in the critical path of the system integration, it is surprising that industrial practice does not incorporate some of the established research results described below.

[2] In fact the arms in the workcell contain exaggerated joint flexibility that was designed for earlier research [131].

Given the perceived need for an automated method of generating reference trajectories for the system, it is not surprising that several different approaches have been proposed over the last decade. These approaches can be roughly classified into *decoupled, coupled, reactive*, and *hybrid*.

Time Parameterization of a Geometric Path (Decoupled Approach) The *decoupled* approach breaks the overall problem in two parts: First a *geometric*[3], collision free, path is obtained using a path planner or some other means. This *geometric* path can be described as a continuous function of some parameter "s" (usually path length): $\mathbf{q} = \mathbf{f}(s)$ with, $\mathbf{f} : [0, s_f] \in \mathcal{R} \rightarrow \mathcal{C}$, where $\mathcal{C}$ is the configuration space of the robot. Given the *geometric* path, the next step is to obtain a time parameterization by finding the time history $s(t)$ that minimizes a pre-specified performance index subject to the dynamic constraints. This latter stage does not change the geometric path, hence the name decoupled. The algorithms presented by Bobrow, Dubowsky and Gibson [72], Shin and McKay [164, 166, 165], Singh and Leu [174], as well as the one presented in this chapter all belong to this class. The decoupled approach is conceptually simple and well suited to interface with standard planning subsystems (which can generate the geometric paths). However, the computational complexity of these algorithms is such that to the author's knowledge, they have always been used off-line. Sacrificing strict optimality in order to obtain more efficient algorithms has already been suggested by Wen and Desrochers [196] and Butler and Tomizuka [25]. Other authors have focused on increasing the efficiency of the optimal methods. For instance, Slotine and Yang [176] exploit the geometric properties of the constraint equations to generate more efficient parameterization algorithms. Several of these approaches will be described in detail once the problem is formulated and the notation introduced.

Combined Path-Planning and Time Parameterization (Coupled Approach) The *coupled* approach, attempts to obtain a collision-free path that is also optimal with respect to some performance index without going through an intermediate geometric path. That is, it solves both the path-planning and the trajectory-generation problems simultaneously. This is the most general approach. Since it taken into account the shape of the path during the optimization process, the coupled approach will (theoretically[4]) yield lower values of the performance index than the decoupled approach. However, its computational complexity (recall that the geometric path planning problem is itself PSPACE-hard) is substantially greater than the decoupled approach. Moreover, the planner and

[3] The name geometric is used to stress the fact that time is not involved.

[4] In practice this will depend on the approximations made. Many implementations of the coupled approach restrict the search space of possible geometric paths.

time-parameterization subsystems must be developed together which makes the approach less modular and precludes the use of already existing planners. This approach has been proposed by Gilbert and Johnson [52], Bobrow [20], Shiller and Dubowsky [162, 163], etc.

Reactive Methods. A different approach is to never generate a path (or trajectory) explicitly. Rather, the desired state of the system is computed as a function of the current measured state and some external "virtual potential field". For instance the potential-function methods presented by Khatib [80] generate reference states by solving the system dynamics, assuming that a virtual potential function (which depends on the system state: position of manipulators, obstacles etc.) is acting on the manipulator. This method can be very reactive to environmental changes. However, it becomes quite difficult to specify the desired (or pre-planned) system behaviour (and hence use goal-oriented planning). Several authors have investigated the construction of potential fields that can be used to guide the robot along a pre-specified path (see Rimon [147] and Arkin [13]). These approaches typically cannot guarantee that actuator limits are not exceeded because the potential function is independent of the current velocity and acceleration of the robot.

Hybrid Methods. Other proposals have been made that combine the decoupled and reactive methods. For instance Quinlan [138, 137] generates a geometric path using standard planners, and then subjects the path to potential forces representing obstacles in the environment. The resulting geometric path is never time-parameterized. Rather, an instantaneous decision on whether the robot should accelerate, decelerate, or coast is made at each time step based on whether the robot can be safely brought to a stop. This approach is computationally expensive if the robot operates on in a region of large velocity to acceleration ratio (the case of free-flying space robots), because for each time step, the entire remaining trajectory may require re-parameterization.

For this research, the decoupled approach was selected for its flexibility and ability to interface to planning subsystems (through the pre-specified system-command interfaces). The remaining sections will describe the decoupled approach in more depth, review several of the proposed solutions in more detail, introduce the efficient proximate-optimal solution proposed in this research, and present experimental results on the performance of the new proposed solution.

7.3 Problem Formulation and Known Results

The decoupled-optimal-time-parameterization (DOTP) problem has been extensively described in the literature [72, 164, 30]. In order to introduce the notation necessary for the description of the proximate-optimal algorithm, and compare it to the existing approaches, a summary of the problem formulation and some other well-known facts is presented below.

The dynamics of a robot, assuming no friction can be written as [38]:

$$\mathbf{M}(\mathbf{q})\ddot{\mathbf{q}} + \mathbf{B}(\mathbf{q})[\mathbf{q}, \mathbf{q}] + \mathbf{C}(\mathbf{q})[\mathbf{q}^2] + \mathbf{g}(\mathbf{q}) = \boldsymbol{\tau} \tag{7.1}$$

Where $\mathbf{q}$ is the vector of generalized coordinates, $\mathbf{M}$ is the mass matrix, $\mathbf{B}$ is the $n \times (n-1)$ matrix of coriolis terms, $\mathbf{C}$ is the $n \times n$ matrix of centripetal terms, $\mathbf{g}(\mathbf{q})$ is the $n \times 1$ matrix of gravity terms, and $\boldsymbol{\tau}$ is the torque vector.

A *geometric* path can be fully described as a function $\mathbf{q} = \mathbf{f}(s)$ where the parameter "s" is in some range: $s \in [s_0, s_f]$. Typically the function $f(s)$ is required to have continuous second derivatives. It is well known that, given this description, the time evolution of the parameter $s = s(t)$ completely determines the full trajectory of the robot, and therefore, the required torques. In other words, given $s = s(t)$ equations (7.2) below, and (7.1) can be used to obtain $\mathbf{q}(t)$, $\mathbf{q}(t)$, $\ddot{\mathbf{q}}(t)$, and $\boldsymbol{\tau}(t)$.

$$\mathbf{q}(t) = \mathbf{q}(s(t)) \quad \Rightarrow \quad \mathbf{q} = \mathbf{f}_s\dot{s} \quad \Rightarrow \quad \ddot{\mathbf{q}} = \mathbf{f}_s\ddot{s} + \mathbf{f}_{ss}\dot{s}^2 \tag{7.2}$$

To formulate an instance of the DOTP problem, we also need a performance index and a set of constraints. The performance index $\mathcal{J}$ is a functional of the trajectory history: $\mathcal{J} = \mathcal{J}[\mathbf{q}, \mathbf{q}, \ddot{\mathbf{q}}, \boldsymbol{\tau}]$. Again using (7.2) and (7.1) we can write $\mathcal{J} = \mathcal{J}[s, \dot{s}, \ddot{s}]$. For example, the total travel time is a common performance index that can be written as: $\mathcal{J} \overset{\text{def}}{=} t_f = \int_{s_0}^{s_f} \frac{ds}{\dot{s}(s)}$.

Similarly, the velocity, acceleration, and torque constraints are specified as a set of inequalities $\boldsymbol{\phi}(\mathbf{q}, \mathbf{q}, \ddot{\mathbf{q}}, \boldsymbol{\tau}) \leq \mathbf{0}$, which can be written as $\boldsymbol{\phi}(s, \dot{s}, \ddot{s}) \leq \mathbf{0}$.

The DOTP problem is the search for the time parameterization $s = s(t)$ with $s(t) \in \mathcal{C}^2[s_0, s_f]$[5] that minimizes the performance index $\mathcal{J}[s, \dot{s}, \ddot{s}]$ subject to the constraint $\boldsymbol{\phi}(s, \dot{s}, \ddot{s}) \leq \mathbf{0}$. Notice that the á priori specification of the geometric path has simplified the problem of minimizing the performance index by replacing the search for vector-valued control history $\boldsymbol{\tau}(t)$ by that of the scalar valued function $\ddot{s}(t)$.

[5]The notation $\mathcal{C}^n[a, b]$ denotes the set of functions with continuous n$^{\text{th}}$ derivative in the interval [a,b].

The approaches in the literature differ in their selection of performance index $\mathcal{J}$, the nature of the constraint $\boldsymbol{\phi} \leq \mathbf{0}$ imposed, and the method used to solve the optimization problem as summarized in Table 7.1.

reference	*perf. index*	*constraints*	*method*	*complexity*
Luh el al. [97]	total travel time	constant bounds on $\dot{q}$ and $\ddot{q}$	modified approximate programming	$\mathcal{O}(L)$/iter.
Bobrow el al. [72]	total travel time	$\boldsymbol{\tau}[q, \dot{q}]$ and $\dot{q}[q]$	direct integration	$\mathcal{O}(L^2)$
Shin el al. [167]	total travel time	$\boldsymbol{\tau}[q, \dot{q}]$ and $\dot{q}[q]$	direct integration	$\mathcal{O}(L^2)$
Shin el al. [166]	general	general	dynamic programming	$\mathcal{O}(L)$/iter.
Singh el al. [174]	average of travel time and energy	$\boldsymbol{\tau}[q, \dot{q}]$ and $\dot{q}[q]$	dynamic programming	$\mathcal{O}(L)$/iter.
Pardo this chapter	total travel time	$\boldsymbol{\tau}[q, \dot{q}]$, $\dot{q}[q]$ and/or $\ddot{q}[q, \dot{q}]$	direct integration	$\mathcal{O}(L)$

Table 7.1: **Related approaches to the Decoupled-Optimal-Time-Parameterization Problem**

Comparison of approaches to the DOTP problem. The notation $\boldsymbol{\tau}[\mathbf{q}, \dot{\mathbf{q}}]$ indicates limits in the torque that may depend on the state $(\mathbf{q}, \dot{\mathbf{q}})$, "general" indicates support for a wide variety of possible performance indices or constraints (e.g. minimum energy, jerk etc.). In the complexity column, "L" stands for path length. For iterative algorithms that require convergence, we give the complexity per iteration. See sections 7.5.4 and 7.5.5 for a justification of the complexity values.

7.3.1 Minimum travel time with limits on actuator torque, acceleration and velocity

If the performance index is the total travel time, and the only constraints are on the maximum velocity, acceleration and torques, it has been shown in [72, 164] that the DOTP problem can be transformed into the simpler "Decoupled Minimum-Time Time-Parameterization Problem" (DMTTP) described in this section.

Using equations (7.2) and (7.1) we obtain:

$$\boldsymbol{\tau}(s,\dot{s}) = \mathbf{M}(s)(\mathbf{f}_s\ddot{s} + \mathbf{f}_{ss}\dot{s}^2) + (\mathbf{B}(s)[\mathbf{f}_s;\mathbf{f}_s] + \mathbf{C}(s)[\mathbf{f}_s^2])\dot{s}^2 + \mathbf{g}(s) \tag{7.3}$$

$$\boldsymbol{\tau}(s,\dot{s}) = \mathbf{m}(s)\ddot{s} + \mathbf{c}(s)\dot{s}^2 + \mathbf{g}(s) \tag{7.4}$$

$$\mathbf{m}(s) \stackrel{\text{def}}{=} \mathbf{M}(s)\mathbf{f}_s(s) \tag{7.5}$$

$$\mathbf{c}(s) \stackrel{\text{def}}{=} \mathbf{M}(s)\mathbf{f}_{ss}(s) + \mathbf{B}(s)[\mathbf{f}_s;\mathbf{f}_s] + \mathbf{C}(s)[\mathbf{f}_s^2] \tag{7.6}$$

The torque constraints can now be written as:

$$\boldsymbol{\tau}_{min}(\mathbf{q},\mathbf{q}) \leq \boldsymbol{\tau}(s,\dot{s}) \leq \boldsymbol{\tau}_{max}(\mathbf{q},\mathbf{q})) \iff \ddot{s}_{min}(s,\dot{s}) \leq \ddot{s} \leq \ddot{s}_{max}(s,\dot{s}) \tag{7.7}$$

$$\begin{aligned} \text{Where:}\quad \boldsymbol{\tau}_{min}(s,\dot{s}) &\stackrel{\text{def}}{=} \boldsymbol{\tau}_{min}(\mathbf{q}(s),\mathbf{q}(s,\dot{s})) \stackrel{\text{def}}{=} \boldsymbol{\tau}_{min}(\mathbf{f}(s),\mathbf{f}_s\dot{s})) \\ \boldsymbol{\tau}_{max}(s,\dot{s}) &\stackrel{\text{def}}{=} \boldsymbol{\tau}_{max}(\mathbf{q}(s),\mathbf{q}(s,\dot{s})) \stackrel{\text{def}}{=} \boldsymbol{\tau}_{max}(\mathbf{f}(s),\mathbf{f}_s\dot{s})) \\ \ddot{s}_{min}(s,\dot{s}) &\stackrel{\text{def}}{=} \max_{i=1\ldots N_{dof}} \{\delta^i_{min}(s,\dot{s})\} \\ \ddot{s}_{max}(s,\dot{s}) &\stackrel{\text{def}}{=} \min_{i=1\ldots N_{dof}} \{\delta^i_{max}(s,\dot{s})\} \end{aligned}$$

$$\delta^i_{min}(s,\dot{s}) \stackrel{\text{def}}{=} \begin{cases} \frac{1}{\mathbf{m}^i(s)}\left[\boldsymbol{\tau}^i_{min}(s,\dot{s}) - \mathbf{c}^i(s)\dot{s}^2 - \mathbf{g}^i(s)\right] & \text{if} \quad \mathbf{m}^i(s) > 0 \\ \frac{1}{\mathbf{m}^i(s)}\left[\boldsymbol{\tau}^i_{max}(s,\dot{s}) - \mathbf{c}^i(s)\dot{s}^2 - \mathbf{g}^i(s)\right] & \text{if} \quad \mathbf{m}^i(s) < 0 \\ -\infty & \text{if } \mathbf{m}^i(s) = 0 \end{cases} \tag{7.8}$$

$$\delta^i_{max}(s,\dot{s}) \stackrel{\text{def}}{=} \begin{cases} \frac{1}{\mathbf{m}^i(s)}\left[\boldsymbol{\tau}^i_{max}(s,\dot{s}) - \mathbf{c}^i(s)\dot{s}^2 - \mathbf{g}^i(s)\right] & \text{if} \quad \mathbf{m}^i(s) > 0 \\ \frac{1}{\mathbf{m}^i(s)}\left[\boldsymbol{\tau}^i_{min}(s,\dot{s}) - \mathbf{c}^i(s)\dot{s}^2 - \mathbf{g}^i(s)\right] & \text{if} \quad \mathbf{m}^i(s) < 0 \\ \infty & \text{if} \quad \mathbf{m}^i(s) = 0 \end{cases} \tag{7.9}$$

It is easy to see that using (7.2), any constraints in the acceleration $\ddot{\mathbf{q}}$, may be expressed in a similar form:

$$\ddot{\mathbf{q}}_{min}(s,\dot{s}) \leq \ddot{\mathbf{q}}(s) \leq \ddot{\mathbf{q}}_{max}(s,\dot{s}) \iff \ddot{s}_{min}(s,\dot{s}) \leq \ddot{s} \leq \ddot{s}_{max}(s,\dot{s}) \tag{7.10}$$

With the appropriate corresponding definitions for $\ddot{s}_{min}(s,\dot{s})$ and $\ddot{s}_{max}(s,\dot{s})$.

Therefore, constraints in the torques and accelerations may be combined into an inequality constraint of the form $\ddot{s}_{min}(s,\dot{s}) \leq \ddot{s} \leq \ddot{s}_{max}(s,\dot{s})$. Or using the fact that $\frac{d\dot{s}}{ds} = \frac{\ddot{s}}{\dot{s}}$, they can be

written as:

$$\begin{aligned} &\alpha_{min}(s,\dot{s}) \leq \frac{d\dot{s}}{ds} = \frac{\ddot{s}}{\dot{s}} \leq \alpha_{max}(s,\dot{s}) \\ &\alpha_{min}(s,\dot{s}) \stackrel{\text{def}}{=} \ddot{s}_{min}(s,\dot{s})/\dot{s} \quad \text{For } \dot{s} \neq 0 \\ &\alpha_{max}(s,\dot{s}) \stackrel{\text{def}}{=} \ddot{s}_{max}(s,\dot{s})/\dot{s} \quad \text{For } \dot{s} \neq 0 \end{aligned} \tag{7.11}$$

Furthermore, symmetric constraints in the velocity, if present, can be written as:

$$\begin{aligned} &-\mathbf{q}_{max}(s) \leq \mathbf{q}(s) = \mathbf{f}_s(s)\dot{s} \leq \mathbf{q}_{max}(s) \iff \dot{s} \leq \gamma(s) \\ &\gamma(s) \stackrel{\text{def}}{=} \min_{i=1 \ldots N_{dof}} \{ \frac{\mathbf{q}^i_{max}(s)}{|\mathbf{f}^i_s(s)|} \} \end{aligned} \tag{7.12}$$

We assume that the torque limits are such that the manipulator can hold its own weight at any point along the trajectory. This condition can be expressed as:

$$\forall s : \boldsymbol{\tau}_{min}(s,0) \leq \mathbf{g}(s) \leq \boldsymbol{\tau}_{max}(s,0) \iff \forall s : \alpha_{min}(s,0) \leq 0 \leq \alpha_{max}(s,0)$$

Borrowing the analogy introduced by [167], the constraints of equations (7.11) can be visualized as a "wedge" associated with each point $(s, \dot{s})$ in phase space. This "wedge" represents the range of allowed slopes of any trajectory which traverses that point in phase space ($\dot{s} = \dot{s}(s)$), and locally satisfies the constraints. Several authors [72, 167] have noted that for each value of s there are values of $\dot{s}$ for which the wedge closes (i.e. $\alpha_{min}(s,\dot{s}) > \alpha_{max}(s,\dot{s})$) meaning there is no phase space trajectory that traverses that point $(s, \dot{s})$ and satisfies the constraints. Figure 7.2 shows these constraints for an example path. Furthermore, in [167] the authors prove that under fairly general assumptions[6], the "allowed" region for $\dot{s}$ has the form $0 \leq \dot{s}_{max}(s)$. We can therefore combine this with (7.12) and write the combined torque, acceleration and velocity constraints in the form:

$$\begin{aligned} &0 \leq \dot{s} \leq \dot{s}_{max}(s) \\ &\alpha_{min}(s,\dot{s}) \leq \frac{d\dot{s}}{ds} \leq \alpha_{max}(s,\dot{s})) \end{aligned} \tag{7.13}$$

Where $\dot{s}_{max}(s)$ is defined so that:

$$0 \leq \dot{s} \leq \dot{s}_{max}(s) \quad \Rightarrow \quad \alpha_{min}(s,\dot{s}) \leq \alpha_{max}(s,\dot{s})$$

To summarize the previous discussion, for the case in which the performance index is minimum travel time, the original DOTP problem can be recast into the simpler DMTTP problem which takes the form:

[6]These results are valid for a manipulator modeled without friction. The only assumption made in the paper is that the torque limits have a dependency on the joint velocities $\mathbf{q}^i$ that is at most quadratic in them.

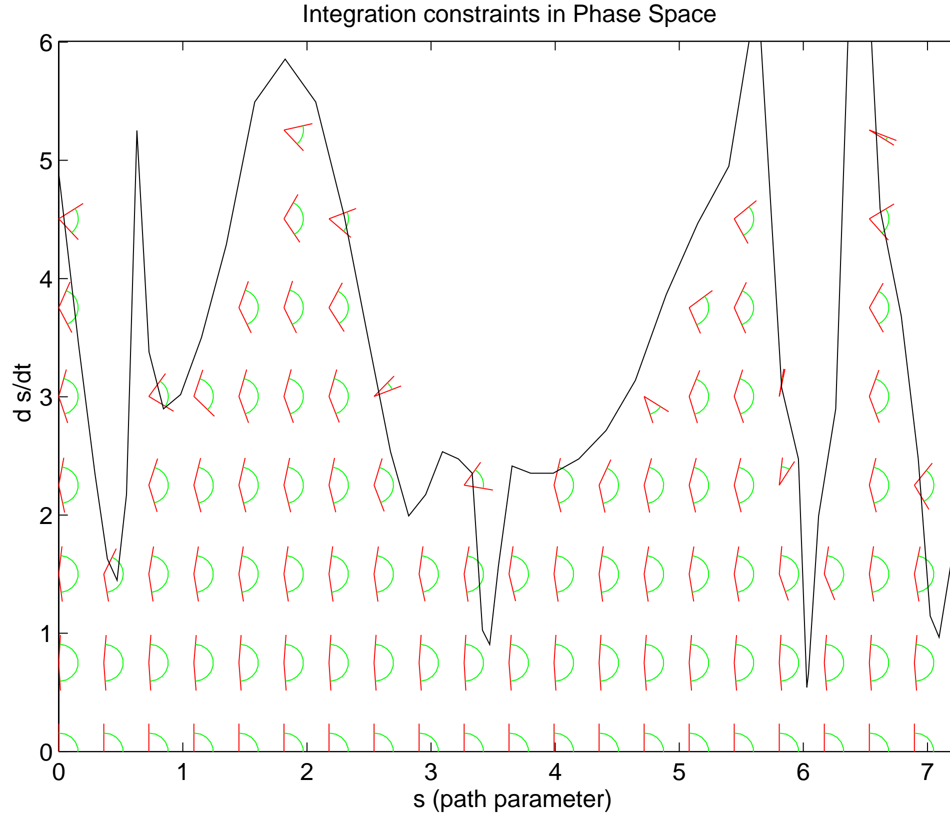


Figure 7.2: **Optimal time-parameterization constraints in phase-space**

For a given geometric path and manipulator, the velocity, acceleration, and torque limits map into two types of phase-space constraints: (1) an allowed region that verifies $\dot{s} \leq \dot{s}_{max}(s)$ and (2) "slope" limits $\alpha_{min}(s, \dot{s}) \leq \frac{d\dot{s}}{ds} \leq \alpha_{max}(s, \dot{s})$. At each point $(s, \dot{s})$ in phase-space, there is a "wedge" representing the range of allowed slopes of legal phase-space trajectories trough that point. The curve $\dot{s}_{max}$ separates the region of the phase-space for which there are no allowed phase-space trajectories because either the velocity limit is exceeded or the "wedge" "closes".

DMTTP problem: Given the functions $\alpha_{min}(s, \dot{s})$, $\alpha_{max}(s, \dot{s})$ and $\dot{s}_{max}(s)$ such that the following properties hold:

$$\forall s \in [s_0, s_f] \quad \dot{s}_{max}(s) \geq 0$$
$$\forall s \in [s_0, s_f] \quad \dot{s} \leq \dot{s}_{max}(s) \quad \Rightarrow \quad \alpha_{min}(s, \dot{s}) \leq \alpha_{max}(s, \dot{s})$$

Find the time parameterization $\dot{s}(s)$ for $s \in [s_0, s_f]$ that minimizes the performance index $\mathcal{J} = t_f = \int_{s_0}^{s_f} \frac{1}{\dot{s}(s)} ds$ subject to the constraints:

$$\dot{s}(s_0) = \dot{s}_0 \quad \text{and} \quad \dot{s}(s_f) = \dot{s}_f$$
$$\forall s \in [s_0, s_f] \quad 0 \leq \dot{s}(s) \leq \dot{s}_{max}(s)$$
$$\forall s \in [s_0, s_f] \quad \alpha_{min}(s, \dot{s}) \leq \tfrac{d\dot{s}}{ds} \leq \alpha_{max}(s, \dot{s})$$

Among the useful results associated with the DMTTP problem the following lemma–proven in [72]–will be specially useful in the description of the proximate-optimal algorithm.

Lemma 1 *Let* $\{[s_0, s_f], \dot{s}_0, \dot{s}_f, \alpha_{min}(s, \dot{s}), \alpha_{max}(s, \dot{s}), \dot{s}_{max}(s)\}$ *be an instance on the DMTTP problem and* $\dot{s}^*(s)$ *be its solution. Then for any function* $\dot{s}(s)$ *that satisfies the constraints of the problem, we have* $\dot{s}(s) \leq \dot{s}^*(s)$, $\forall s \in [s_0, s_f]$.

This lemma and the results that preceded it are used to prove that specific phase-space integration schemes will yield the optimal trajectory. In particular, the algorithms presented in [72, 167, 166] solve the DMTTP problem.

7.3.2 Case of velocity-independent torque limits

In the case where the actuator torque limits can be approximated as being velocity-independent (i.e. $\boldsymbol{\tau}^i_{max}(s, \dot{s}) = \boldsymbol{\tau}^i_{max}(s)$ for each degree-of-freedom "i"), the constraints in the phase-space slope can be simplified. Replacing $\boldsymbol{\tau}^i_{max}(s, \dot{s}) = \boldsymbol{\tau}^i_{max}(s)$ and $\boldsymbol{\tau}^i_{min}(s, \dot{s}) = \boldsymbol{\tau}^i_{min}(s)$ in equations (7.7) and (7.8), the slope limits of equation (7.11)

$$\alpha_{min}(s, \dot{s}) \leq \frac{d\dot{s}}{ds} = \frac{\ddot{s}}{\dot{s}} \leq \alpha_{max}(s, \dot{s})$$

Can now be written as:

$$\underline{\alpha}^i_1(s)/\dot{s} - \alpha^i_2(s)\,\dot{s} \leq \frac{d\dot{s}}{ds} \leq \overline{\alpha}^i_1(s)/\dot{s} - \alpha^i_2(s)\,\dot{s}, \quad \text{for} \quad i = 1, ..N_{dof} \tag{7.14}$$

The proximate-optimal algorithm uses this approximation to simplify the integration of the trajectory.

7.4 Formulation of the Proximate-Optimal Problem

This section presents the main contribution of this chapter: a non-iterative $\mathcal{O}(L)$[7] algorithm to find a proximate-optimal solution to the DMTTP problem. The algorithm gains its efficiency from

[7] L stands for the length of the path.

transforming the DMTTP problem into one with stricter constraints. The algorithm then calculates the optimal solution to this modified problem. In many cases these stricter constraints will result in trajectories that are not significantly slower (this has been observed empirically). Further research is needed to precisely characterize the performance loss with respect to the true time-optimal path.

The key idea to derive an $\mathcal{O}(L)$ algorithm is to find a criterion which allows the phase-space integration to be broken into several independent sections. As a side benefit, the resulting algorithm will be highly parallel. Note that an instance of the DMTTP problem (and hence its solution $\dot{s}^*(s)$, is completely determined by the boundary conditions $\{\dot{s}_0, \dot{s}_f\}$ and the constraint functions $\{\alpha_{min}(s, \dot{s}), \alpha_{max}(s, \dot{s}), \dot{s}_{max}(s)\}$. Therefore, if we knew in advance any point in the optimal path $\{s_d, \dot{s}^*(s_d)\}$ with $s_0 < s_d < s_f$, we could break the problem into two independent ones: first solve for $s_0 \leq s \leq s_d$ and then for $s_d \leq s \leq s_f$. In other words, the value $\dot{s}^*(s_d)$ provides the boundary condition at $s = s_d$ that allows the problem to be divided. Any point along the optimal phase-space trajectory, $\dot{s} = \dot{s}^*(s)$ can be used in this manner. Figure 7.4 shows one such phase-space trajectory. The algorithm is based in the following characterization of a subset of the points in the optimal trajectory. This characterization allows early identification of these points:

Theorem 1 *Let* $\{[s_0, s_f], \dot{s}(s_0), \dot{s}(s_f), \alpha_{min}(s, \dot{s}), \alpha_{max}(s, \dot{s}), \dot{s}_{max}(s)\}$ *be an instance of the DMTTP problem and* $\dot{s}^*(s_d)$ *its solution.*

Assume that:

$$\forall s \in [s_0, s_f] : \quad \dot{s} \leq \dot{s}_{max}(s) \;\Rightarrow\; \alpha_{min}(s, \dot{s}) \leq 0 \leq \alpha_{max}(s, \dot{s}) \tag{7.15}$$

Then the following property (see Figure 7.3) holds:

$$\forall s_1, s_2, s_d \in [s_0, s_f] : \quad \left.\begin{array}{l} s_1 < s_d < s_2 \\ \dot{s}_{max}(s_d) = \min_{s_1 \leq s \leq s_2}\{\dot{s}_{max}(s)\} \\ \dot{s}^*(s_1) = \dot{s}_{max}(s_d) = \dot{s}^*(s_2) \end{array}\right\} \;\Rightarrow\; \dot{s}^*(s_d) = \dot{s}_{max}(s_d)$$

Proof. It suffices to show that the phase-space trajectory defined by Equation (7.16) below satisfies all the constraints.

$$\dot{s}(s) = u(s) \stackrel{\text{def}}{=} \begin{cases} \dot{s}_{max}(s_d) & \text{for } s_1 < s < s_2 \\ \dot{s}^*(s) & \text{otherwise} \end{cases} \tag{7.16}$$

Once the above statement is proven, applying lemma 1, we see that $\dot{s}^*(s_d) \geq u(s_d) = \dot{s}_{max}(s_d)$ which combined with the constraint $\dot{s}^*(s_d) \leq \dot{s}_{max}(s_d)$ implies $\dot{s}^*(s_d) = \dot{s}_{max}(s_d)$.

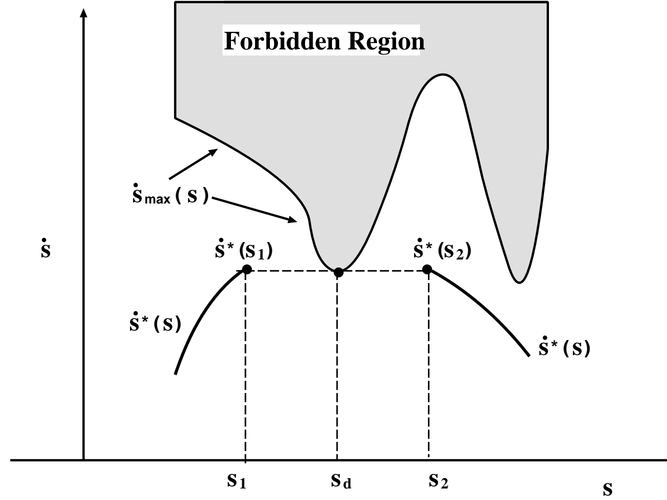


Figure 7.3: **Illustration of conditions of Theorem 1**

In view of its definition, we only need to show that $u(s)$ satisfies the constraints in the interval $]s_1, s_2[$. Now, in this interval, our hypothesis guarantees $\dot{s}(s) = \dot{s}(s_d) \leq \dot{s}_{max}(s)$, and since $\dot{s}(s)$ is constant $\frac{d\dot{s}}{ds} = 0$, and therefore, $\alpha_{min}(s, \dot{s}) \leq 0 = \frac{d\dot{s}}{ds} \leq \alpha_{max}(s, \dot{s})$.

Corollary 1 *The theorem holds even if we relax the equality* $\dot{s}^*(s_1) = \dot{s}_{max}(s_d) = \dot{s}^*(s_2)$ *to* $\dot{s}^*(s_1) \geq \dot{s}_{max}(s_d) \leq \dot{s}^*(s_2)$

Proof. Since $\dot{s}^*(s)$ is continuous and $\dot{s}^*(s_d) \leq \dot{s}_{max}(s_d)$, the intermediate value theorem guarantees that there are values $\tilde{s_1}, \tilde{s_2}$ such that $s_1 \leq \tilde{s_1} \leq s_d \leq \tilde{s_2} \leq s_2$ with $\dot{s}^*(\tilde{s_1}) = \dot{s}_{max}(s_d) = \dot{s}^*(\tilde{s_2})$. We can now apply the theorem to $\tilde{s_1}, \tilde{s_2}, s_d$.

The above theorem and its corollary provide a sufficient condition for a phase-space point $\{s, \dot{s}_{max}(s)\}$ to belong to the optimal phase-space trajectory $\dot{s}^*(s)$. This characterization only applies when we can guarantee that the pre-condition $\{\dot{s} \leq \dot{s}_{max}(s) \quad \Rightarrow \quad \alpha_{min}(s, \dot{s}) \leq 0 \leq \alpha_{max}(s, \dot{s})\}$ holds. Notice that this condition can always be enforced by redefining $\dot{s}_{max}(s)$ to be :

$$\dot{s}_{max}(s) \leftarrow \min \begin{cases} \dot{s}_{max}(s) \\ \min\{\dot{s} \mid (\alpha_{min}(s, \dot{s}) = 0) \text{ or } (\alpha_{max}(s, \dot{s}) = 0)\} \end{cases}$$

This new $\dot{s}_{max}$ is the more strict limit that we were referring to.

The more restrictive (proximate-optimal) constraint imposed on the allowable trajectories physically means that we require enough authority left in the actuators at every state in the trajectory

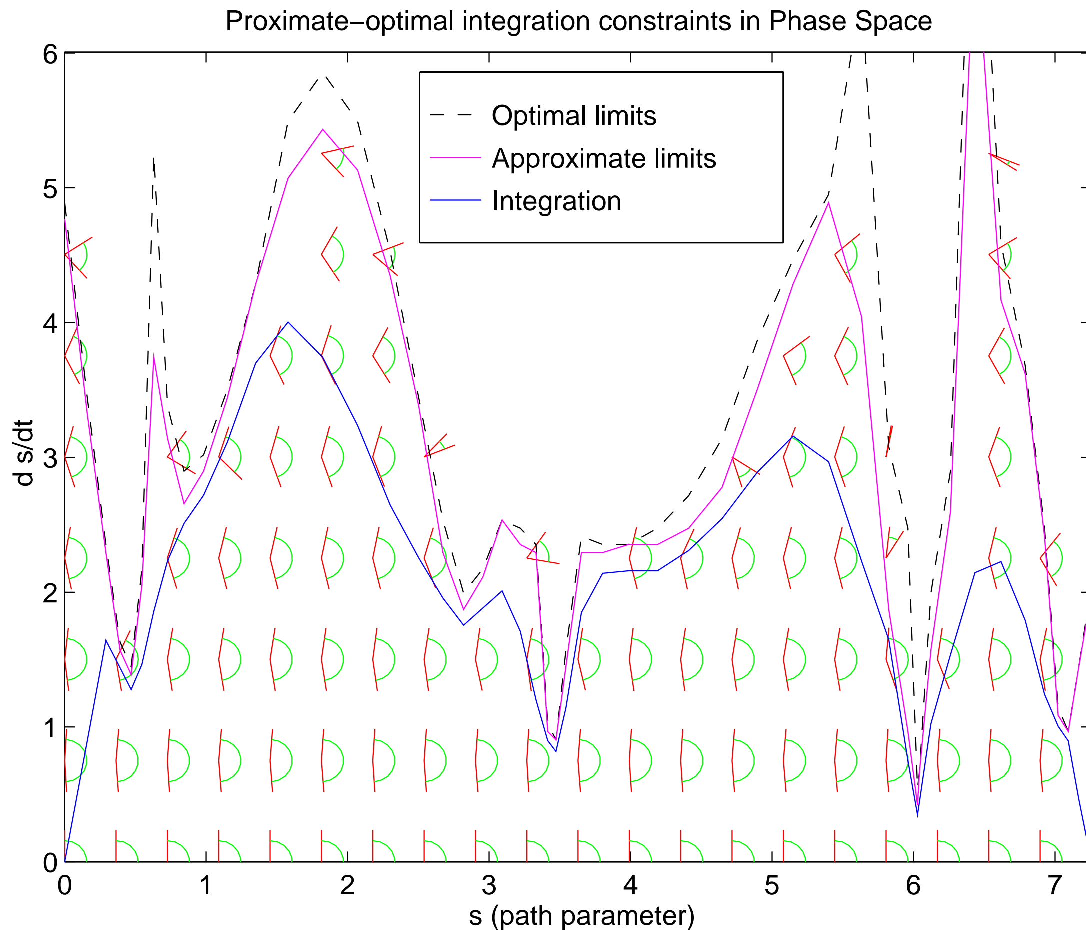


Figure 7.4: **Integration trajectory in phase space using proximate-optimal constraints**

This figure shows the proximate-optimal phase-space trajectory. Also shown are the constraints in the phase-space trajectories that correspond to the manipulator dynamic constraints for trajectories along the given geometric path. See Figure 7.2 for the interpretation of these constraints. The strict (optimal) limit occurs when the "wedge" "closes". The proximate-optimal algorithm imposes a more demanding requirement and disallows phase-space regions where the "wedge" does not contain the zero slope. Therefore the $\dot{s}_{max}(s)$ (approximate) curve above represents the points for which either the maximum or minimum allowed phase-space slope is zero.

that the system is able to change its speed along the trajectory in either direction: $\ddot{s} > 0$ (increase in speed), and $\ddot{s} < 0$ (decrease in speed)[8].

The fact that the optimal trajectory touches the boundary curve $\dot{s} = \dot{s}_{max}(s)$ at a finite number of points is also key to the algorithms presented in [72, 167, 176]. Reference [167] shows that

[8]This interpretation assumes the parameter "s" is either the path length or related by a strictly monotonic (increasing) function to the path length. This is the usual case.

for the case in which there are no limits in $\mathbf{q}$ (and therefore the boundary $\dot{s}_{max}(s)$ is given by the equation $\alpha_{min}(s,\dot{s}) = \alpha_{max}(s,\dot{s})$) the "switching points" satisfy the necessary condition $\frac{d\dot{s}_{max}(s)}{ds} = \alpha_{min}(s,\dot{s}_{max}(s))$. The algorithm presented in [176] exhaustively classifies these points and presents an efficient method to calculate them.

The proximate-optimal approach differs from the above in that thanks to the introduction of the extra requirement: $\{\dot{s} \leq \dot{s}_{max}(s) \quad \Rightarrow \quad \alpha_{min}(s,\dot{s}) \leq 0 \leq \alpha_{max}(s,\dot{s})\}$ we have obtained a *sufficient* condition. This is key to reducing the algorithmic complexity from $\mathcal{O}(L^2)$ to $\mathcal{O}(L)$ as discussed in sections 7.5.4 and 7.5.5.

7.5 The Proximate-Optimal Time-Parameterization Algorithm

This section describes the proximate-optimal algorithm and proves its correctness in the continuous domain. The discrete implementation is discussed in section 7.5.3.

7.5.1 Continuous-Domain version

Assume an instance of the DMTTP problem:

$$\{[s_0, s_f], \dot{s}_0, \dot{s}_f, \alpha_{min}(s,\dot{s}), \alpha_{max}(s,\dot{s}), \dot{s}_{max}(s)\}$$

that satisfies the requirement (7.15) in Theorem 1.

The algorithm to integrate $\dot{s} = \dot{s}_a(s)$ consists of three steps illustrated in Figure 7.5:

1. **Initialization**. Let

$$\mathcal{H} = \{s \in [s_0, s_f] \quad | \quad \dot{s}_{max}(s) \text{ is a local minimum}\}$$

$$\text{And, } s_l \leftarrow s_0,\ \dot{s}_l \leftarrow \dot{s}_0,\ s_r \leftarrow s_f,\ \dot{s}_r \leftarrow \dot{s}_f$$

2. **Integration**. Let

$$\begin{aligned}
&s_1 \leftarrow s_l,\ \dot{s}_a(s_1) \leftarrow \dot{s}_l,\ s_2 \leftarrow s_r,\ \dot{s}_a(s_2) \leftarrow \dot{s}_r \\
&\mathcal{H} = \mathcal{H} \cap\ ]s_l, s_r[\\
&s_d \leftarrow \{s_p \in \mathcal{H} \quad | \quad \dot{s}_{max}(s_p) = \min_{s\in\mathcal{H}}\{\dot{s}_{max}(s)\}\} \\
&\dot{s}_a(s_d) \leftarrow \dot{s}_{max}(s_d)
\end{aligned}$$

While $([s_1 < s_2] \ \wedge \ [(\dot{s}_a(s_1) < \dot{s}_a(s_d)) \vee (\dot{s}_a(s_2) < \dot{s}_a(s_d))])$ **Do :**

If $\dot{s}_a(s_1) \leq \dot{s}_a(s_2)$ integrate forward (i.e. increasing s_1) along the maximum acceleration curve:

$$\frac{d\dot{s}_a}{ds}(s_1) = \begin{cases} \alpha_{max}(s_1, \dot{s}_a(s_1)) & \textbf{if} \quad \dot{s}_a(s_1) < \dot{s}_{max}(s_1) \\ \min\{\frac{d\dot{s}_{max}}{ds}(s_1), \alpha_{max}(s_1, \dot{s}_a(s_1)\} , & \textbf{otherwise} \end{cases} \tag{7.17}$$

Else $\dot{s}_a(s_1) > \dot{s}_a(s_2)$ and we integrate backward (decreasing s_2) along the maximum deceleration curve:

$$\frac{d\dot{s}_a}{ds}(s_2) = \begin{cases} \alpha_{min}(s_2, \dot{s}_a(s_2)) & \textbf{if} \quad \dot{s}_a(s_2) < \dot{s}_{max}(s_2) \\ \min\{\frac{d\dot{s}_{max}}{ds}(s_2), \alpha_{min}(s_2, \dot{s}_a(s_2))\} , & \textbf{otherwise} \end{cases}$$

At any point in the integration, if any element of $\mathcal{H}$ falls outside the (changing) interval $]s_1, s_2[$, we eliminate it from $\mathcal{H}$ and recompute s_d if required.

The condition $s_1 = s_2$ indicates the integration between s_l and s_r has completed and we return. The condition $[\dot{s}_a(s_1) \geq \dot{s}_a(s_d)] \wedge [\dot{s}_a(s_2) \geq \dot{s}_a(s_d)]$ indicates we can break the problem into two independent ones and we continue with the next step.

3. **Separation**. Here we know that $[\dot{s}_a(s_1) \geq \dot{s}_d] \wedge [\dot{s}_a(s_2) \geq \dot{s}_d]$. We divide the problem into the following two:

   i) $s_l \leftarrow s_1,\ \dot{s}_l \leftarrow \dot{s}_a(s_1),\ s_r \leftarrow s_d,\ \dot{s}_r \leftarrow \dot{s}_d = \dot{s}_{max}(s_d)$

   ii) $s_l \leftarrow s_d,\ \dot{s}_l \leftarrow \dot{s}_d = \dot{s}_{max}(s_d),\ s_r \leftarrow s_2,\ \dot{s}_r \leftarrow \dot{s}_a(s_2)$

   And then invoke (recursively) the integration step on each one of the subproblems.

These steps are sketched in Figure 7.5. The actual process for an example path of one of the arms in the workcell can be seen in Figure 7.2. Each local minimum of the solution $\dot{s}_a(s)$ served as a decoupling point at some point during the integration.

7.5.2 Algorithm correctness

This section proves that the algorithm integrates the "optimal[9]" phase-space trajectory i.e. $\dot{s}_a(s) = \dot{s}^*(s)\ \forall s \in [s_0, s_f]$.

It suffices to show that, given boundary conditions $(s_l, \dot{s}_l)$ and $(s_r, \dot{s}_r)$ in the optimal trajectory, the integration step always generates points in the optimal trajectory. Once we show this, Theorem 1 guarantees that the separation step generates boundary conditions in the optimal trajectory.

[9] Optimal with respect to the modified constraints.

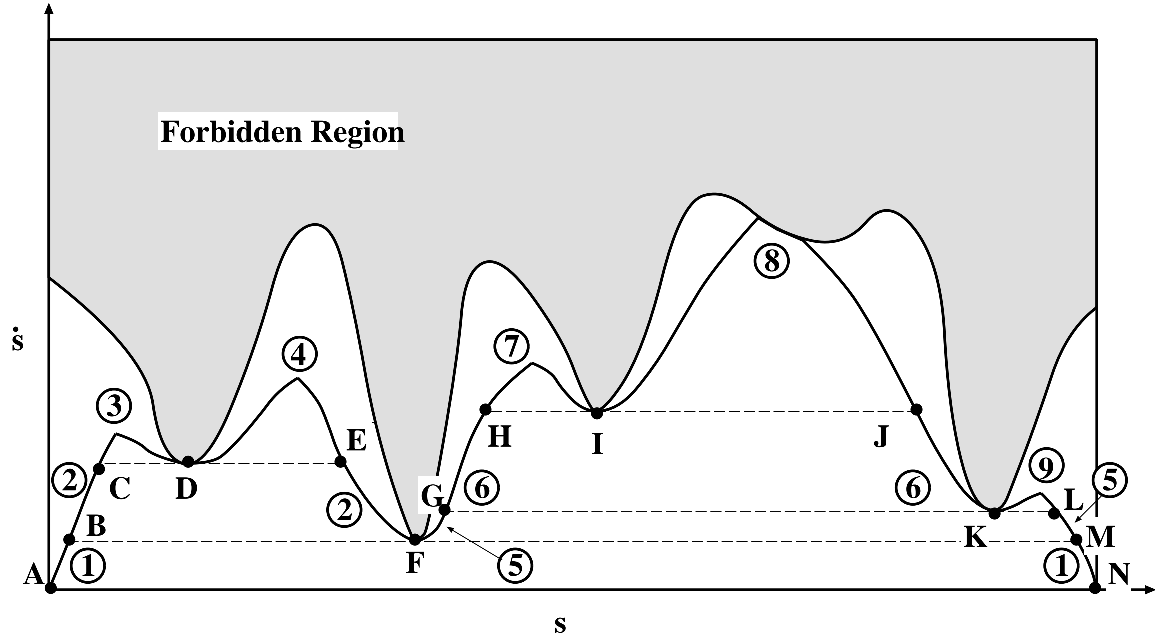


Figure 7.5: **Steps of proximate-optimal algorithm**

Phase-space illustration of the steps taken by the proximate-optimal algorithm to integrate a trajectory. Same numbered pieces are integrated simultaneously. Initially integration proceeds forward from A and backwards from N, keeping both branches balanced (1). As soon as the first local-minima is reached (F), the trajectory is broken into two independent pieces: B–F and F–M. B–F is integrated first (2), until another local minimum is reached at D. From here C–D is integrated (3) finishing that branch and D–E is integrated (4) completing the B–F branch. The right branch is integrated similarly after been separated at points K and I. Notice the possible existence of intervals where the integral follows the boundary of the forbidden region (8).

It is clear by construction that $\dot{s}_a(s) \geq \dot{s}'(s)$ for any function $\dot{s}'(s)$ that satisfies the constraints and the boundary conditions. In view of Lemma 1 it is sufficient to prove that $\dot{s}_a(s)$ itself satisfies the constraints.

Clearly by construction $\dot{s}_a(s) \leq \dot{s}_{max}(s)\ \forall s \in [s_l, s_r]$, so this constraint is always satisfied. This is because of the way the integration (see Equation (7.17)) changes the value of $\frac{d\dot{s}_a}{ds}$, to always be smaller than $\frac{d\dot{s}_{max}}{ds}(s)$ whenever $\dot{s}_a(s)$ intersects the boundary curve $\dot{s}_{max}(s)$.

So all we need to prove is that $\frac{d\dot{s}_a}{ds}$ does not violate the slope limits. This will proven through contradiction. Let $s_v \in [s_l, s_r]$ be the first value of s for which $s_a(s)$ violates the slope constraints. Given the way we choose $\frac{d\dot{s}_a}{ds}(s)$ to be either $\alpha_{max}(s_v, \dot{s}_a(s_v))$ or $\alpha_{min}(s_v, \dot{s}_a(s_v))$ whenever $\dot{s}_a(s_v) < \dot{s}_{max}(s_v)$, the only way the constraint $\alpha_{min}(s_v, \dot{s}_a(s_v)) \leq \dot{s}_a(s_v) \leq \alpha_{max}(s_v, \dot{s}_a(s_v))$ can be violated is at a point where $\dot{s}_a(s_v) = \dot{s}_{max}(s_v)$. Given how the integration (Equation (7.17))

chooses $\frac{d\dot{s}_a}{ds}(s_v)$, a slope violation can only occur if either during forward integration we have:

$$s_1 = s_v \quad \text{and} \quad \frac{d\dot{s}_a}{ds}(s_v) = \frac{d\dot{s}_{max}}{ds}(s_v) < \alpha_{min}(s_v, \dot{s}_a(s_v))$$

or during backward integration we have:

$$s_2 = s_v \quad \text{and} \quad \frac{d\dot{s}_a}{ds}(s_v) = \frac{d\dot{s}_{max}}{ds}(s_v) > \alpha_{max}(s_v, \dot{s}_a(s_v))$$

This will be shown to be impossible for the forward integration case; the backward integration case being analogous. First we must realize that the terminating conditions of the integration step and our selection of $\mathcal{H}$ and s_d guarantee that the invariant $\forall s \in]s_1, s_2[: \dot{s}_a(s_1) \leq \dot{s}_{max}(s) \leq \dot{s}_a(s_2)$ holds at all times during the integration. Then it is obvious that $\dot{s}_a(s_v) = \dot{s}_{max}(s_v)$, $s_1 < s_2$ and $\frac{d\dot{s}_{max}}{ds}(s_v) < \alpha_{min}(s_v, \dot{s}_a(s_v)) \leq 0$ violate this invariant. This contradicts our assumptions, and therefore there is no point s_v where the constraint is violated.

7.5.3 Discrete Implementation

This section presents how the algorithm is adapted to the fact that the input is a discrete sequence of via points $\{\mathbf{q}[k]\}_{k=0}^{N}$ (and not a continuous and differentiable path). The "standard" way to handle this situation would first fit a smooth curve through the via points, obtaining a continuous curve $\mathbf{q} = \mathbf{f}(s)$, and then apply the continuous version of the algorithm to obtain $\dot{s}(s)$. Once $\dot{s}(s)$ is known, we can integrate $t(s) = \int_0^s \frac{ds'}{\dot{s}(s')}$ and (numerically) invert the function to obtain $s(t_k)$. At the times of interest t_k, we can use $s(t_k)$, $\dot{s}_a(s(t_k))$ in combination with $\mathbf{f}_s$, $\mathbf{f}_{ss}$ and equations (7.1) and (7.2) to obtain whichever values of $\mathbf{q}$, $\mathbf{q}$, $\ddot{\mathbf{q}}$ and/or $\boldsymbol{\tau}$ are required by the controller.

This procedure has drawbacks in the context of on-line trajectory generation: It is computationally expensive, requiring $s(t_k)$ to be inverted at every sample. Requires high communication bandwidth between the time-parameterization and controller modules (the desired state must be produced at each sample time). And, it is not suited for the case in which the sample rate is not known to the time-parameterization algorithm. For these reasons an alternative approach has been adopted.

The alternative approach is based on the assumption that, in most cases, the via points are fairly sparse compared to the distance moved by the robot between consecutive samples. Via points are often generated in a way that reflects the geometric regularity of the path in the sense that portions of the path that are geometrically complicated (small curvature, rapid changes in the curvature etc.) require for their description, a greater density of via points per unit of path length[10]. In the proposed

[10]These qualitative statements need further research to be formalized

method, the first few steps are similar to the prototype "standard" method presented above, but the latter steps are significantly different:

1. Approximate the path length at each via point.

$$s[0] = 0 \; ; s[k+1] = s[k] + ||\mathbf{q}[k+1] - \mathbf{q}[k]|| \text{ , for } k = 0...N$$

   The norm used is the standard $\mathcal{R}^n$ norm: $||\mathbf{q}||^2 = \sum_{i=1}^{n} q_i^2$. For a given geometric path, path length will depend on the units chosen for each degree of freedom[11].

2. Fit a third order spline interpolation $\mathbf{q} = \mathbf{f}(s)$ to the sequence of points $\{(s[k], \mathbf{q}[k])\}_{k=0}^{N}$

3. Use the spline interpolation to obtain $\mathbf{f}_s[k]$, $\mathbf{f}_{ss}[k]$, $\dot{s}_{max}[k]$, $\underline{\alpha}_1^i[k]$, $\overline{\alpha}_1^i[k]$, and $\alpha_2^i[k]$ at each via point.

4. During the integration of $\dot{s}_a(s)$, use the following interpolating function between via points:

$$\dot{s}_a(s) = \sqrt{\dot{s}_a[k]^2 + \frac{s - s[k]}{s[k+1] - s[k]}(\dot{s}_a[k+1]^2 - \dot{s}_a[k]^2)} \quad \text{for} \;\; s[k] \leq s \leq s[k+1] \qquad (7.18)$$

   This interpolating function corresponds to the first-order approximation to $\dot{s}_a(s)$ assuming $\ddot{s}(s)$ constant in the $s[k] \leq s \leq s[k+1]$ interval. In other words, if we assume $s(t) = s[k] + \dot{s}[k]t + \frac{1}{2}\ddot{s}[k]t^2$ in the $s[k] \leq s \leq s[k+1]$ interval and we solve for $\dot{s}(s)$ we obtain the interpolation equation (7.18).

5. To ensure all constraints are met in the interpolated region between via points, the constraint equations must be also be interpolated. The proximate-optimal algorithm uses the velocity-independent approximation to the torque limits already described in equation (7.14) (see Section 7.3.2). Therefore, the algorithm only needs to interpolate between successive via points the constraint functions $\dot{s}_{max}(s)$, $\underline{\alpha}_1^i(s)$, $\overline{\alpha}_1^i(s)$, and $\alpha_2^i(s)$ for each degree of freedom "i". The following interpolations are used for $s[k] \leq s \leq s[k+1]$, and for $i = 1, ..N_{dof}$:

$$\dot{s}_{max}(s) = \sqrt{\dot{s}_{max}[k]^2 + \frac{s - s[k]}{s[k+1] - s[k]}(\dot{s}_{max}[k+1]^2 - \dot{s}_{max}[k]^2)}$$

$$\underline{\alpha}_1^i(s) = \underline{\alpha}_1^i[k] + \frac{s - s[k]}{s[k+1] - s[k]}(\underline{\alpha}_1^i[k+1] - \underline{\alpha}_1^i[k])$$

$$\overline{\alpha}_1^i(s) = \overline{\alpha}_1^i[k] + \frac{s - s[k]}{s[k+1] - s[k]}(\overline{\alpha}_1^i[k+1] - \overline{\alpha}_1^i[k])$$

$$\alpha_2^i(s) = \alpha_2^i[k] + \frac{s - s[k]}{s[k+1] - s[k]}(\alpha_2^i[k+1] - \alpha_2^i[k])$$

6. The integration step will find a value of $\dot{s}_a[k+1]$ that approximates the maximum acceleration (deceleration) curve without violating the constraints.

[11] The results should be independent of this provided the constraints and equations of motion use these same units.

7. Once the values of $\dot{s}_a[k] \stackrel{\text{def}}{=} \dot{s}_a(s[k])$ has been obtained, the via times are derived by simple integration:

$$\Delta t[k] \stackrel{\text{def}}{=} \int_{s[k]}^{s[k+1]} \frac{ds}{\dot{s}_a(s)} = \int_{s[k]}^{s[k+1]} \frac{ds}{\sqrt{\dot{s}_a[k]^2 + \frac{s-s[k]}{s[k+1]-s[k]}(\dot{s}_a[k+1]^2 - \dot{s}_a[k]^2)}}$$

   This integral can be computed analytically.

8. Once the via times $\{t[k]\}_{k=0}^{N}$ are known, third order splines are fit to the sequence of via-time and via-point pairs $\{(t[k], \mathbf{q}[k]\}_{k=0}^{N}$. This involves one spline interpolation for each degree of freedom. This step might introduce small violations of the constraints, but in exchange, provides a trajectory described as a continuous function of time which can now be sampled at any rate.

7.5.4 Complexity of the algorithm

The algorithm presented (and its discrete implementation) have running times that are proportional to the length of the path (i.e it is $\mathcal{O}(L)$). Figure 7.6 shows the execution-time dependency with the number of via points and degrees of freedom for sequences of via points of increasing length. The sequences of via points used to evaluate the computational complexity of the algorithm must truly represent paths of increased length[12]. In addition for the comparisons to be meaningful, the paths must be ergodic in their geometric properties. The generation of such paths is described in Appendix E.

The time-complexity of the proximate-optimal algorithm is $\mathcal{O}(N_{via} \times N_{dof})$, N_{via} being the number of via points and N_{dof} being the number of degrees of freedom (DOF). To prove this, we can associate each operation in the algorithm with the via point being integrated at that time. Since each via point is integrated just once, and there is a constant number of such operations per integration step (there is no backtracking, each via point is integrated exactly once), the integration is $\mathcal{O}(N_{via})$. The remaining steps are also $\mathcal{O}(N_{via} \times N_{dof})$:

- The pre-computation stage (steps 1, 2 and 3) of the discrete algorithm are clearly $\mathcal{O}(N_{via} \times N_{dof})$. In particular each spline interpolation for each DOF is an $\mathcal{O}(N_{via})$ operation.
- The integration (step 4) is a $\mathcal{O}(N_{via})$ operation as we have justified before.
- Computing the via times by integrating $1/\dot{s}$ (step 5) is clearly an $\mathcal{O}(N_{via})$ operation.
- Fitting the final splines (step 6) is an $\mathcal{O}(N_{via} \times N_{dof})$ operation, each spline (to each DOF) being $\mathcal{O}(N_{via})$.

[12]As opposed to representing the same geometric path sampled with greater density of via-points.

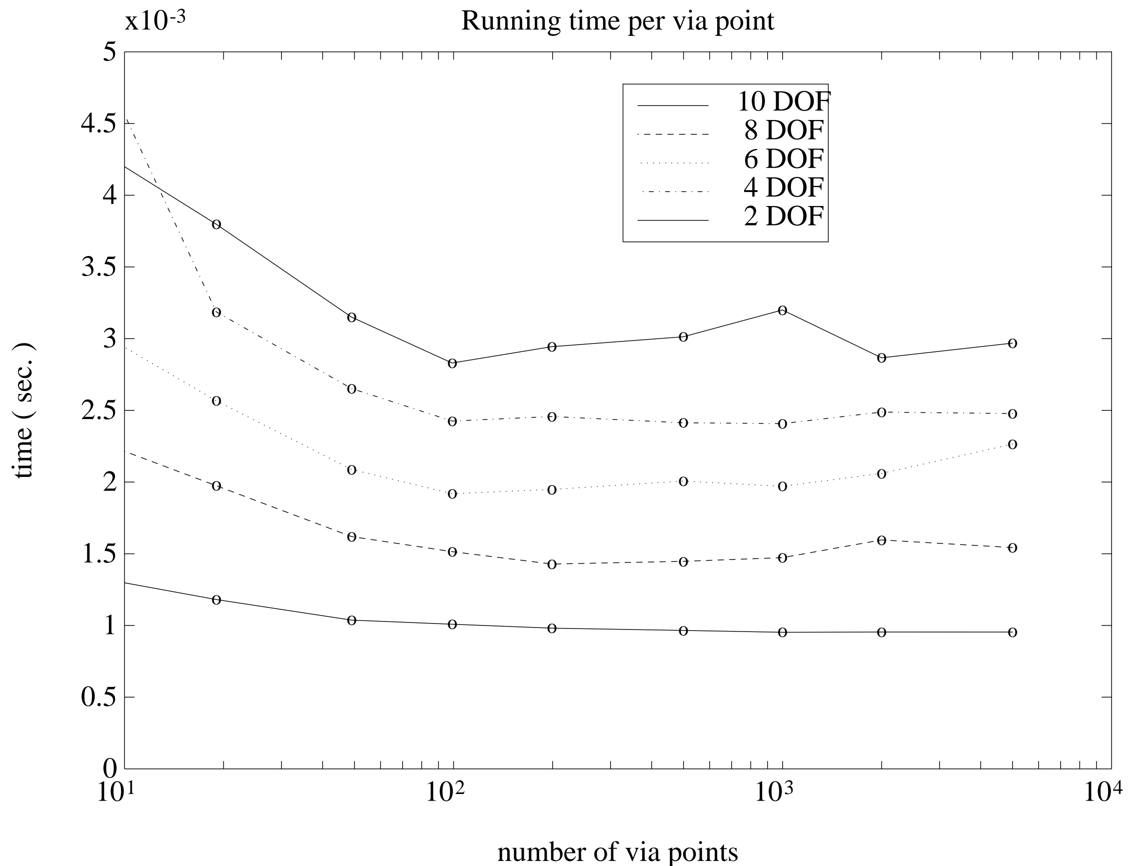


Figure 7.6: **Running time versus number of via points for different number of degrees of freedom**

This figure illustrates that the time-complexity of the algorithm is linear with the number of via points (or path length). We show that the execution time per via point is approximately constant over a 3 order of magnitude change in the number of via points. The timing corresponds to a Sparc station 2.

Figure 7.6 corroborates this analysis.

7.5.5 Complexity of other approaches

In the previous section we have shown the worst-case complexity of the proximate-optimal algorithm to be $\mathcal{O}(L \times N_{dof})$ where L is the length of the path. In this section we will discuss the worst-case complexity of other approaches in the literature. Obviously, worst-case complexity is a very crude measure of the practical performance of an algorithm. Not only does it measure asymptotic behavior which may represent "pathological" scenarios which may never arise in practice, but it also neglects the constant and lower-order coefficients that may be the most relevant in realistic

situations. Typical performance of different optimal and proximate-optimal time-parameterization algorithms in practical paths should be determined using statistical methods by comparing the performance of different algorithms in different sets of paths. To this end, the canonical paths introduced in Appendix E may prove useful. This should be the topic of future research. Worst-case complexity is nevertheless a useful metric in that it provides an upper bound that may be very useful for on-line applications.

We will not be able to address the worst-case complexity of many approaches that have been proposed in the literature. Rather, a brief justification of Table 7.1 for some of the most characteristic algorithms will be provided.

Algorithms using dynamic programming

Dynamic programming approaches have been described in [165] and [174]. *Shin et. al.* [165] show that this approach has a computational complexity of $\mathcal{O}(L \times N_\mu^2)$ where N_μ is the number of points in which they discretize the possible values of $\dot{s}$. Aside from the large space requirements of dynamic programming algorithms, there is a further complication derived from the need to estimate adequate values N_μ which may be problem and path specific. *Singh et. al.* [174] propose a recursive refinement of N_μ to address the problem of its selection. In any case, dynamic programming algorithms end-up being iterative, and the compute-time will depend on the specific characteristics of the paths, hence making it very difficult (impossible) to provide accurate estimates of their running time for a specific situation.

Classical direct-integration algorithms

The algorithms presented by *Bobrow, Dubowsky and Gibson* [72] and *Shin and McKay* [164] use a direct integration approach. The aforementioned papers do not address the computational complexity of the algorithms, but it can be shown through example that they have worst-case complexities of $\mathcal{O}(L^2)$. The algorithm presented by Shin and McKay in [164] is sketched in Figure 7.7. The important point to note is that, whenever the accelerating (decelerating) trajectory intersects the boundary region, it is necessary to search for a suitable switching point along the boundary region, and then backtrack until the original integral is met. Backtracking can take us past the region previously integrated, all the way back to the initial trajectory. The number of required backtracks, and the fact that we may end-up integrating the same piece over and over are the reasons why this algorithm has $\mathcal{O}(L^2)$ worst-case complexity. Figure 7.8 illustrates the general features of

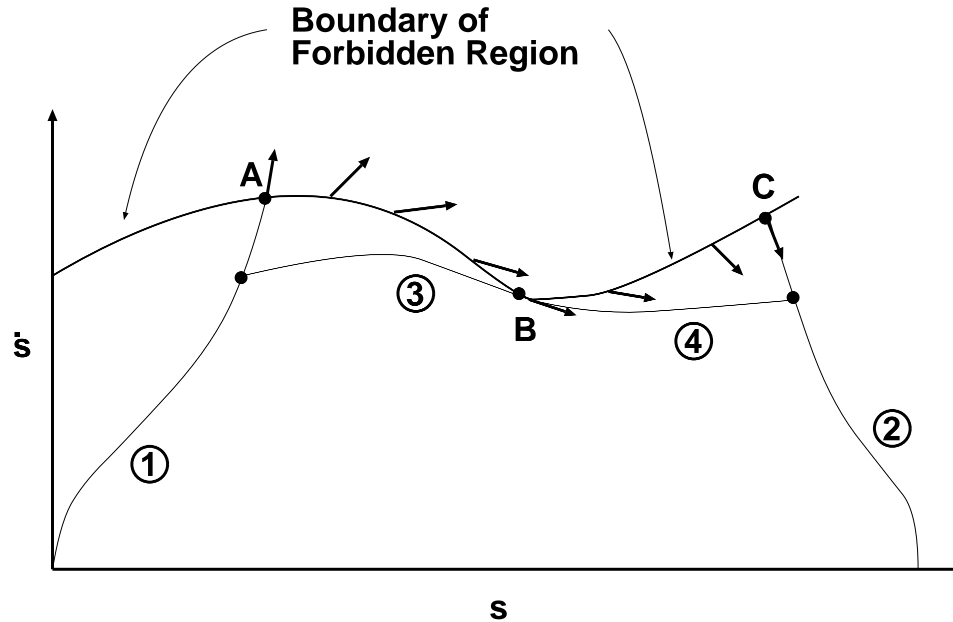


Figure 7.7: **Sketch of Shin's direct-integration algorithm (from [164], Figure 6)**

The algorithm integrates forward from the start along the maximum acceleration curve (1), and backwards from the end along the maximum deceleration curve (2), until either both branches meet or the boundary region is intersected (points A and C). Then starting from A, the boundary is searched for a point where the slope of the boundary matches the angle at which the "wedge" closes at that boundary point (indicated by arrows in the Figure). Once this point **B** *is reached, integration proceeds backwards (3) until the initial branch is met, and then forward (4), until the either the last branch is met, or the boundary curve is hit again (in which case, the whole process is repeated).*

a worst-case scenario with running time $\mathcal{O}(L^2)$. Appendix F gives a simple analytical path where the scenario described in Figure 7.8 actually occurs. Similar examples can be found for Bobrow's algorithm.

Proposed modification to direct-integration algorithms

There is a modification to Shin and McKay's algorithm that would avoid the worst-case situation sketched in 7.8. To the best of the author's knowledge, this modification has never been suggested in the literature. The modification changes the order in which the integrations are made and proceeds in three phases: initialization, forward integration, and backward integration. This algorithm is sketched in Figure 7.9 and described below.

Initialization The algorithm starts the same way integrating forward from the beginning along the $\frac{d\dot{s}}{ds} = \alpha_{max}(s, \dot{s})$ curve, and backwards from the end along the $\frac{d\dot{s}}{ds} = \alpha_{min}(s, \dot{s})$ until they

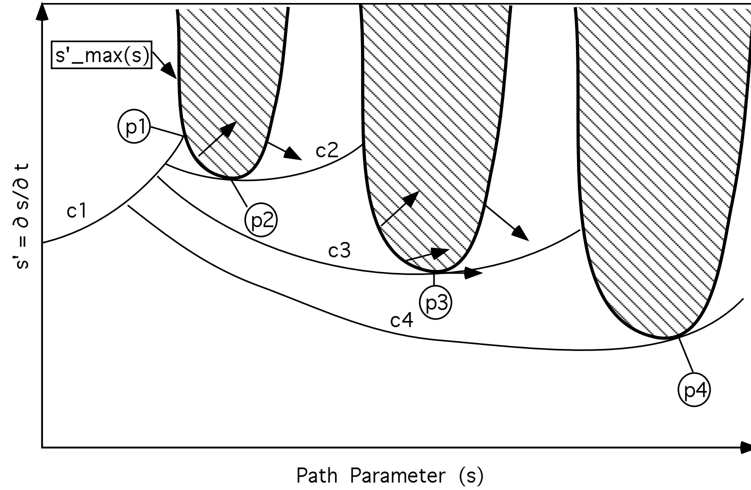


Figure 7.8: **Worst-case scenario for direct-integration algorithms.**

The algorithm used to obtain the optimal trajectory involves forward integration in phase space along the maximum-acceleration curve. If this curve (c_1 above) leaves the admissible region of phase space (point p_1 above), the boundary curve of the admissible region (curve $\dot{s} = \dot{s}_{max}(s)$) is searched for a point where its slope $\frac{d\dot{s}_{max}}{ds}$ matches the slope of the "closed wedge" described in Figure 7.2 (point p_2 above). Starting at p_2 we integrate backward along the maximum decelerating trajectory until the curve intersects c_1, and forward as before (this generates curve c_2). This process is repeated to generate curves c_3 and c_4. Notice that in this worst-case scenario, each time we have to integrate backwards all the way to the initial curve c_1 (that is, c_3 does not intersect c_2, c_3 does not intersect c_2 nor c_3, etc.). Therefore, since the backward integration takes time proportional to the path length L and the number of c_i curves is also proportional to L the running time is $\mathcal{O}(L^2)$.

either meet or intersect the boundary region (step (1) in Figure 7.9). Note that one of the basis of Shin-McKay's algorithm is that the forward branch can only intersect the boundary region at points s_1 where:

$$\phi(s_1) \stackrel{\text{def}}{=} \frac{d\dot{s}_{max}}{ds}(s_1) - \alpha_{max}(s_1, \dot{s}_{max}(s_1)) < 0$$

while, the backward branch can only intersect the boundary at a point s_2 if

$$\phi(s_2) \stackrel{\text{def}}{=} \frac{d\dot{s}_{max}}{ds}(s_2) - \alpha_{min}(s_2, \dot{s}_{max}(s_2)) > 0$$

and therefore, there must be an intermediate point $s_i \in [s_1, s_2]$ where $\phi(s_i) = 0$.

Forward integration The forward integration along the maximum-acceleration curve: $\frac{d\dot{s}}{ddt} = \alpha_m ax(s, \dot{s})$ has intersected the boundary region (point B_1 in Figure 7.9). Let this point be s_1. The point s_1 must verify $\phi(s_1) < 0$. Search the boundary region forward until a point s_2 with $\phi(s_2) > 0$ is found (point C in Figure). Resume integrating forward from that point (curve

(2) in Figure). Each time the boundary region is intersected, the procedure is iterated until the final backward-integration branch (curve $A_2 - B_2$ in Figure) is met (point K in Figure).

Backward integration Start at the point where the forward-integration branch met the initial backward-integration branch (point K in Figure). Immediately jump to the last point where the forward-integration branch departed from the boundary curve (point J). From this point J integrate backwards until the preceding forward-integration branch is intersected (point L). Every time the backwards integration meets a forward-integration branch other than the very first one, the procedure is iterated jumping back to the point where the corresponding forward integration branch departed from the boundary region (jumps from L→G, M→E, and N→C in bottom of Figure 7.9). The algorithm completes when the initial forward-integration branch is intersected (point O in Figure 7.9). Notice that the boundary region is never intersected during the backward integration (other than jumping to the beginning of the forward branch) because by construction the only pieces of the boundary region that are "exposed" (can be reached without first crossing the forward integration curve) are those with $\phi(s) < 0$ and the backward integration curve can only intersect the boundary region at points where $\phi(s) > 0$.

The preceding algorithm would have $\mathcal{O}(L)$ complexity. Following the boundary curve to find the potential switching points can be computationally expensive analytically, although characterizations such as the one presented by Slotine and Yang [176] can be used to speed this process (or it could be done numerically assuming that the different quantities can be linearly interpolated between via points). However, the above approach has several drawbacks compared with the proximate-optimal approach: **(1)** it potentially integrates every point twice (once forward, the other backward), **(2)** it is potentially very sensitive to the detailed shape of the path (which may not be well characterized if paths are described as sequences of via points), and **(3)** its running-time is very dependent on the number of "potential switching points," and therefore, it will be difficult to predict á priori the computational delay: a 'jagged" boundary region such as the one in appendix F will generate many potential "switching points" and hence be much slower to compute than a smoother version of essentially the same path.

Other proximate-optimal algorithms

The proximate-time optimal algorithm presented in *Butler and Tomizuka* [25] also uses a simplified constraint in place of individual joint-torque constraints which speeds up the computations but

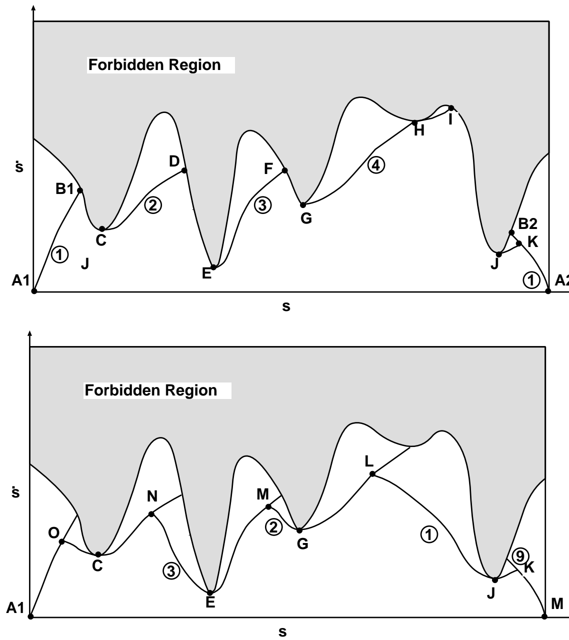


Figure 7.9: **Modified direct-integration algorithm that avoids worst-case complexity problem**

Forward-integration stage shown above, backward-integration stage below. For ease of illustration we are assuming that along the boundary, where $\alpha_{max}(s, \dot{s}) = \alpha_{min}(s, \dot{s})$, the value of this slope is zero. When the forward-integration branch starting from the beginning (A1) reaches the boundary at B_1 it is searched for the first point where $\phi(s) > 0$ (point C), forward integration proceeds along (2) until the next intersection at **D** *and so on until the last branch is intersected at point* **K**. *The backward integration (bottom diagram) starts at* **J** *along the $\alpha_{min}(s, \dot{s})$ curve until a forward branch is intersected at* **L**, *at this point, we jump to the beginning of that branch (point* **G**) *and keep repeating the process until the first branch is intersected at point* **O**.

the unmodified direct integration method is still used to integrate the trajectory, and therefore, the worst-case complexity is still $\mathcal{O}(L^2)$.

Iterative algorithms

Shin and McKay [166] have also proposed an iterative any-time algorithm combination of a gradient and binary search techniques. Starting from a path that satisfies all constraints, each iteration brings the path closer to the optimal by increasing the intermediate velocities "$\dot{s}$," whenever this is compatible with all the constraints. The authors also show that the complexity of their algorithm is $\mathcal{O}(N_\lambda^2)$. Where "λ" is the number of points in which the trajectory parameter "s"is discretized. This result, however, refers to the complexity with respect to using smaller step sizes in s for the *same* path. The complexity with respect to path length (i.e. keeping the step-size in s constant but

increasing the length of the path) is clearly $\mathcal{O}(L \times N_{iter})$. Where N_{iter} is the number of iterations required by the algorithm to converge. N_{iter} should be reasonably independent from the path length but depends on how close we desire the final approximation to the true optimal path. N_{iter}, however, is likely to be very dependent on the specific geometry of the path, so that accurate á priori predictions of the running time for a given path cannot really be made.

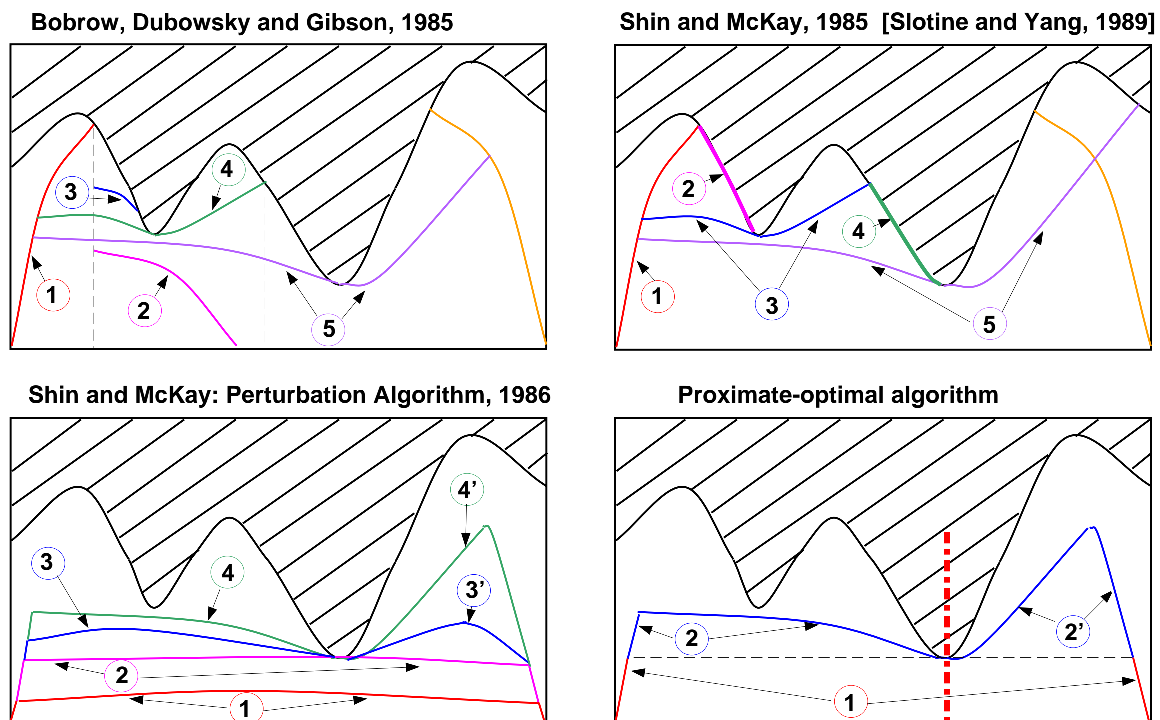


Figure 7.10: **Comparison of approaches to time-parameterization**

This figure compares the proximate-optimal algorithm with several classical optimal algorithms. Each call number indicates a sequential stage in the algorithm. Notice that the proximate-optimal algorithm is the **only** *one that never integrates the same region twice. From this comparison it is clear that it is "minimal" in the number of integration steps.*

Algorithm comparison summary

Figure 7.10 gives a visual comparison of several approaches. From this it becomes clear that for any scenario (not just the worst case) the proximate-optimal algorithm presented in this chapter is the "baseline" for computational efficiency because each via point is integrated exactly once, and all the other algorithms have to at least do that (and most likely iterate or backtrack). Therefore, the

proximate-optimal algorithm could also be used as a first step to provide a seed for other iterative (truly optimal) algorithms, such as the ones described before.

7.5.6 Configuration-independent limits on velocities and accelerations

In this section we discuss the specific case of configuration-independent limits in velocities and accelerations. These constraints may arise naturally in certain scenarios. For example as pointed out by Luh [97], our path may be given in Cartesian space and be limited not by actuator capabilities but by interactions with other objects (they use the example of moving a liquid in an open container as a case in which these constraints would be natural). There are many cases in which our constraints can be approximated in this way without significant performance loss (e.g mobile robots).

The proximate-optimal approach is, of course, applicable to this specific case. Nevertheless, by paying special attention to this case, and making some conservative approximations, a computationally more efficient solution can be derived.

These approximations replace the acceleration constraint: $-\ddot{\mathbf{q}}_{max} \leq \ddot{\mathbf{q}} \leq \ddot{\mathbf{q}}_{max} \iff \left|\frac{\ddot{\mathbf{q}}^i}{\ddot{\mathbf{q}}^i_{max}}\right| \leq 1\ ,\ i = 1..N_{dof}$ by the approximate constraint $\sum_{i=1}^{N_{dof}}(\frac{\ddot{\mathbf{q}}^i}{\ddot{\mathbf{q}}^i_{max}})^2 \leq 1$. It is useful to write this constraint using a metric tensor $\mathbf{Q}$.

$$\mathbf{Q} \stackrel{\text{def}}{=} \mathbf{diag}[..,\ \frac{1}{\ddot{\mathbf{q}}^i_{max}}\ ,..]$$
$$||\mathbf{x}||_Q \stackrel{\text{def}}{=} \mathbf{x}^T\mathbf{Q}\mathbf{x}$$

The constraint now becomes

$$||\ddot{\mathbf{q}}||_Q \leq 1 \tag{7.19}$$

Obviously, the new constraint (7.19) is more strict than (7.5.6). A simple geometric interpretation follows: constraint (7.5.6) requires $\ddot{\mathbf{q}}$ to be within the parallelepiped with sides intersecting the axis at $\pm\ddot{\mathbf{q}}^i_{max}$. The new constraint (7.19) replaces this parallelepiped with the ellipsoid with principal axes $\ddot{\mathbf{q}}^i_{max}$. It is clear from this interpretation that the sacrifice in performance will not be great. Presentation of detailed comparisons of the tradeoff between computation time and trajectory-optimality is beyond the scope of this thesis, but typically we see an order of magnitude reduction in computation time with an travel time that is within 30% of the optimal.

The use of constraint (7.19) allows several simplifications: First path-length can be computed using the metric tensor $\mathbf{Q}$. That is $ds^2 = d\mathbf{q}^T\mathbf{Q}d\mathbf{q}$. From this definition it follows that:

$$||\mathbf{f}_s||_Q = 1$$

$$\mathbf{f}_s{}^T \mathbf{Q} \mathbf{f}_{ss} = 0$$
$$\|\ddot{\mathbf{q}}\|_Q^2 = \left\|\mathbf{f}_s \ddot{s} + \mathbf{f}_{ss}\dot{s}^2\right\|_Q^2 = \ddot{s}^2 + \|\mathbf{f}_{ss}\|_Q^2 \, \dot{s}^4$$
$$\|\ddot{\mathbf{q}}\|_Q \leq 1 \iff |\ddot{s}| \leq \ddot{s}_{max}(s, \dot{s})$$
$$\ddot{s}_{max}(s, \dot{s}) \stackrel{\text{def}}{=} \sqrt{1 - \ddot{s}^4 \|\mathbf{f}_{ss}(s)\|_Q^2} = \sqrt{1 - (\frac{\ddot{s}}{\dot{s}^a_{max}(s)})^4}$$
$$\dot{s}^a_{max}(s) \stackrel{\text{def}}{=} \frac{1}{\sqrt{\|\mathbf{f}_{ss}(s)\|_Q}}$$

We see that the acceleration constraint imposes the limit $|\dot{s}| \leq \dot{s}^a_{max}(s)$. We can combine this limit by the one imposed by the velocity constraint (7.12) and write the constraints as:

$$|\dot{s}| \leq \dot{s}_{max}(s) \stackrel{\text{def}}{=} \min\{\dot{s}^a_{max}(s), \gamma(s)\} \tag{7.20}$$
$$|\ddot{s}| \leq \ddot{s}_{max}(s, \dot{s}) \iff \left|\frac{d\dot{s}}{ds}\right| \leq \alpha_{max}(s, \dot{s}) \tag{7.21}$$
$$\alpha_{min}(s, \dot{s}) = \ddot{s}_{min}(s, \dot{s})/\dot{s} \text{ For } \dot{s} \neq 0 \tag{7.22}$$

Again we have reduced the constraints to the general form of an instance of the DMTTP problem. The difference is that now $\dot{s}_{max}(s)$ and $\alpha_{min}(s, \dot{s})$ are very simple (and fast) to compute. Also we note that $\dot{s} \leq \dot{s}_{max}(s) \leq \dot{s}^a_{max}(s) \quad \Rightarrow \quad \alpha(s, \dot{s}) \geq 0$. So Theorem 1 applies without any further limitations on $\dot{s}_{max}(s)$.

Section 7.6 shows the performance of this algorithm on several "canonical" paths.

7.5.7 Modification of Ongoing Trajectories

In a dynamic environment it is not advisable to fully commit to a trajectory, even if it is computed on-line. For instance, the trajectory that takes a manipulator above a moving object in order to grasp it, needs to be computed ahead based on estimates on the future motion of the object. These estimates may change as the trajectory proceeds, and therefore, it is important for the trajectory generator to incorporate mechanisms that allow modification (*patching*) of the ongoing trajectory.

Figure 7.11 illustrates the concept of patching an ongoing trajectory: The remaining piece of the geometric path beyond a certain point (called the *patch point*) is replaced by a new piece (the *patch path*). As a result, the trajectory which was already started, needs to be modified. The modification starts at the *merge point*, located between the current and the patch point. The original trajectory remains unchanged until the *merge time* (time when the trajectory reaches the merge point). From there on, the patch trajectory is followed.

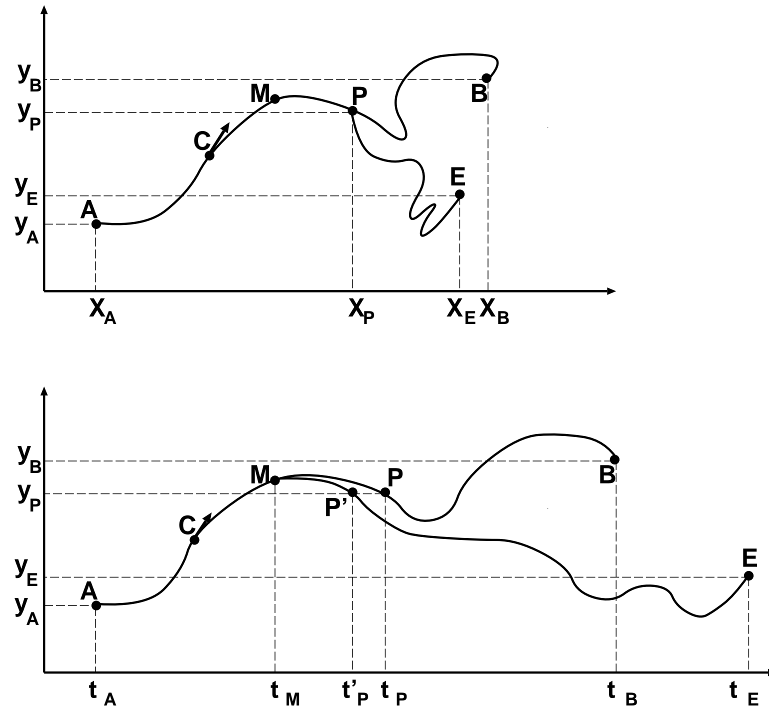


Figure 7.11: **Patching of an ongoing trajectory**

The top diagram shows the original and patched geometric paths in two dimensions (X and Y). The diagram underneath illustrates the corresponding trajectories for one of the coordinates (there is a corresponding one for every coordinate). The initial geometric path joined points A and B. The trajectory is currently at point C, when a new patch replaces the P-B section by the P-E patch. The resulting trajectory takes the system from C to E. As seen in the bottom diagram, even though the geometric path is the same until point P is reached, the trajectories start to differ at some intermediate point M between C and P. P is called the **patch point**, *M is the* **merge point**, *and t_M the* **merge time**. *Notice how the patch point P is reached at a time "t'_p" different from the original t_p.*

Clearly, a trajectory that has been initiated cannot be arbitrarily patched. Given the current state of the trajectory, the patch may be impossible to achieve without violating the dynamic constraints on the system (for example, the patch may require the trajectory to stop suddenly, or make a sharp corner, and the system may be moving too fast to do that without exceeding the torque limits on the actuators). Given the initial geometric path and trajectory, the patch to the geometric path, and the current state, the following issues must be addressed by the trajectory-modification algorithm:

1. Is the patch feasible? That is, can the path be patched as required without exceeding the dynamic constraints on the system.

2. For a feasible patch, what is an appropriate (optimal) merge time?

3. Given that the geometric path from the merge to the patch point has not changed, can this information be used to recompute the trajectory from the merge point more efficiently?

One of the benefits of the proximate-optimal algorithm, is that due to its simplified properties, it provides efficient mechanisms to address all the above issues. The operation of the trajectory-modification algorithm is essentially identical to the regular proximate-optimal algorithm except that integration starts at the end of the trajectory, and whenever a decoupling point is reached, the left (earlier) piece of trajectory is computed first. The process can be seen in the phase-space plots of Figure 7.12. First the phase-space limits are recomputed for the modified region and patched into the original, resulting on new phase-space constraints (middle of Figure 7.12), next since the compute-time of the algorithm is well characterized as a function of path length, the current state of the trajectory can be advanced forward to account for the compute time (point C in Figures 7.11 7.127.13). At this point, integration proceeds backwards from the final point E. The proximate-optimal constraints allow the algorithm to advance backwards (towards the beginning of the path) as soon as the integrated value of $\dot{s}$ is higher than that of a local minimum (whenever this happens we know that each piece is independent and we can choose to integrate the earlier one first). This proceeds until the $\dot{s}$ value of the leftmost integration can be joined with a horizontal line with some point between the current and patch point of original phase-space trajectory (curve from C to P), without intersecting the $\dot{s}_{max}(s)$ boundary (this occurs at point F in Figure 7.13). At this point we know (thanks to Theorem 1) that the patch is feasible. Finally, each one of the independent regions is integrated, including the one from the current point C to F, and this yields the patched trajectory.

The particular order in which the integration is performed has been chosen to minimize the time required to determine whether the patch is feasible or not. This gives whoever is specifying the patch (the planner in our case) a chance to try something different. Figure 7.13 shows another example of a trajectory being patched. This figure also illustrates some of the optimizations that have been incorporated to find the merge point that is closest to the patch point. This is important because a common use of patching is to modify the latter stages of the trajectory where the current point is quite far from the patch point. Moreover, these modifications can occur frequently. For instance, the proximate-optimal algorithm has been used to track a seam visually with a robot-mounted camera. As the robot tracks the seam, new points appear in the field of view (the patch is just an addition of a piece at the end). This process occurs continuously as the robot follows the seam so efficiency is critical (see Morse [112] for details on the visual seam-following application).

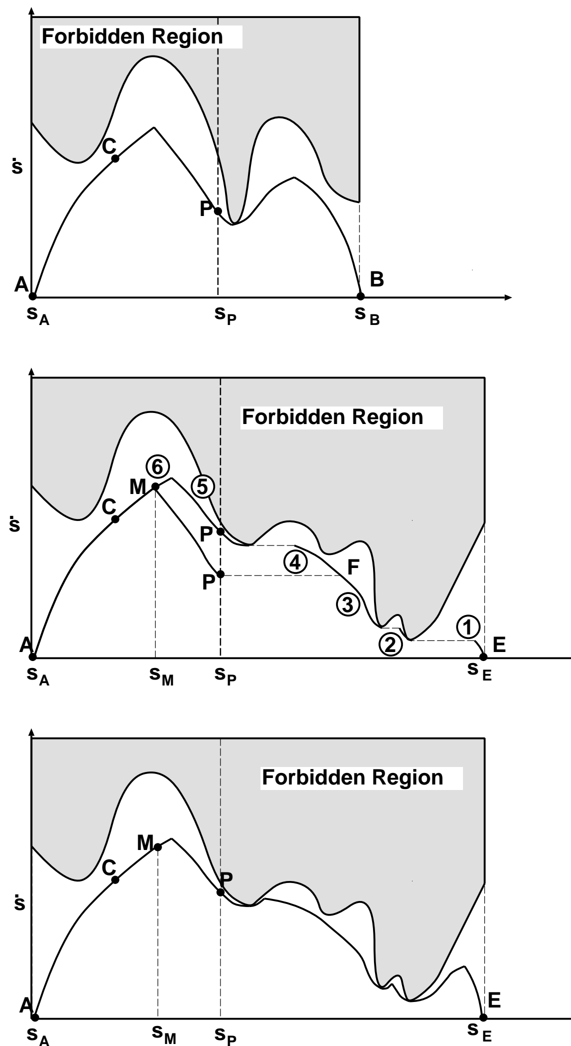


Figure 7.12: **Trajectory Modification Algorithm**

Proximate-optimal algorithm to modify an ongoing trajectory. These plots are the phase-space equivalent to Figure 7.11. The top plot represents the original trajectory which is patched with a new geometric path starting at the patch point P. The middle plot represents the new phase-space constraints (only the $\dot{s}_{max}(s)$ *represented for simplicity) that have changed from the patch point onwards. To determine whether the patch is feasible and the merge point M, the integration proceeds backwards from E (stages 1, 2, and 3). Whenever a local minimum of the same height is reached, the integration can jump left to the local minimum (decoupling). As soon as point F is reached, the algorithm determines that the patch is feasible. The remaining backward integrations steps (4, 5, and 6) determine the merge point M. Finally each independent section (horizontal jump) is integrated to obtain the final parameterization (bottom plot).*

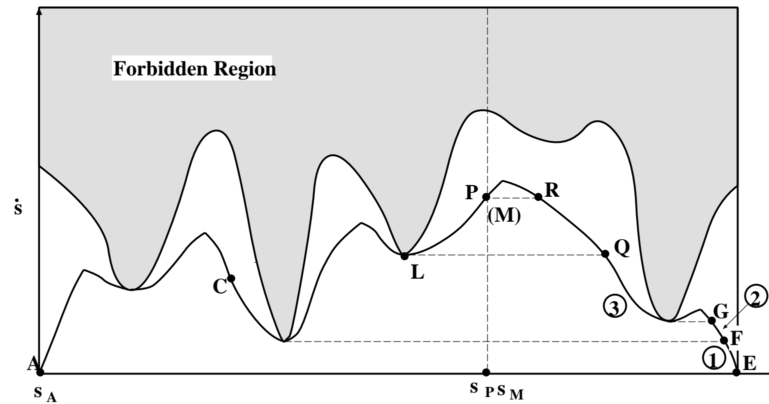


Figure 7.13: **Example of Trajectory Modification Algorithm**

This example illustrates several optimizations that often occur when the patch is added far ahead of the current point. The patch point **P** *and the trajectory up to that point (A to P) are already computed, and the trajectory is being followed. The system is currently at point* **C** *(already accounting for the computational delay). The trajectory-modification algorithm starts at* **E** *along (1). As soon as the integral curve reaches* **F***, we know that the patch is feasible because there is a horizontal line that reaches the original trajectory between C and P without crossing the boundary region. The integration proceeds along (2) and is immediately separated into two pieces at point G. The left piece is selected first (3). The integration proceeds, until a horizontal line can be drawn to* **L** *(the last decoupling point in the original trajectory before the patch point P). The remaining piece is integrated until either we can joint the integral curve with the patch point with a horizontal line (point R), or else the integral curve intersects the original trajectory. The different independent pieces left are integrated as usual. Note that in this case the merge point* **M** *is just the patch point P.*

Two things are worth noting in the example of Figure 7.13: First, the determination of the feasibility of the path is almost immediate due to the existence of a fairly strict constraint (local minimum K) in the original trajectory between the current and patch points (C-P piece). Second, the original trajectory is never recomputed (i.e. the piece from C to P is used without change or extra effort). In general, only the piece from the last decoupling point before the patch (point L) to the patch may need re-computation.

7.6 Experimental Results: Time Parameterization

This section presents results of applying the proximate-optimal time-parameterizing to several sequences of via points. These sequences have been selected to represent geometrically simple

paths for illustration purposes. In actual practice, all planner generated-motions of both manipulators and manipulated-objects in the workcell (see Chapter 3) use the proximate-optimal time-parameterization algorithm to interpolate smooth trajectories that traverse the via points generated by the planner.

7.6.1 Straight-line trajectories for a point mass

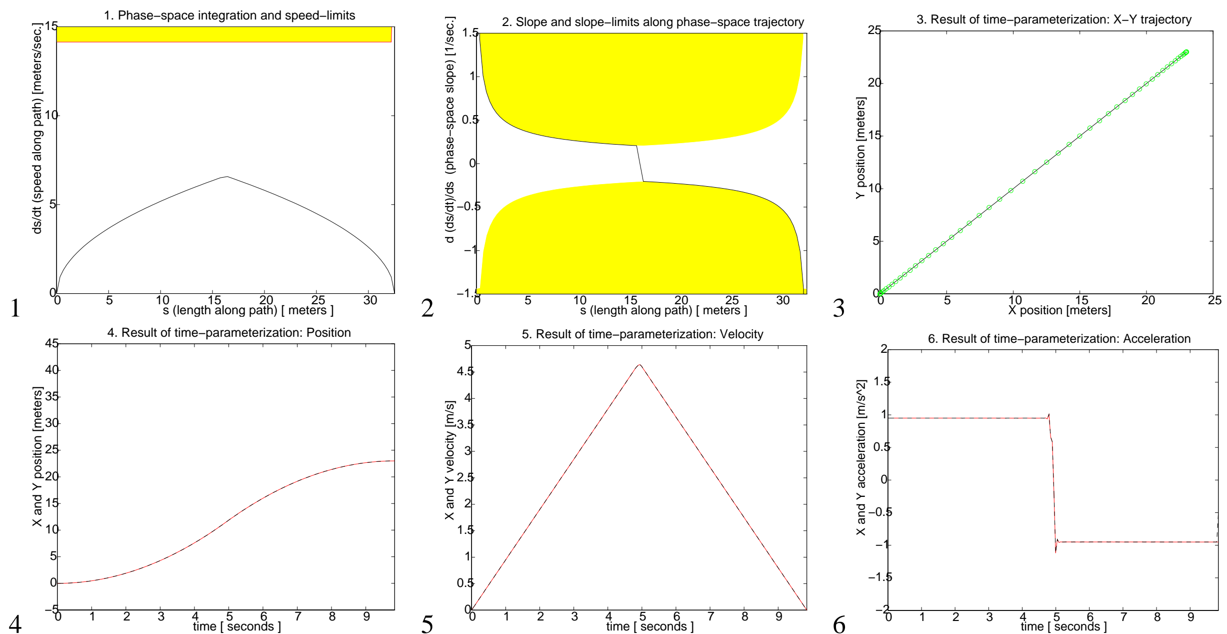


Figure 7.14: **Time-parameterization of straight-line path with acceleration limits**

The straight-line trajectory is illustrated in the third plot. The acceleration limits are the only ones exercised in this path. The first two plots illustrate the phase-space constraints and the results of the phase-space integration. The last three plots illustrate the integrated trajectory (position, velocity, and acceleration) for of the two degrees-of-freedom (they are on top of each other). As expected, the optimal trajectory is bang-bang, accelerating with the maximum acceleration ($0.95m/s^2$) for the first half and then decelerating with the maximum deceleration in the last half.

Figures 7.14 and 7.15 illustrates the result of time-parameterizing a cartesian straight-line path in two dimensions. In these examples, the equations of motion provided to the trajectory-generation algorithm are those of a 1 Kg pure mass. Therefore, torque limits map directly into acceleration limits. These examples have been included because the minimum-time solution can be easily computed analytically and the different constrains are simple to interpret. They will be used to introduce the plots that will later be used to present the results of time-parameterizing real paths of the workcell manipulators. In these two examples, the geometric path as a function of path length "s" is:

$x(s) = y(s) = s/\sqrt{2}$, and therefore the velocity, and acceleration limits are: $\min\{\dot{x}_{min}, \dot{y}_{min}\} \leq \dot{s}/\sqrt{2} \leq \max\{\dot{x}_{max}, \dot{y}_{max}\}$ and $\min\{\ddot{x}_{min}, \ddot{y}_{min}\} \leq \ddot{s}/\sqrt{2} \leq \max\{\ddot{x}_{max}, \ddot{y}_{max}\}$.

Figure 7.14 illustrates the case where the trajectory is acceleration-limited (force-limited). The velocity and acceleration limits are $\dot{x}_{max} = \dot{y}_{max} = -\dot{x}_{min} = -\dot{y}_{min} \stackrel{\text{def}}{=} v_{max} = 10m/s$, and $\ddot{x}_{max} = \ddot{y}_{max} = -\ddot{x}_{min} = -\ddot{y}_{min} \stackrel{\text{def}}{=} a_{max} = 0.95\ m/s^2$. The first plot in Figure 7.14 contains the phase-space limits on the speed $\dot{s} <= \dot{s}_{max}(s)$ along with the time-optimal solution $\dot{s} = \dot{s}^*(s)$ computed by the proximate-optimal algorithm. For this simple problem, the solution is to accelerate with the maximum acceleration for the first half of the path and then decelerate with the maximum deceleration. In phase-space the maximum-acceleration curve is $\dot{s}_{max}(s) = \sqrt{2\sqrt{2}\ s\ a_{max}}$. The second plot in Figure 7.14 illustrates the "running" constraints on the slope $|\frac{d\dot{s}}{ds}| = |\ddot{s}/\dot{s}| \leq a_{max}\sqrt{2}/\dot{s}$ along the phase-space trajectory $\dot{s} = \dot{s}^*(s)$. For the first half of the trajectory, the running constraints on the slope are $\frac{d\dot{s}}{ds} \leq \sqrt{a_{max}/s\sqrt{2}}$. The optimal trajectory $\dot{s}^*(s)$ must always be along either the boundary of the allowed region $\dot{s} = \dot{s}_{max}(s)$ shown in the first plot, or else exercise the "running" slope limits shown in the second plot. The third plot in Figure 7.14 contains the two-dimensional straight-line path. The circles represent positions along the path at constant time intervals. The remaining three plots in Figure 7.14 contain position, velocity and accelerations for each degree-of-freedom for the trajectory computed by the proximate-optimal algorithm. This trajectory corresponds to the solution which simply accelerates with the maximum acceleration of 0.95 m/s^2 for the first half and then decelerates with acceleration $-0.95\ m/s^2$ for the second half[13]. The theoretical minimum-time trajectory for this path takes 9.84 sec. which corresponds almost exactly with the result from the proximate-optimal algorithm.

Figure 7.15 illustrates the same path and equations of Figure 7.14 where now the velocity limits $\dot{x}_{max}$ and $\dot{y}_{max}$ have both been reduced to $2.85m/s$. The first plot illustrates the optimal phase-space trajectory accelerating along the maximum-acceleration curve, until the maximum velocity (which corresponds to a limit in $\dot{s}_{max}(s)$) is reached. The trajectory coasts at the maximum velocity until the final part where it decelerates with the maximum deceleration. The second plot corroborates that indeed the "slope" constraints on $\frac{d\dot{s}}{ds}$ are fulfilled, and exercised during the acceleration and deceleration phases. The last four plots in Figure 7.15 illustrate the result of the proximate-optimal algorithm, which correspond to the optimal bang-off-bang solution. The theoretical optimal minimum time is 11.1 sec. which again, corresponds almost exactly with the result from the proximate-optimal algorithm.

[13]The small discrepancies are due to the discrete integration and subsequent spline-fitting.

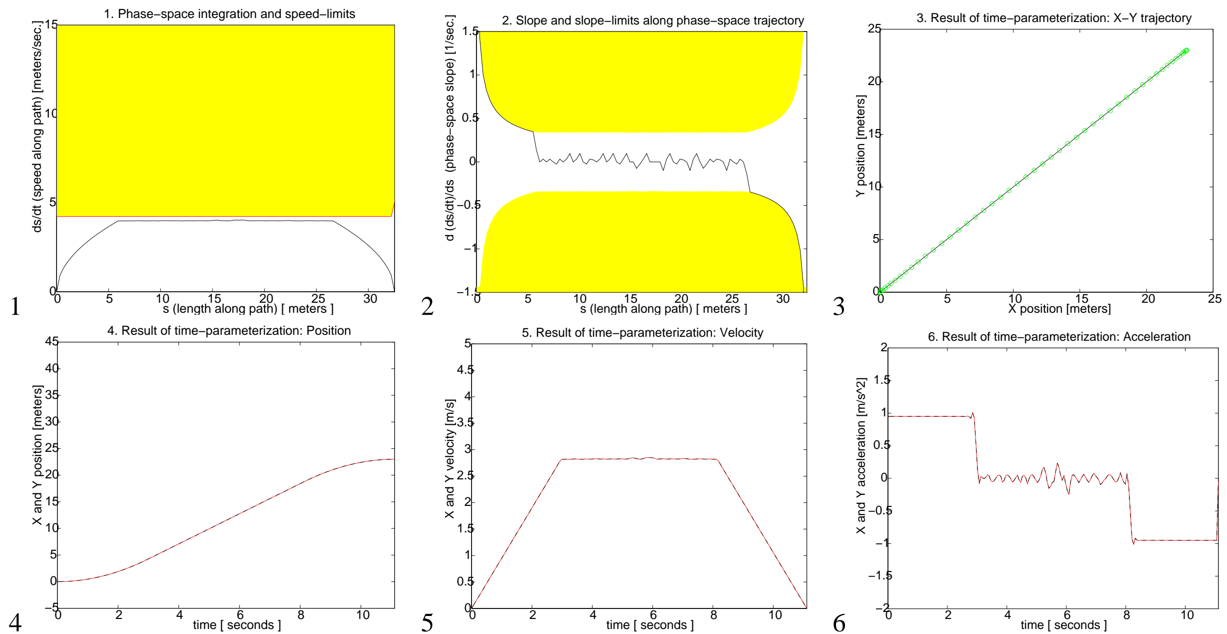


Figure 7.15: **Time-parameterization of straight-line path with velocity and acceleration limits**

The straight-line trajectory is illustrated in the third plot. In this trajectory both velocity, and acceleration limits are exercised. The first two plots illustrate the phase-space constraints and the results of the phase-space integration. The last three plots illustrate the integrated trajectory (position, velocity and acceleration) for of the two degrees-of-freedom (they are on top of each other). As expected, the optimal trajectory is bang-off-bang, accelerating with the maximum acceleration ($0.95m/s^2$), then coasting at the maximum velocity ($2.85m/s$), and finally decelerating with the maximum deceleration.

7.6.2 Trajectories for the workcell manipulators

The results in this section correspond to trajectories for the manipulators in the manufacturing workcell. To compute these trajectories, the proximate-optimal algorithm uses the equations of motion of the manipulators (see Section (6.4)) to ensure that the torque limits are not exceeded. The torque limits for these experimental results where $9.5Nm$ for both the shoulder and elbow joints[14]. The remaining degrees-of-freedom (Z and Yaw) are not shown for simplicity, but they are also modelled and accounted for by the trajectory-parameterization algorithm. These limits correspond to the torque that appears in Equation (6.4), not the actual torque delivered by the motors (which is applied through the flexible, geared drivetrain). The use of these limits corresponds to modeling the manipulators as having rigid drivetrains and assuming the joint-torque loops are capable of achieving these rigid-joint dynamics within the requested bandwidth. This approach is consistent with the

[14]These torques are well within the capabilities of the shoulder and elbow motors. However, they are larger than the (safer) limits used during normal operation.

layered control hierarchy presented in Chapter 6 and the fact that both planner and strategic controller specify paths for this idealized manipulator. The trajectory-generation algorithm could generate trajectories for any other model of the arm dynamics, requiring only that both the appropriate sequence of via points and equations of motion are provided.

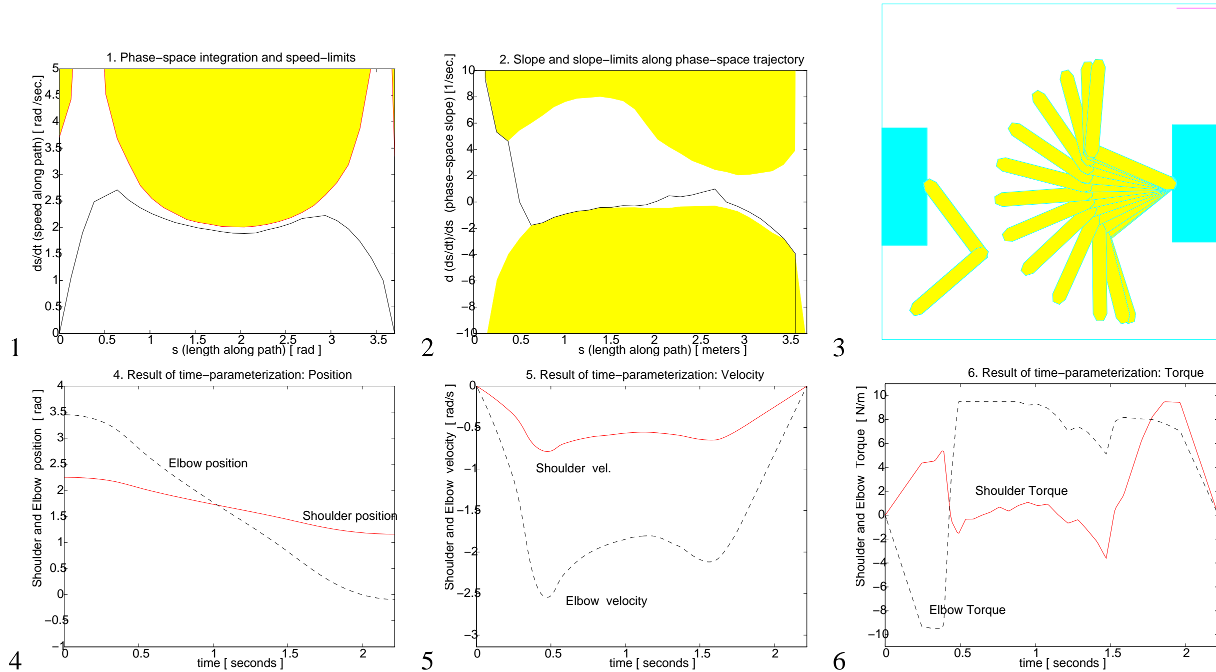


Figure 7.16: **Time-parameterization of a joint-space straight-line path**

The third plot illustrates the manipulator path. This path has been parameterized using the equations of motion of the right SCARA manipulator in the workcell. The bottom row illustrates the resulting trajectory (positions and velocities) and the required torques for this trajectory. The optimal trajectory would always keep at least one actuator saturated (at the maximum or minimum torque) at all times. The proximate-optimal algorithm comes close to this result as seen in the last plot. The differences are due both to the more strict constraints imposed and errors introduced during the integration process.

Figure 7.16 illustrates the result of time-parameterizing a planner-generated sequence of via points that moves the right arm across the workspace (third plot in Figure). The particular planner used defaults to straight joint-space moves whenever they are safe. The first plot in Figure 7.16 illustrates the proximate-optimal trajectory in phase space along with the phase-space limit $\dot{s} \leq \dot{s}_{max}(s)$. The second plot illustrates the slope limits (i.e. limits on $\frac{d\dot{s}}{ds}$) along the proximate-optimal trajectory. As expected, the proximate-optimal solution switches between these limits in order to satisfy them while remaining within the legal phase-space boundary. Plots three through six

illustrate the position, velocities, and torque along the proximate-optimal trajectory. In the sixth plot, we wan see that, as expected, the trajectory keeps at least one actuator close to saturation.

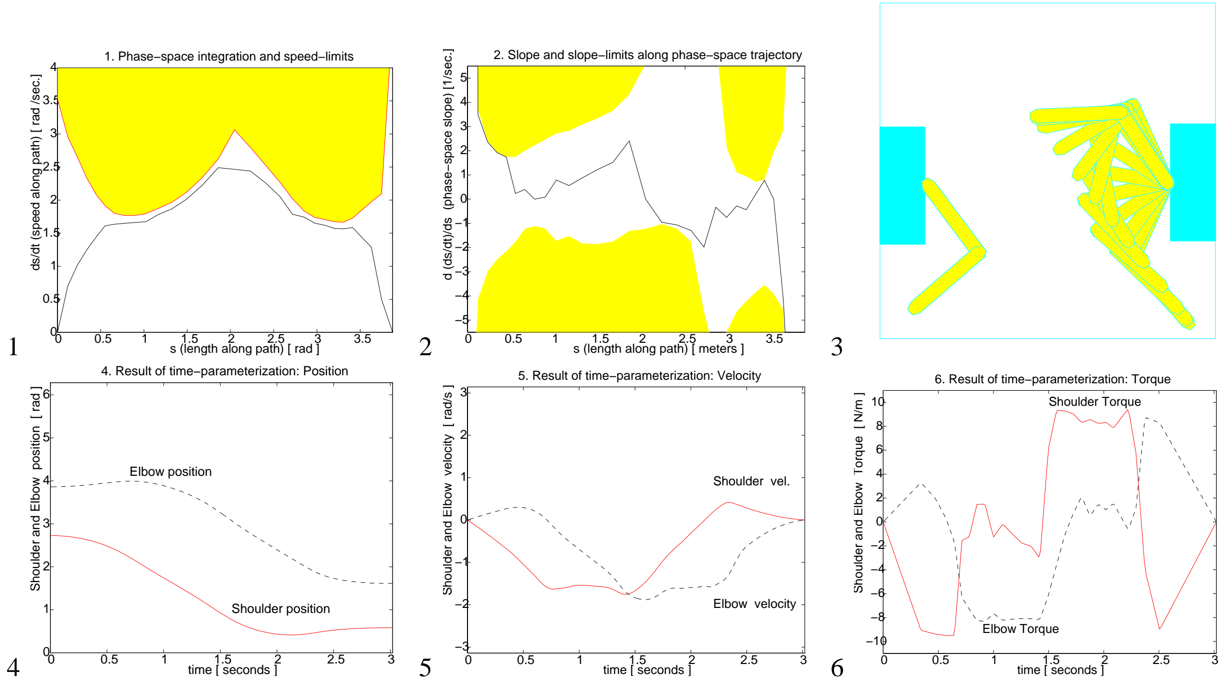


Figure 7.17: **Time-parameterization of a cartesian straight-line path**

The third plot illustrates the manipulator path. This path has been parameterized using the equations of motion of the right SCARA manipulator in the workcell. The last three figures illustrate the resulting trajectory (positions and velocities) and the required torques for this trajectory. Again we can see that one of the actuators is always close to saturation ($\pm 9.5 Nm$).

Figure 7.17 illustrates the result of time parameterizing a cartesian straight-line path. The phase-space integration can be seen in the first plot. It corresponds to a torque-limited trajectory that hugs the phase-space limit $\dot{s} \leq \dot{s}_{max}(s)$. From the second plot, we can see that in the regions where the proximate-optimal solution is not touching the maximum-speed curve $\dot{s}_{max}(s)$, it is constrained by the slope limits on $\frac{d\dot{s}}{ds}$. The fact that the phase-space trajectory does not ride along the slope limits of the second plot (as it should) is not due to having reached any velocity limits, rather, it is due to the small conservative margin used during the numerical integration. This margin prevents the phase-space trajectory from touching the phase-space boundary $\dot{s}_{max}(s)$ as seen in the first plot. In the absence of velocity limits, if this boundary where touched, one of the corresponding limits on the slope in the second plot would reach the zero-slope leaving no numerical margin for the discrete integration. Although this margin causes suboptimal trajectories, we can see from the sixth plot

that the values of the torque in those regions is still above 80% of its maximum value, resulting in trajectories close to the time-optimal.

Figure 7.17 illustrates the result of time-parameterizing a planner-generated path for two manipulators simultaneously. To the trajectory-generation algorithm this paths are just like the single-arm paths, except the via-points correspond to an 8$^{\text{th}}$-dimensional space (four degrees-of-freedom per manipulator), and the dynamic equations of motion (EOM) correspond to the concatenation of the EOM for each manipulator[15]. The dual-arm path is illustrated in the third plot of Figure 7.17. The first two plots illustrate that the phase-space trajectory satisfies all limits and, as expected, switches between the two slope limits (second plot). The last two plots illustrates that all accelerations are within their limits of ± 9.5 Nm.

7.6.3 Modifications to on-going trajectories

This section presents results on the time-parameterization of on-going trajectories. These results are difficult to illustrate with the dual-arm manipulator because the paths are quite short, and the resulting modifications are so close to each other that the results are too cluttered to be easily visualized. Instead, results will be resented for a simpler "made-up" system. The trajectory modification algorithm has been extensively used by following research, and detailed experimental results on a macro-mini manipulator system will be presented in [112].

The system used for this section corresponds to a point mass in two dimensions. This system is dynamically equivalent to the X and Y degrees of freedom of the free-floating space robots used by other experiments in the laboratory [189, 29], in fact, follow-on research with these vehicles [179] also uses the proximate-optimal algorithm described in this chapter.

Figure 7.19 illustrates the original path and the two subsequent patches. The equations of motion correspond to those of a 1 Kg mass. Only force limits of 1.5 N are used, these limits correspond to acceleration limits of 1.5 m/s^2.

Figure 7.20 contains the time parameterization of the initial path (shown in the third plot of the Figure). The first two plots illustrate the proximate-optimal solution and how it satisfies all phase-space limits. As expected, the solution switches between the slope limits (second plot) while remaining within the $\dot{s} \leq \dot{s}_{max}(s)$ limit (first plot). Plots three to six illustrate the position, velocity,

[15]This is a general feature of the time-parameterization algorithm, all it really needs is the sequence of via-points in some suitable "configuration-space," along with the corresponding equations-of-motion mapping from position, velocity and acceleration in this space to actuator torques.

and acceleration for each degree of freedom. From the sixth plot we can see that it is the Y acceleration the one that is always close to its limit.

While the trajectory illustrated in Figure 7.20 is in progress, it is suddenly modified. Figure 7.21 illustrates this process. The third plot illustrates the *current point* (point where the trajectory is at the time the modification is made) and the *patch point* (the point beyond which the trajectory is modified). These points were described in section 7.5.7. As seen in the third plot, the trajectory is modified at time 7 $sec.$, and the modification changes the trajectory beyond 12.3 sec. This modification causes the algorithm to recompute the remaining beyond the current point. This computation is depicted in the phase-space plots (first and second plot) of Figure 7.21. The modified trajectory beyond the *patch point* has smaller curvatures, and as a result, the phase-space limits (first plot) are higher. The velocity and accelerations of the modified trajectory are shown in the fifth and sixth plots, and they exhibit the expected alternation between maximum and minimum acceleration for each degree-of-freedom.

Figure 7.22 illustrates the second modification to the on-going trajectory. The third plot in the Figure contains the final geometric path after both modifications have been made. The second and third plot also show the *current point* (at time 17 $sec.$) and the *patch point* at 39 sec. The first two plots illustrate the phase-space integration starting at the current point. Plots three trough six show the complete trajectory resulting from both modifications. It is clear from the sixth figure that the acceleration limits are met at all times.

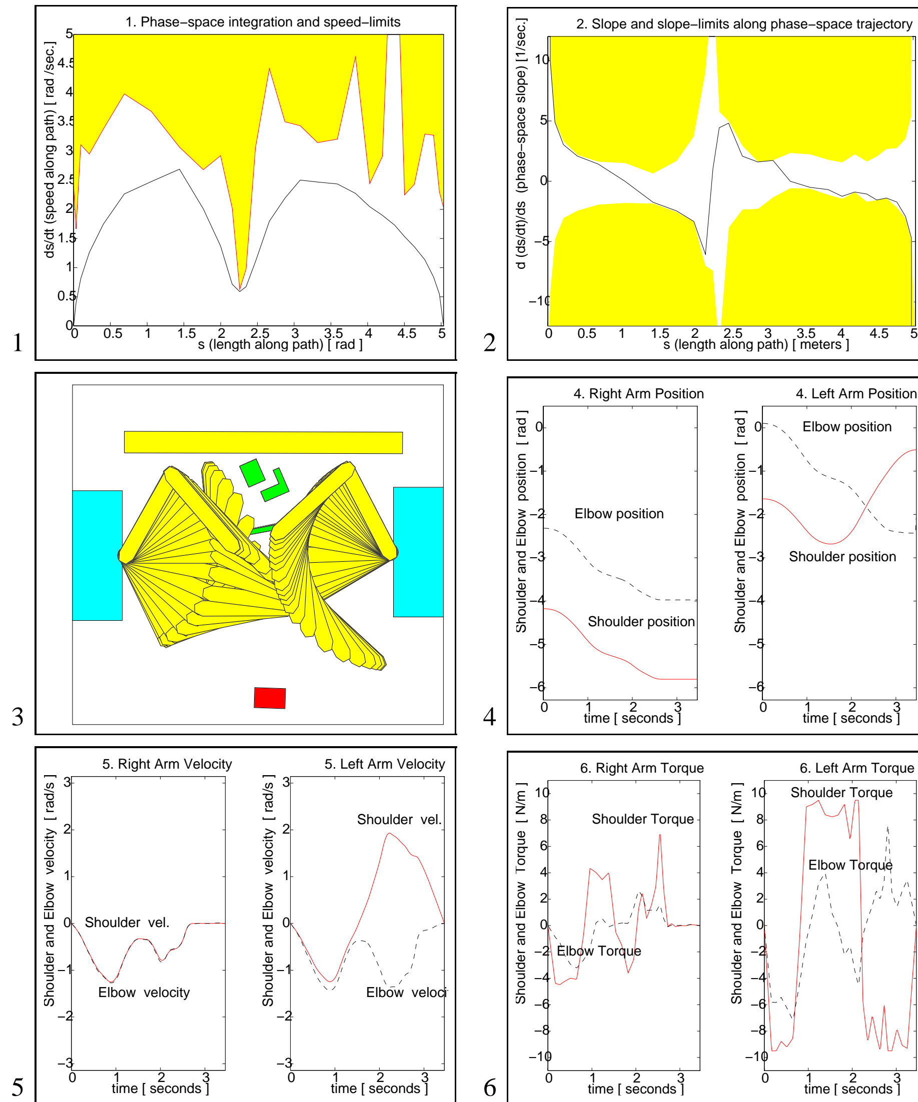


Figure 7.18: **Time-parameterization of a dual-arm path**

A coordinated motion for both manipulators is shown in the 3rd figure. The phase-space constraints and integral curve look similar to those of Figures 7.16, and 7.17. The only difference being that the "path" contains now the coordinates of both manipulators. The resulting trajectory (position and velocity) for each one of the manipulators is shown in figures 4 and 5. The 6th figure shows the actuator torques along the computed trajectory. Again one of the actuators (in this case belonging always to the right arm) is always close to saturation.

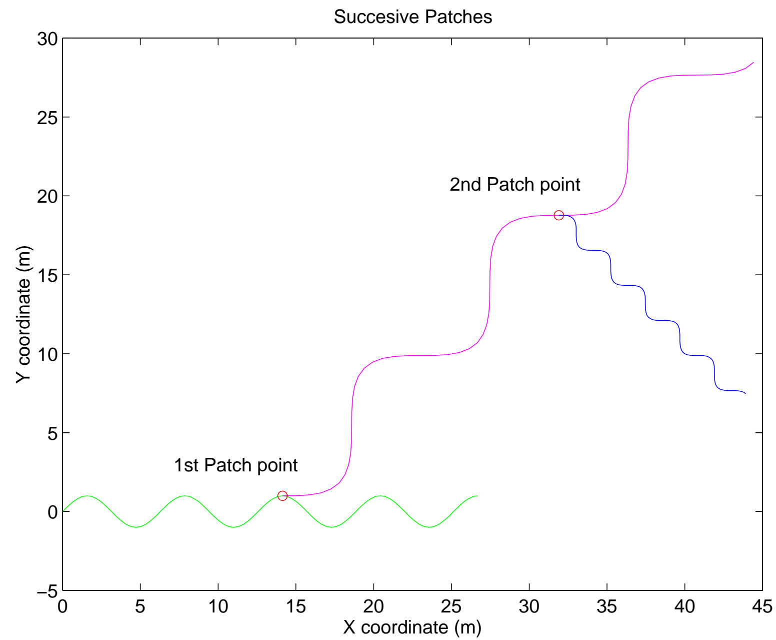


Figure 7.19: **Initial path and two subsequent modifications**

Illustration of the initial path and each one of the patches as given to the time-parameterization algorithm. These patches occur while the trajectory is being executed. The time-parameterization proceeds for the original path and each one of the two patches is illustrated in Figures 7.20, 7.21, and 7.22.

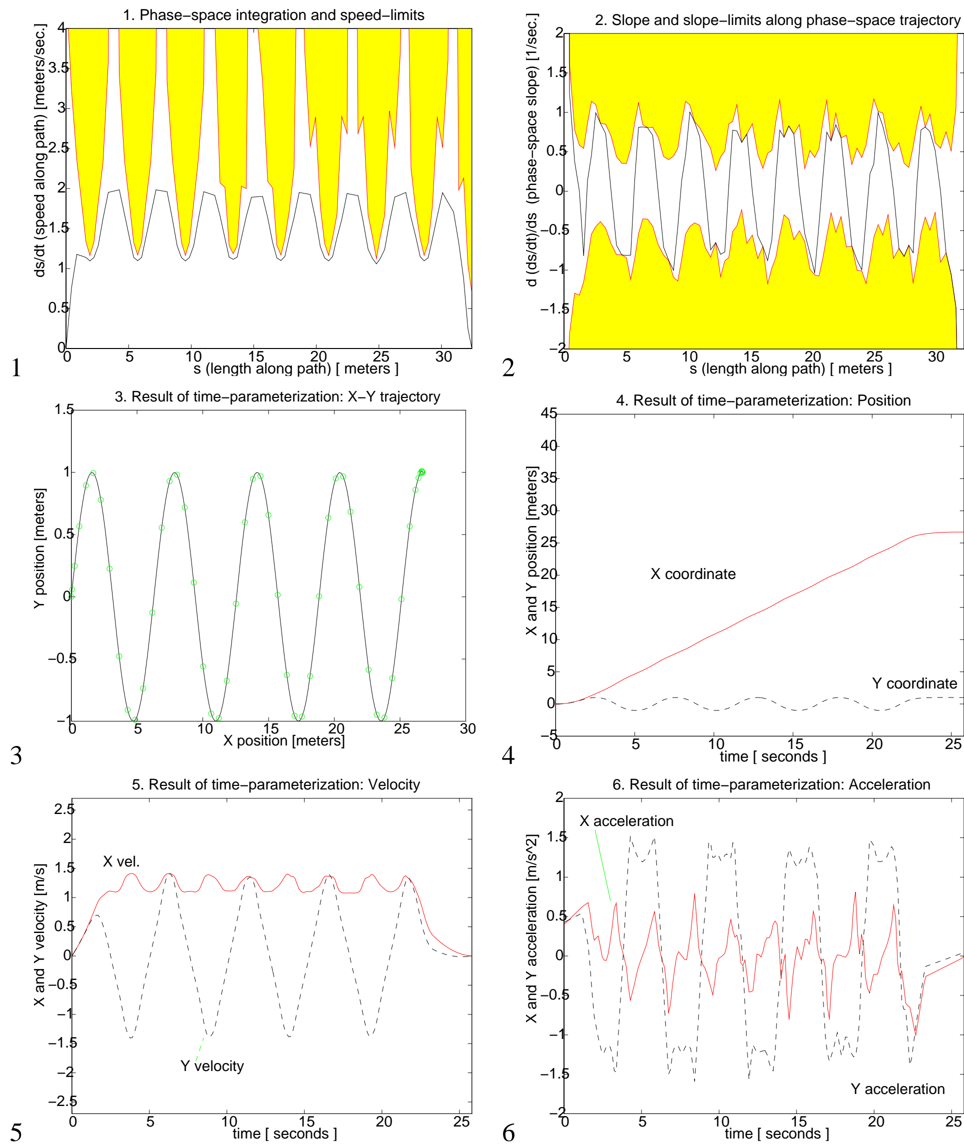


Figure 7.20: **Time-parameterization of initial path**

The initial path shown in the 3rd plot (corresponding to the first piece in Figure 7.19) is parameterized before the trajectory starts. The phase-space constraints and integral are shown in the first two plots. The velocity, and acceleration profile are close to those expected for a sinusoidal path with only acceleration constraints.

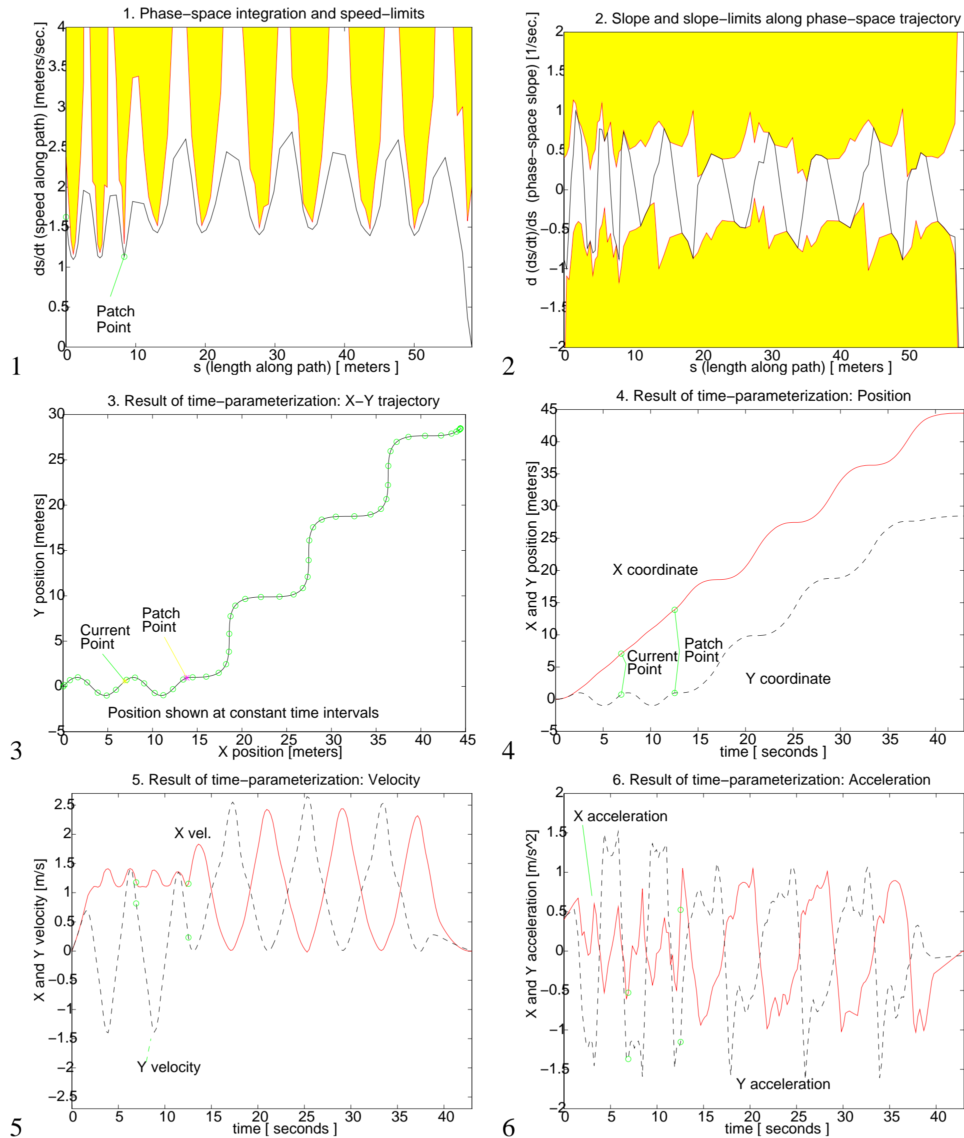


Figure 7.21: **Results after first modification to on-going trajectory**

The 3rd plot illustrates the current point in the trajectory when it is modified beyond the patch point*. The constraints (first two plots) only change beyond the patch point (compare with Figure 7.20). The trajectory, however, may change at any point beyond the current point. The full trajectory, after the modification is made, is shown in plots 4, 5, and 6.*

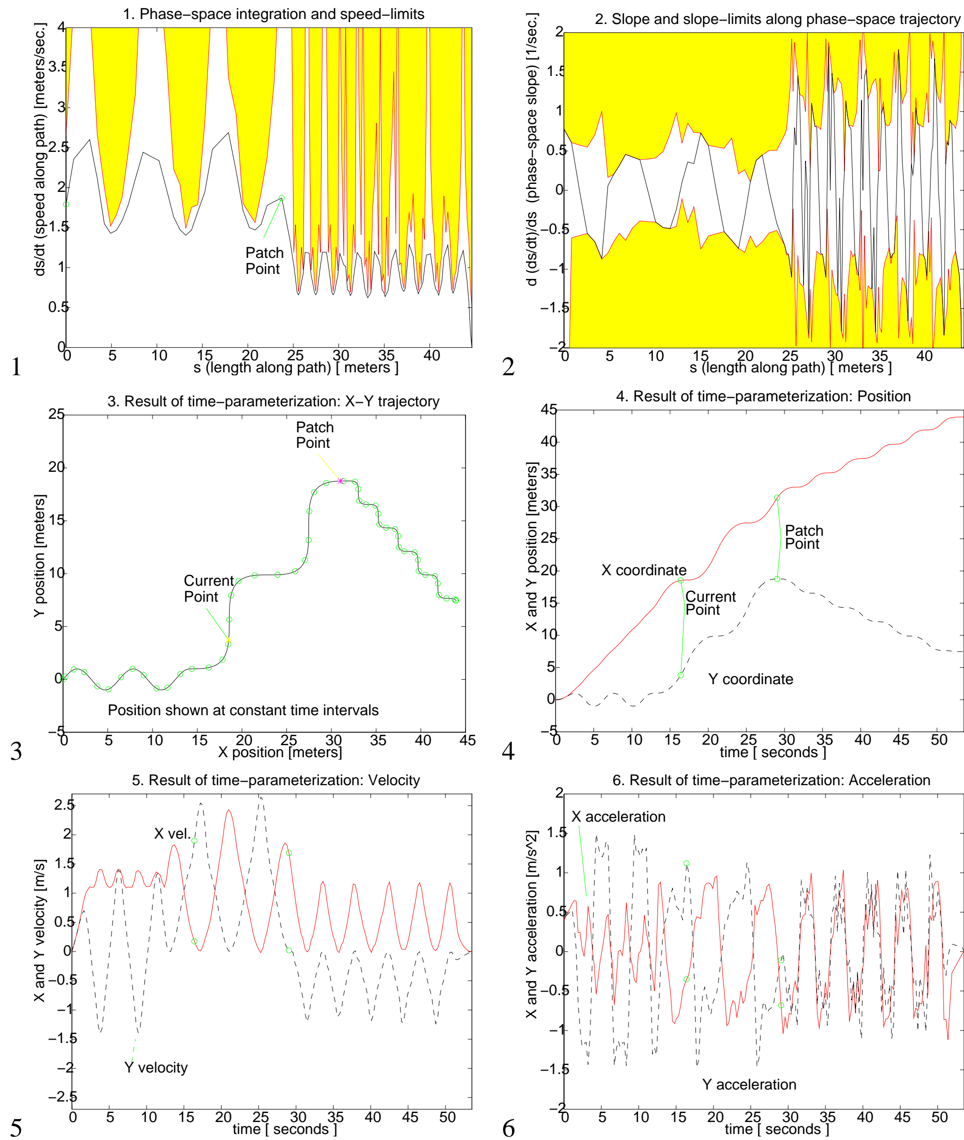


Figure 7.22: **Results after second modification to on-going trajectory**

The 2nd trajectory modification is shown in the 3rd plot. The algorithm is able to modify the trajectory while satisfying all constraints. The complete trajectory resulting from both modifications and corresponding time-parameterizations is shown in plots 4, 5, and 6.

7.7 Summary and Conclusions

This chapter has described a new (proximate-optimal) algorithm to time parameterize geometric paths described as sequences of via points. The trajectory is proximate time optimal subject to the specified dynamic constraints that can be any combination of velocity, acceleration, or torque limits, which may be configuration dependent.

The algorithm gives up strict optimality by imposing more strict (yet physically meaningful) constraints on top of the regular dynamic constraints. The proximate-optimal algorithm achieves efficient, predictable performance (run-time linear with respect to the number of via points), that enables its use on-line. The algorithm is also well suited to allow modifications (patching) of trajectories already in progress (an important feature in not-fully-structured environments).

The predictability and performance of the algorithm has been evaluated experimentally using the "canonical" sequences of via points. For the 4 DOF manipulators in the workcell, the computational time is about 15 ms per via point in a Sparc-Station 2 machine.

This algorithm is used to create all the trajectories (single-arm, dual-arm, and object) that correspond to pre-planned paths in the robotic workcell. Experimental results are presented for several trajectories computed for the workcell manipulators. The algorithm has since been used in a variety of other systems such as underwater robotic systems [104, 195], visual-tracking with a robot-mounted camera [112], and free-floating space robots [179].

Chapter 8

Conclusions

The previous chapters have presented the background, approach and results of this research in detail. This chapter concludes the thesis and is divided into three sections: The first summarizes the research and attempts to draw general conclusions from the specific results presented in the thesis. The second explores future extensions and follow-on research. The last presents some of the philosophical/methodological lessons learned by the author in the process of managing this multifaceted project which should (hopefully) be of general value.

8.1 Summary and Conclusions

This thesis comprised an experimental study of the issues involved in the development of an intelligent, dual-arm robotic workcell that includes on-line planning as an integral part of the architecture. The system combines a high-level graphical user interface (GUI), an on-line motion planner, real-time vision, and an on-line simulator to control and monitor the workcell in real-time.

The graphical user interface provides high-level user direction. The motion planner generates complete on-line plans to carry out these directives, specifying both single and dual-armed motion and manipulation. Combined with the robot control and real-time vision, the system is capable of performing object acquisition from a moving conveyor belt as well as reacting to environmental changes on-line.

The experimental objective, which was successfully met, allowed a user to specify high-level pick-and-place assembly-like tasks (e.g. the placement of multiple parts) with simple mouse motions in a graphical interface. From there on, the system performed the full operation *safely* and *autonomously*. Some of these tasks were quite complex, involving multi-step operations (such as

handing parts from one arm to another, acquiring the parts from the moving conveyor, using two arms for the odd-shaped or heavy parts etc.), control-mode changes, and on-line management of environmental changes (e.g. repositioning of any part or obstacle in the system). Safety requires not only planning collision-free motions for the manipulators, but also considering inter-part collisions as well as collisions between the parts and other objects in the workspace. The operation of the system is completely on-line; there are no fixtures or á priori scheduling.

This research addressed in detail a number of issues of general value to intelligent distributed robotic systems: the system design process, architecture and interfaces, a hierarchical workcell-control system, information sharing models and mechanisms, and automatic trajectory generation. This work has led to the conclusions that follow.

System architecture, design and integration process. Perhaps the hardest part of any project is the beginning, where both the architecture and design process need to be determined sometimes before the full scope of the project is understood[1]. In this endeavor, the interfaces-first methodology introduced in Chapter 3 was very successful:

1. The interfaces-first approach allowed the complete project to be boot-strapped, and provided architectural integrity to the system. By focusing on the information required for the operation of the workcell and developing information interfaces (rather than subsystem interfaces, which require the number and type of subsystems to be fairly well decided), the design process could be started quickly, and perhaps more importantly, the information interfaces provided the unifying thread that held the project together.

   The interfaces were (of course) refined as the project took a more defined shape; but the architectural integrity of the project was maintained, because the subsystems had to use the information with the syntax and semantics defined by the information interfaces. For instance, the fact that all the interfaces are unidirectional (with no handshaking), forced the planner subsystem to rely completely on the world-state information in order to determine what the system is doing at any time (i.e. it cannot simply assume that it is following the last command that was sent). Similarly, the fact that the world-state updates are asynchronous and subscription-based means that the planner and graphical user interface must be ready to completely change their internal models whenever a new update is received. The disadvantage is that these subsystems become more complicated; but the great bonus is that they are more

[1]In some cases "the full scope" is never really understood, since the system remains open and continually evolves in different directions.

robust and stand-alone, for they have to be designed for a continuously changing, unpredictable environment. This allowed, for instance, the operator to re-start or re-initialize the robot-control subsystem independently of the planner (or any other subsystem). This in itself is a great benefit during the development process, because new approaches and variations need to be tried constantly.

2. The use of primitive information interfaces to build custom interfaces for each subsystem was critical to the fast integration of the total system. During subsystem definition and development, the interface requirements often expand and change. Customization allows the interface to a particular subsystem to be changed with minimal impact to the remaining subsystems.

   For instance, the world-state interface allows selection of specific information interests (e.g positions of a specific object but not velocities; shape of the object but not its dynamic properties; visual cues but not grasp positions), and also allows specification of required update rates. In this manner, several human interfaces were quickly prototyped and as the sophistication of the graphics changed, so did the information needs (and manageable rates). This flexibility to adapt the interfaces, even though they were already defined, was key to keeping the system open and allowing easy exploration of new approaches.

3. The consistent use of *anonymous* information interfaces provided substantial benefits during the testing and integration of the complete system. Anonymous interfaces interconnect subsystems without their knowing of the number or identity of their peers. Consequently, the subsystems remain effectively decoupled and are more amenable to being tested in isolation.

   This is one of the main advantages of modular design per se. However, anonymous information interfaces allow the decision of which subsystems shall intervene in the final system to be delayed until run-time (or even changed at run-time). This capability provides several benefits beyond those of simple modular design: (1) the subsystems can be developed "as if" they were in the fully integrated system, since the true interface and communications are already in place. (2) Testing and normal operation can be interleaved in a variety of manners. For instance, a command sent by the planner that causes a problem can be saved into a file, and immediately replayed by using a "surrogate planner" that simply reads the file and re-sends the command[2]. This allows tackling the very difficult full-system problems that only arise

[2]The fidelity of these tests obviously depends on that of the surrounding surrogate systems, which can be improved as need arises

through the interaction of all the subsystems. Moreover, it allows one to do this "in vivo", that is without stopping and re-starting the system (as would normally be required in other re-play schemes). The resulting productivity increase was a great simplification in the integration process.

Interfaces for intelligent robotic systems. One of the contributions of this thesis is the development of experimentally proven examples of information interfaces for robotic workcells that may serve as a basis for the design of more sophisticated systems. Definition of these interfaces is non-trivial; their nature will have system-wide repercussions. Several conclusions can be drawn from the development of these interfaces:

1. The selection of fairly non-committal, strategic-level commands for the system-command interface proved very advantageous. The use of non-committal strategic-level commands gave the control subsystem the freedom to select the best available approach for each task. For instance, reference trajectories are specified as via points both in operational and joint space. This allows the control subsystem to use any available control mode, or even select (and change) control policies depending on the task.

2. The use of strategic-level commands permitted delegating the responsibility for tasks requiring high-bandwidth sensing/control to the control subsystem, and made tasks such as picking objects from the conveyor belt doable even in the presence of significant communication delays or initial uncertainty. This approach can be extended to other tasks that are typically considered the responsibility of the planner subsystem, such as fine-motion and contact, which could be described roughly by the planner and ultimately implemented with event-driven feedback strategies.

Automatic trajectory generation. The trajectory-generation problem, critical to interfacing a geometric planner with a dynamic-control subsystem, was covered in detail in Chapter 7. Automatic trajectory generation relieves the user from the most tedious of tasks: trajectory tuning.

1. The use of via-points to communicate simply the geometric paths along with additional time constraints of the resulting trajectory was very successful. First it required far less communication bandwidth than feeding set points at the sample rate of the controller (which may be unknown to the planner). Second, it is robust to communication delays, because the control subsystem is free to execute them at any time provided that the explicit time

constraints are satisfied. Third, via-points are geometrically intuitive, and can be generated in a variety of manners other than a planning subsystem. For instance, the user could select them by clicking at certain positions in the graphical interface.

2. The á priori predictability of the run time of the algorithm (based on path length alone) allowed the planning subsystem to account for this computational delay when planning trajectories for the arm to capture moving objects. The proximate-optimal time-parameterization algorithm presented in this thesis was a key component in the on-line operation of the system. Aside from being computationally efficient and predictable, it provided a simple mechanism to modify ongoing trajectories. This facility is critical to several concurrent projects in our laboratory [112].

Hierarchical control system design. The hierarchical approach decomposes the control system into several layers, each closing a control loop. Each layer presents to the layer above a more ideal system to be controlled. The hierarchical approach is well established (see references in Chapter 6); however, the definition of the layers and their interfaces is non-trivial, and yet critical. The layered decomposition used in this thesis had already been proposed by previous researchers [154, 131].

1. Isolating the lower-level details (such as non-ideal joint dynamics) from the manipulator and object-level controllers allowed standard controllers to be used for the arm and object layers.

2. The combination of a high-performance, inverse dynamics, arm controller with the singularity-free, lower-performance, joint controller provided good manipulator control over the complete workspace. The approach followed in this thesis is specific to the manipulators used, but the general characteristics may be applicable to a large class of manipulators. This research made clear the need to address this issue rigorously.

3. The design of a strategic-control layer composed of multiple finite-state machines (one per manipulator) that communicate and synchronise allowed the manipulators to act both independently and cooperatively. The use of functionally identical finite-state machines for each manipulator provided an architecture that could be easily extended to accommodate more manipulators.

World modeling and sensor integration. World modeling is necessary in order to represent á priori (often domain-specific) knowledge, and take advantage of this information to process the

sensor data itself. This thesis used a two-layer approach (see Chapter 5) where the top (object-based) layer modeled the different physical elements in the workcell (objects, manipulators, conveyors) as software objects, and the bottom (dataflow) layer was used for sensor processing.

1. The two-layer approach provided the benefits of both current object-oriented modeling techniques and "classical" sensor fusion algorithms (Kalman filtering, best-likelihood).

2. The integration of both layers by allowing the object-based layer to control dataflow (changing its parameters and rerouting the signal flow) was able to accommodate the wide variety of operating modes present in the workcell. For instance, as a result of a manipulator grasping an object, there is a kinematic chain that allows the position of the object to be inferred from that of the manipulator. The dataflow layer can be reconfigured to take advantage of this extra information and generate the missing height that must be provided to the two-dimensional vision system so it can continue tracking the object. Although this example is quite specific, the general approach should be widely applicable.

Network communication in distributed control systems. Communication is inherent to distributed systems. The identification of the specific needs of distributed robotic systems, and the subsequent development of a communication architecture (NDDS) to address these needs was one of the greatest outcomes of this work. This communication system, described in Chapter 4, provided a simple and natural mechanism for the implementation of the previously mentioned anonymous information interfaces.

1. The use of a network-transparent *publish/subscribe* interaction paradigm allowed the interfaces to be customizable in their content (for instance, the graphical interface can subscribe only to the information it needs), and provided data distribution with minimal communications delay.

2. The combination of a realistic model of time and a simple, user-customizable, mechanism for arbitrating among multiple producers was extremely successful in creating robust many-to-many communications.

3. The use of a connectionless, unreliable, protocol (UDP/IP) as the underlying layer for NDDS was essential for its fitness to real-time applications. A reliable protocol makes assumptions about the characteristics of the information that may be counter productive for real-time

applications[3]. Unreliable protocols provide only basic communications, and allow the user (through the mechanisms provided by NDDS) to customize the communication semantics to his or her needs.

Experimental demonstration of the total intelligent workcell. The experiment consisting of multi-part pick-and-place operations was engineered to emphasize the main aspects of the research. Its quite comprehensive success–summarized in Section 1.4, and described in detail in Chapter 3–illustrates the potential of intelligent workcells.

An interesting powerful possibility is the concept of *part-driven* manufacturing. In this model, the manufacturing process is driven (within certain limits) by the availability of parts, not by some pre-conceived schedule. In this manner, the workcells can adapt much more rapidly to changing manufacturing needs: more demand is met by increasing the availability frequency of the required parts. This is especially valuable if the workcells are multi-functional (i.e. are able to manufacture different products concurrently); then the out-coming product mix can be *driven* by the availability of parts rather than *changed* by reprogramming and down-loading new schedules to all the participating workcells. Part-driven manufacturing has the potential to be much more adaptable to market demands.

8.2 Suggestions for Future Research

This section outlines a number of ideas and research topics constituting possible extensions and worthwhile topics for future research. Smaller issues worth investigating in more detail are also listed below.

8.2.1 Extensions to the Architecture and Interfaces

System interfaces and command primitives. The system-command and world-state interfaces developed in this project have been tailored to the specific needs of the experiments at hand. Although an attempt has been made to make them general in their form, no effort has been spent searching for a complete self-consistent set with general applicability. These interfaces could be extended to become a language suitable for task specification, communication between planning subsystems, and world description.

[3]For instance, a lost packet may cause the whole information stream to be halted until the packet is successfully retransmitted. This may prevent more critical information from reaching its destination.

Specifically, better ways to describe the status of the robot and the possible failure conditions are needed to extend the functionality of the system while maintaining the metaphor of communicating anonymous subsystems. The research in this thesis has only begun to explore ways of expressing spatial and temporal constraints so that planning subsystems can issue self-contained commands that can be interpreted at some later time, and can contain enough freedom (i.e. are not fully committed) for the strategic layer to adapt to variations that were unknown at planning time. For instance, the current system performs the final conveyor-capture stages autonomously, directly interpolating intercept trajectories that are away from the path commanded by the planner. To ensure safety, a local collision checker will disable the robots if a near-collision situation is detected. Although the local safety check should probably remain in place (to handle unexpected events), it would be better to use a "collision-prevention" approach where the planner can tell the strategic layer the limits of the safe regions around the nominal path so that the autonomous operations (captures, contact-assemblies etc.) are safe (at least in the absence of unexpected events).

System architecture. This thesis has provided only a glimpse of the possibilities attainable by architectures composed of distributed subsystems communicating through anonymous information interfaces. The infrastructure has been set for the exploration of new ways for users and planners to interact with robotic workcells: multiple interfaces, some combining VR[4] or AR[5] capabilities can be used in a complementary manner. In this way, multiple users could collaborate and share the same workcell. Some of the same techniques should be applicable to systems other than robotic workcells: for instance, teleoperation and tele-presence facilities can be integrated to enable operation of robotic systems in completely unstructured environments (e.g. hazardous environments).

8.2.2 Enhancements to the Subsystems

Hierarchical control subsystem. In this research the focus was to develop a control system that accomplished an adequate level of performance to allow tracking of object and manipulator reference trajectories. Many important control issues were not fully addressed. For instance, an approach was presented to merge a high-performance arm controller with a lower-performance singularity-free controller to achieve stability while passing through singularities. The stability issues of such combinations were not addressed. Also, object trajectories were required to keep both arms away from singularities, which forced the arms to regrasp long objects frequently. It would

[4] Virtual Reality.
[5] Augmented Reality.

be worthwhile to investigate object-control techniques that offer the benefits of object-impedance control away from singularities, while remaining stable when one of the manipulators reaches a singular configuration.

The strategic-level operations that the system performed were limited to pick-and-place type actions. Other operations incorporating contact maneuvers (insertions, stacking, motion along surfaces while maintaining force profiles etc.) are used in manufacturing. Extension of the strategic layer to incorporate such operations will require further research in both controls and strategic task specification. Operations such as insertions of cards into slots (e.g. assembling PC boards) will require new strategic programs.

Use of adaptive control and learning techniques. Adaptive control can be used to develop control systems that are robust to certain system unknowns (e.g. mass of parts being manipulated) and whose performance can be continuously tuned (as system parameters vary with time, adaptive algorithms can modify the control parameters to compensate for this). Learning techniques can be used to bring these ideas one step higher, where complete strategies can be learned or perfected over time. For instance, a rough strategy for performing a card-into-slot insertion may be programmed ahead, but many parameters such as insertion force, frequency and direction of wiggling may be left for the system to learn at run-time.

World modeling. The world-modeling subsystem developed in this research has used fairly simple ways to describe objects geometrically (all objects are described as a set of non-intersecting parallelepipeds). To support learning, assembly planning, manipulation, and reaction to changing manufacturing needs, several extensions are required from the world-modeling subsystem. For instance, the world modeler could combine solid modeling [101][6], physical modeling (to support model-based sensing, integration and fusion of multiple sensors, adaptive control to the payload), and logical inference (e.g. creation of new objects as elementary objects are assembled, and deduction of the new physical parameters from those of its parts). Equipped with this, the world-modeling subsystem could be used to predict contact forces, friction coefficients, equilibrium of parts when placed on top of each other, etc.

And a major advance lies in drawing upon human perception ability. Currently the information in the world modeler is either available at initialization time (through configuration files), directly sensed at run-time, or derived from the existing information though the sensor-integration/fusion

[6]Similar to CAD systems, solid modeling can be used for assembly, grasp and motion planning.

process. For systems that must remain continuously operational, it may be advantageous to provide powerful new mechanisms by which the user can add new information at run time–mechanisms by which the superior perception and inference capability of the human can be drawn upon, with great benefit. For instance, the geometry of a new part could be described by the user interactively. Different ways to grasp an existing part can be suggested by the user through a suitable interface. Concurrent research at ARL by Eric Miles is pursuing this direction.

Automatic trajectory generation. The proximate-optimal trajectory generator developed as part of this research trades optimality in the final path for computational efficiency and predictability of the algorithm itself. Although these issues have been demonstrated in worst-case-type scenarios, and during the experimental operation of the system, a precise characterization of this tradeoff is lacking. In other words, one would like to answer questions like: How far away from optimal are the proximate-optimal solutions? For which paths is the proximate-optimal solution optimal? Are there paths for which the proximate-optimal solution is very suboptimal? If so how are they characterized?

Geometric paths have been described using via-points, however for every set of via points there are infinite continuous paths that go through them. The use of splines provides a physically intuitive interpolation mechanism[7]. In the current implementations, via-points are equally spaced in joint space (for the Z degree of freedom, 0.05 m is weighted as 1 radian). This could be improved so that via points reflect the geometrical regularity of the path: portions of the path that are geometrically complicated (small curvature, rapid changes in the curvature etc. as when avoiding some obstacle) use a greater density of via points (points per unit length) for their description.

8.2.3 Higher Degree of Planning Integration

Use of multiple planners. The anonymous information interfaces provide mechanisms for multiple planners to communicate and command the workcell. This functionality could be exploited to allow optimistic, computationally-incomplete planners (which may arrive to a solution quickly in most cases, but either fail or take an unbounded time for some problems) in parallel with slower, complete planners. This could even be combined with ways in which a human could provide hints on how to solve pathological situations. Such systems have the potential to be more robust.

[7]The name spline is derived from the thin wood or metal strips used for technical drawings by pinning them at certain points to generate smooth interpolating curves.

By keeping the door open for human intervention, the transition to full automation can be done incrementally (and perhaps be more easily accepted by end-users).

Integration of assembly and fine-motion planning. The experiments described in this thesis have integrated manipulation planners developed at the Stanford Computer Science Robotics Laboratory [91] within the architecture described in this thesis. Researchers at the Computer Science Robotics Laboratoryhave demonstrated assembly and fine-motion planners in simulation [199, 77]. Integrating these planning techniques into the intelligent workcell could provide significant new functionality, (For instance instead of the user describing manually the grasp positions, assembly sequences and "approach points", they could be automatically generated by the planners.) Integrating these new planners will require modifications to the world-state and system-command interfaces (to communicate the extra information required) as well as modifications to the planners themselves (so that they can operate on-line in a dynamic, uncertain environment).

Planning of strategic programs. Finite-State Machines (FSMs) have been used to program the strategic-control layer. All these programs have been hand-generated (using a graphical editor). This is time consuming and requires the user to pre-establish and program these behaviours in advance. On the other hand, these FSMs could be automatically generated by a planner or even learned using techniques such as genetic programming [84]. The automatic generation of FSM programs would allow the system to enhance its functionality autonomously (without human intervention) and partially program itself.

8.2.4 Enhancements to the Communications Layer.

The NDDS package developed in this thesis provides facile network connectivity that is tailored to the requirements of distributed robotic systems. The model of information exchange between anonymous entities is quite powerful, but presumes communications on a safe environment. To be truly scalable, security issues must be addressed: What if extraneous entities participating in the communications unintentionally (or maliciously) access the information exchanged in the system and/or send unwanted messages? We need mechanisms to restrict and enforce the identity of the participants in the communications process. For instance, public-key encryption could be used to prevent unauthorized access to the information; special capabilities may be required to be appended to the messages before they are accepted by the communication endpoints.

The naming mechanisms can also be extended to allow grouping related information into subjects and subscribing to complete subjects without knowledge of the specific information they contain.

Publish/subscribe communications have been shown to be more efficient for repetitive updates. However, they do not provide natural flow-control of client/server-type interactions. For instance, a slow consumer may be overly optimistic on the amount of updates that it can handle, and then be overwhelmed by the fast up-dates. Mechanisms to detect these situations and correct them automatically should be investigated.

Several hardware-independent data representations other than XDR have been proposed. Some of these representations are self-describing (allowing the receiver to interpret the data at run-time). Self-description is useful for applications such as data-browsers that don't know the format of the information ahead of time. However self-describing representations are usually less efficient due to the need for interpreting the data. An intermediate approach that may provide the best of both worlds may consist of pre-compiling the necessary functions and dynamically loading them as need arises.

8.2.5 Possible Follow-on Experiments.

This thesis has provided a first-step in the experimental demonstration of an intelligent workcell that can be commanded by a user at a high level. The richness of possible tasks can be extended dramatically with the addition of more manipulators, or manipulators with configurations other than the SCARA. Use of more sophisticated vision techniques could allow elimination of the required LED-tagging of all the objects. Also precise acquisition and contact tasks with small parts could be enabled by the use of an arm-mounted camera. Drawing upon human perception in world-modeling as mentioned before, could help in this endeavor.

The set of tasks that can be planned and commanded is so far quite restrictive. The addition of contact and assembly tasks would allow the system to attack a wider range of real problems such as PC-board assembly.

8.3 Concluding Remarks

This has been an experimental thesis. Experimental verification is likely to be far more time consuming than the preliminary, tentative reassurance offered by simulation. But the far stronger verification of an experiment is several-fold. First it demonstrates that one has really not overlooked some important aspect that may render the simulated approach invalid. Second, it forces

a prioritization of the research consistent with that of "real" problems, avoiding the "solution-in-search-of-a-problem" fallacy. But perhaps of most importance, it brings credibility to the whole project. (The old saying: "It works, therefore something must be right" is deeply engraved in the minds of most people.)

Experiments do offer a potential danger in research, that of being bogged down into solving technical problems without addressing any fundamental issues. To avoid this, it is important to identify the research issues early, and choose carefully the experiments to perform, focusing on the essential aspects and finessing the unnecessary details.

This project benefited from the participation of several people from two different laboratories[8]. The following recommendations follow from the author's experience as project leader.

Fix the interfaces first. To preserve architectural integrity interfaces should be the responsibility of few, preferably one, person; however, they should not be defined in isolation. Early informal meetings/discussions should be used to shape the interfaces. The resulting interface specifications should be made constantly available to the whole team to encourage revision and focus effort.

Motivate the whole team. Everybody must gain from the total success of the project. Do not fall into the blame-assignment game. Ultimately it doesn't matter whose fault it is, only whether the total system works.

Manage the project. Select milestones, deadlines etc. with specific experimental demonstrations of functionality. Reflect about goals and schedules. This provides sanity checks, gets the project focused, reveals design flaws/shortcomings early, prevents time-consuming excursions, and avoids the infamous "creeping functionality syndrome"[9]. The above tendency must be balanced with the temptation to micro-manage the project which could be fatal (especially in an academic/research environment) because bright, creative people need the freedom and time to explore in new directions. The balance can be achieved by selecting reasonably spaced milestones.

Intersperse periods of intense colocated team-work previous to each milestone with periods of relatively independent work. Having all the team participate in the system integration is sometimes necessary; but even when it is not so, having the team involved (or at least present) in the process gives valuable global perspective which will influence the work of each team member.

Share and discuss problems and progress with your peers. Be open to criticism and ready to justify and defend everything you do. Continuous catharsis is the key to solid designs. Software is the ultimately sharable material so whenever possible develop and package software in such way

[8]A total of six people from the Stanford Aerospace Robotics Laboratory and the Stanford Computer Science Robotics Laboratory were involved in the project at one time or another.

[9]This has also been referred as the "second system effect" (Brooks [23]).

that it can be used by other people outside the project (and then encourage them to do so). The time spent supporting and enhancing software for other people will result in increased quality and reliability, and be rewarded through reciprocity and the resulting team atmosphere.

Keep an open, no-discipline-boundaries attitude. Be ready to do anything and learn anything that is required by the project. In a large multidisciplinary project one will be exposed to many areas where one's own particular expertise is lacking. Whether the effort should be spent on it should be determined by how critical it is for the project, not "whether this is my stuff". Time spent learning something is far more efficient (and enjoyable) than time spent trying to side-step the issue, patch a solution, or convince somebody else to help or fix it for you. Knowledge is the most important thing you will carry away with you. Effort spent in ad-hoc solutions is ultimately wasted.

Seek and use anything already available; do not "reinvent the wheel". This applies equally to software tools, computer algorithms, mechanical designs, and control methodologies. (This is perhaps the hardest recommendation to follow, which regretfully I did not master until the later stages in the project.) The temptation is strong, because as researchers we take pride in our work, and conceiving our own solution will seem far more satisfying than crediting somebody else for theirs. However, one should focus the effort instead into adding to the existing knowledge. Perhaps the only way (at least for me) to avoid this trap is to conduct early literature reviews and peer discussions before embarking on a new project. Once a substantial personal effort has been invested, it is hard (sometimes unbearably so) to drop it in favor of somebody else's approach. As a result, extra effort will be spent (wasted) trying to differentiate one's approach from that of others.

Learn to enjoy what you do[10]. Take advantage of the freedom of graduate research, and the rich university environment. Seldom will you find such an intellectually stimulating environment.

[10] Also do what you enjoy; but this is the easiest part.

Appendix A

Calibration

A.1 Vision System Calibration

The wide angle lens (6mm) and the inexact verticality of the camera's axis introduce geometrical distortions in the camera image. Optical distortions are always of odd order due to symmetry. Deviations from exact verticality introduce a linear-type distortion while the primary lens distortions (pin-cushion, barrel) introduce third-order distortions. To compensate for this, a jig consisting of an array of equally-spaced LED's spanning most of the field of view was placed on the table, and the coefficients of a third-order multivariate polynomial fit to the known position of the LED's where obtained. With this calibration, absolute accuracies of 3 mm over the entire workspace were achieved as seen in Figure A.1

A.2 Arm Base Location Calibration

In order to use kinematic information to deduce the position of the arm end-point in the global (vision) frame as explained in section 5.4.1, the location of the manipulator bases, and the arm kinematic parameters must be accurately identified. The approach taken consists of sweeping the manipulator to a grid of locations in the workspace[1], while collecting both vision data and joint-angle data.

[1]This grid was defined by dividing stepping the shoulder angle in 0.5 rad increments between -1 and 1 rad (5 steps), and for each shoulder angle, stepping the elbow angle in 0.9 rad increments between -1.8 rad and 1.8 rad (5 steps). This forms a grid of 25 points that covers the workspace of the arm and is symmetric with respect to arm handedness.

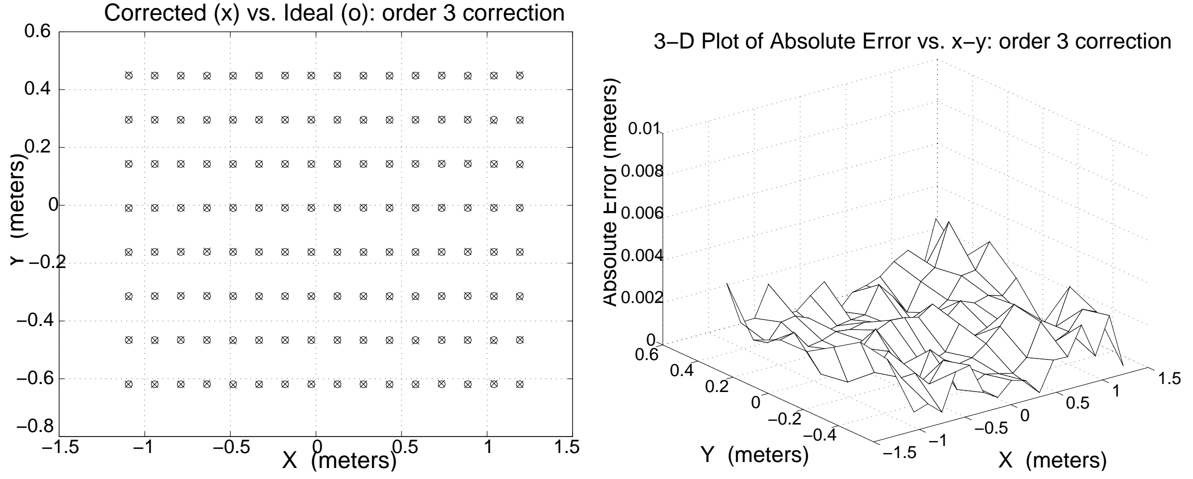


Figure A.1: **Vision system calibration**

The above figures show the discrepancy between the coordinates reported by the vision system and the true coordinated after vision-system calibration. In the left figure, the crosses indicate vision system coordinates and the circles the true coordinates of each one of the LED's in the calibration grid. The correspondence is excellent. The right figure illustrates the errors over the complete vision field. It is clear that the calibration has removed most of the distortion introduced by the lens. The maximum error over the whole vision field is $3.8mm$.

For each joint location $(\theta_{sh}(k), \theta_{el}(k))$, the position of the visual fiducial $(x_{kin}(k), y_{kin}(k))$, can be derived from arm kinematics:

$$x_{kin} = x_0 + l_0 \cos(\theta_{sh} + \theta_{sh}(0)) + l_1 \cos(\theta_{el} + \theta_{el}(0))$$

$$y_{kin} = y_0 + l_0 \sin(\theta_{sh} + \theta_{sh}(0)) + l_1 \sin(\theta_{el} + \theta_{el}(0))$$

where (x_0, y_0) is the position of the shoulder axis in the global frame, l_0 is the length of the shoulder link, l_1 is the length from the elbow axis to the visual fiducial in the elbow link, and $\theta_{sh}(0)$ and $\theta_{el}(0)$ are initial offsets that relate measured angles to the global frame.

This data is post-processed in MATLAB to identify the values of the the unknown parameters $(x_0, y_0, l_0, l_1, h, \theta_{sh}(0), \theta_{el}(0))$ that minimize the discrepancy between kinematics and vision as measured by the performance index:

$$J(x_0, y_0, l_0, l_1, h, \theta_{sh}(0), \theta_{el}(0)) = \sum_k \sqrt{(hx_{vis}(k) - x_{kin}(k))^2 + (hy_{vis}(k) - y_{kin}(k))^2}$$

The parameter h results from the fact that the visual fiducial on the elbow link isn't located at the height where the vision was calibrated (table height), and represents the ratio of visual-fiducial-height to camera-height (all heights are measured from the table).

The result of the identification for the right arm illustrated in Figure A.2. This figure illustrates that the method achieves a correspondence better than 4mm over the complete sweep (which is consistent with the vision error). The identified parameters are listed in Table A.1.

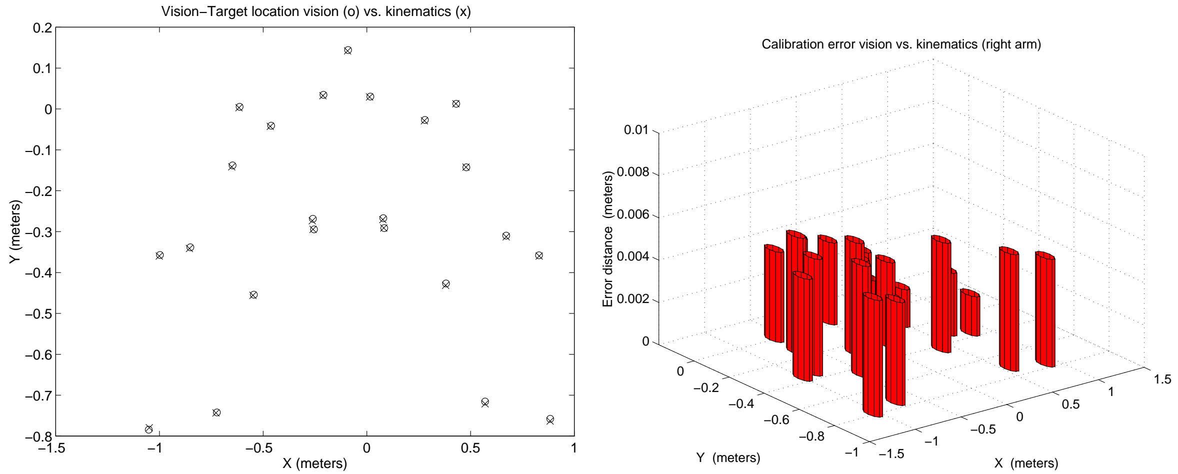


Figure A.2: **Kinematic calibration**

The above figures show the discrepancy between the arm endpoint coordinates reported by the vision system and the arm kinematics after kinematic calibration. The correspondence is excellent as corroborated by the right figure which illustrates the difference between kinematics and vision at each one of the previous arm configurations. The worst case discrepancy is under $4mm$*. This error is mostly due to the vision system.*

	x_0	y_0	l_0	l_1	h
right arm	-0.0831 m	-.9445 m	0.6096 m	0.4778 m	0.62 m
left arm	-0.1057 m	0.76 m	0.6096 m	0.4871 m	0.62 m

Table A.1: **Calibrated arm parameters**

In the final identification, the known length of the shoulder link ($0.6096m$) *was kept constant since it didn't significantly affect the quality of the fit.*

A.3 Inertial Properties Measurement

The inertial properties of the manipulator links and objects were directly measured. The masses using an electronic scale accurate to 0.1 gram. The center of mass positions were determined by balancing the parts on two knife-edges, measuring the support forces on each knife, and calculating

the mass center locations consistent with such forces. Most parts were symmetric so only one coordinate of the center of mass needed to be measured. Only one of the objects required two coordinates to be measured in this manner.

The inertias were measured using a trifilar pendulum built by Hollars (see appendix A of [66]). The system was later automated by Schneider and Pfeffer [156] and upgraded by Ims [68]. It is accurate to about 2%.

Some of the inertias could not be measured directly because the bearings connecting different degrees of freedom could not be separated (this was the case for the shoulder and elbow bearings as well as the ones in the the spline-screw assembly), however due to their location close to the rotational axis, their impact on the overall inertia of the corresponding link was estimated to be less than 1%.

Appendix B

Joint Control Parameters

subsystem	Φ	Γ	L_p	H
right shoulder	$\begin{pmatrix} .9675 & -.1990 \\ .1990 & .9625 \end{pmatrix}$	$\begin{pmatrix} .7933 & -.7157 \end{pmatrix}$	$\begin{pmatrix} 2.2174 \\ -.7687 \end{pmatrix}$	$\begin{pmatrix} .7933 & .7157 \end{pmatrix}$
left shoulder	$\begin{pmatrix} .9691 & -.1990 \\ .1990 & .9649 \end{pmatrix}$	$\begin{pmatrix} .7975 & -.7198 \end{pmatrix}$	$\begin{pmatrix} 2.2264 \\ -.7775 \end{pmatrix}$	$\begin{pmatrix} .7975 & .7198 \end{pmatrix}$
right elbow	$\begin{pmatrix} .9422 & -.2766 \\ .2766 & .9322 \end{pmatrix}$	$\begin{pmatrix} .6102 & -.5273 \end{pmatrix}$	$\begin{pmatrix} 2.5333 \\ -.4918 \end{pmatrix}$	$\begin{pmatrix} .6102 & .5263 \end{pmatrix}$
left elbow	$\begin{pmatrix} .9560 & .2604 \\ -.2604 & .9522 \end{pmatrix}$	$\begin{pmatrix} .5994 & .5241 \end{pmatrix}$	$\begin{pmatrix} 2.6072 \\ .5829 \end{pmatrix}$	$\begin{pmatrix} .5994 & -.5241 \end{pmatrix}$

Table B.1: **Estimator parameters for the right shoulder motor plant.**

This table summarizes the parameters for the state-space predictive-estimator used to control the right shoulder motor plant. This parameters were obtained using a LQG/LQE design method for a sample rate of 360 Hz.

subsystem	N_x	N_u	K
right shoulder	$\begin{pmatrix} .7391 \\ .5780 \end{pmatrix}$	.1752	$\begin{pmatrix} .9946 & .1323 \end{pmatrix}$
left shoulder	$\begin{pmatrix} .7293 \\ .5812 \end{pmatrix}$	.1734	$\begin{pmatrix} 1.0017 & .1366 \end{pmatrix}$
right elbow	$\begin{pmatrix} 1.0003 \\ .7389 \end{pmatrix}$	.4298	$\begin{pmatrix} 1.0464 & -.0830 \end{pmatrix}$
left elbow	$\begin{pmatrix} .9788 \\ -.7887 \end{pmatrix}$	.4145	$\begin{pmatrix} .9982 & .0745 \end{pmatrix}$

Table B.2: **Controller and parameters for the right shoulder motor plant.**

This table summarizes the parameters for the state-space controller and predictive-estimator used to control the right shoulder motor plant. This parameters were obtained using a LQG/LQE design method for a sample rate of 360 Hz.

Appendix C

Arm Control Parameters

$diag(\mathbf{M_{imp}})$	$diag(\mathbf{K_p})$	$diag(\mathbf{K_v})$	$diag(\mathbf{K_p^{pid}})$	$diag(\mathbf{K_v^{pid}})$
$\begin{pmatrix} 1.0Kg \\ 1.0Kg \\ 1.0Kg \\ 1.0Kg\,m^2 \end{pmatrix}$	$\begin{pmatrix} 280N/m \\ 280N/m \\ 250N/m \\ 80N/rad \end{pmatrix}$	$\begin{pmatrix} 20N/m\,s \\ 20N/m\,s \\ 10N/m\,s \\ 2N/rad\,s \end{pmatrix}$	$\begin{pmatrix} 450N\,m/rad \\ 450N\,m/rad \\ 250N\,m/rad \\ 80N\,m/rad \end{pmatrix}$	$\begin{pmatrix} 30N\,m\,s/rad \\ 30N\,m\,s/rad \\ 10N\,s/m \\ 2N\,m\,s/rad \end{pmatrix}$

Table C.1: **Parameters for the arm-level controllers.**

The above are all diagonal matrices. The notation $diag(\mathbf{v})$ where $\mathbf{v}$ is a vector indicates the diagonal matrix constructed by using the elements of $\mathbf{v}$ along the diagonal.

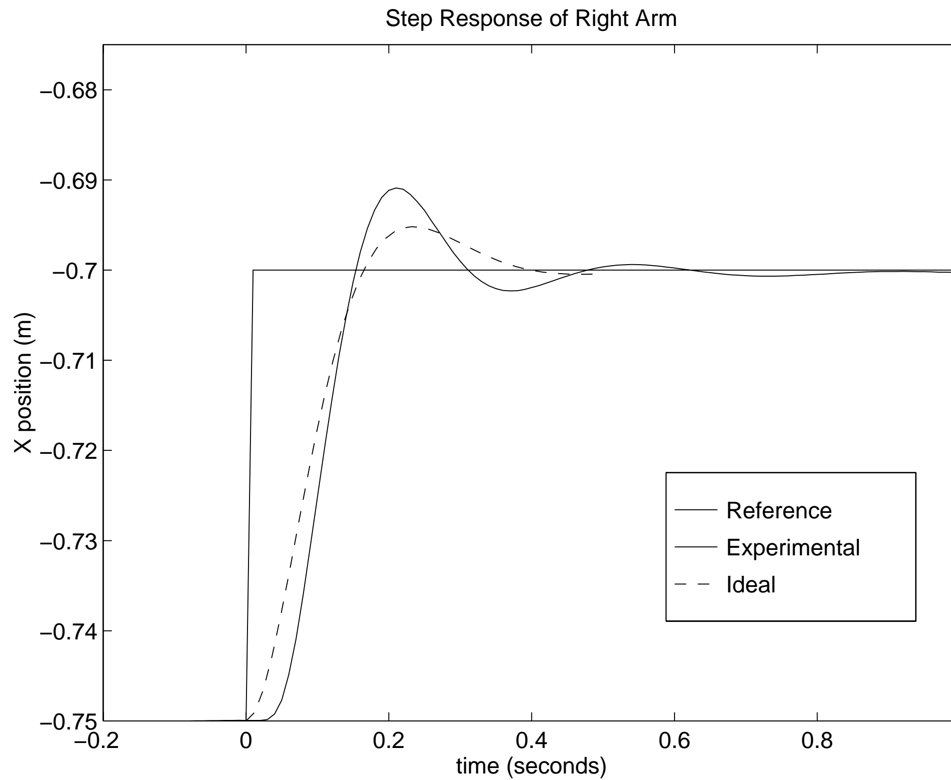


Figure C.1: **Step Response of Inverse-Dynamic Arm Controller**

This figure illustrates a 5 cm step response for the right manipulator arm. The size of this step was limited to prevent actuator saturation. The speed of the arm step response is consistent with the expected second order behavior and closed-loop bandwidth of 3.2 Hz. However, the experimental response shows significantly less damping (the selected values of K_p *and* K_v *should correspond to a damping ratio of 0.6) and is slightly delayed at the onset. The discrepancies are due to the imperfect cancellation of the lower-level control layers. The initial delay of 20 ms with respect to the ideal trajectory is consistent with the closed-loop bandwidth of the underlying joint-control loop (40 Hz).*

Appendix D

Manipulator Equations of Motion

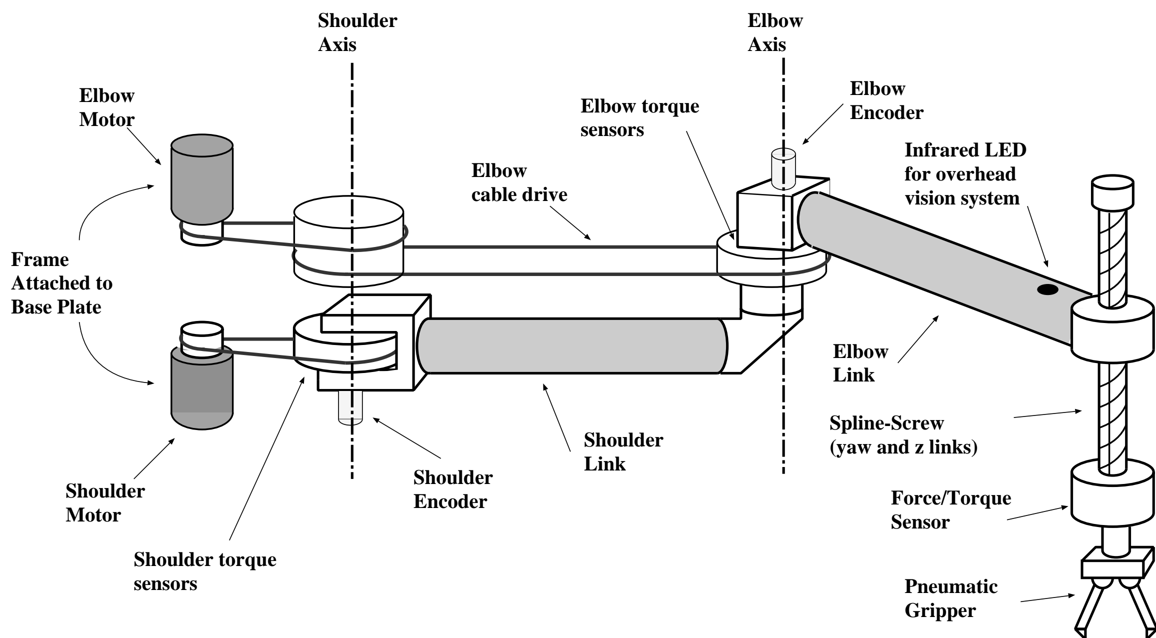


Figure D.1: **Arm Schematic**

Note in Figure D.1 that the shoulder motor applies its torque directly between the base and the elbow link, not between the shoulder and elbow link as is often the case in SCARA manipulators. The kinematic and dynamic equations for are simplest if the coordinates $\mathbf{q}$ are chosen as illustrated in Figure D.2. Table D.1 contains definitions of the different symbols used in this equations. The value of the different masses and inertias was measured before the arm was assembled with the aid of

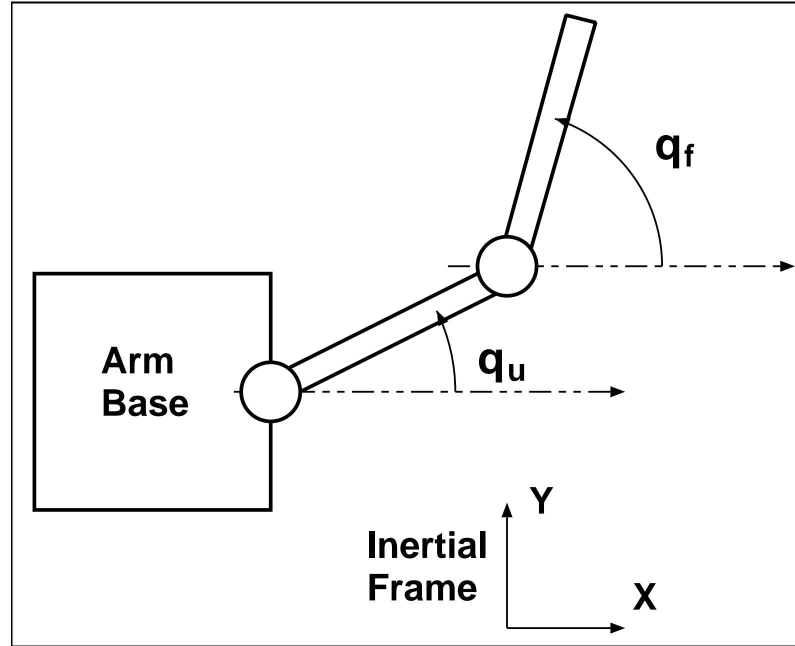


Figure D.2: **Manipulator Coordinates**

The joint angles q_u (shoulder) and q_f (elbow) are measured with respect to an inertial frame (i.e. th elbow angle is not the relative angle to the shoulder link).

an inertia measurement system as described in section A.3 in appendix A. The inertia-measurement system was developed by previous researchers [66, 156].

With the above choice of coordinates, the different matrices used in the equations of motion of the manipulator (see Equation 7.1) have the following form:

$$\mathrm{M}(\mathbf{q})\mathbf{q} = \begin{pmatrix} J_u & cos(q_{fu})\ J_{fu} & 0 & -J_y \\ cos(q_{fu})\ J_{fu} & J_f & 0 & -J_y \\ 0 & 0 & M_z & 0 \\ 0 & -J_y & 0 & Jy \end{pmatrix} \ddot{\mathbf{q}} \tag{D.1}$$

$$\mathbf{B}(\mathbf{q})[\mathbf{q}, \mathbf{q}] = \begin{pmatrix} -J_{fu}\ sin(q_{fu})\ \dot{q}_u^2 \\ J_{fu}\ sin(q_{fu})\ \dot{q}_f^2 \\ 0 \\ 0 \end{pmatrix} \tag{D.2}$$

$$\mathbf{g}(\mathbf{q}) = \begin{pmatrix} 0 \\ 0 \\ -g\ M_z \\ 0 \end{pmatrix} \tag{D.3}$$

$$\mathbf{q} = \begin{pmatrix} q_u \\ q_f \\ z \\ \phi \end{pmatrix} \tag{D.4}$$

(D.5)

Similarly, the manipulator Jacobian introduced in Equation (6.5) has the form:

$$\mathbf{J}(\mathbf{q}) = \begin{pmatrix} -l\cos(q_u) & -l\sin(q_f) & 0 & 0 \\ l\cos(q_u) & l\cos(q_f) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{D.6}$$

(D.7)

Parameter	Meaning	value
q_u	upper-link (shoulder) angular position	N/A
q_f	fore-link (elbow) angular position	N/A
q_{fu}	elbow relative angle	$q_f - q_u$
J_u	Effective shoulder joint inertia at $q_{fu} = 90°$	$3.32 Kg\ m^2$
J_f	Effective elbow joint inertia at $q_{fu} = 90°$	$1.40 Kg\ m^2$
J_{fu}	Inertia coupling term	$1.51 Kg\ m^2$
M_z	Effective mass lifted along the Z axis	$0.08 Kg$
J_y	Effective rotational inertia of last two links along Z axis	$0.01 Kg\ m^2$

Table D.1: **Meaning of symbols in dynamic equations for the arms.**

This table defines the symbols appearing in the dynamic equations for the arms and lists the values of the mass parameters. M_z and J_y account for the gearing of the elbow-mounted motors driving the last two degrees of freedom.

Appendix E

Canonical Paths for Time-Parameterization

To test the algorithm's performance and obtain experimental information on its time-complexity, we need "canonical" paths with different number of via points/DOF. This sequences of via points must be ergodic in their geometrical properties. Our approach has been to generate a long sequence of via points and extract from it sub-sequences of different lengths.

We have used smoothed random walks to generate our sequences of via points. A random walk is inherently ergodic and can be used to generate arbitrarily long sequences of via points in any number of degrees of freedom. Smoothing the random walk brings two benefits: It clusters together via points in a manner proportional to the local curvature and, it produces "smoother" paths that can be traversed at higher speeds by the manipulator without exceeding its dynamic limits. We have used a zero-phase forward and reverse digital low-pass butterworth filter (smoother). The parameters used for the examples in this paper are: A random-walk step of 1 radian in each degree of freedom, and an order N_{dof} low-pass butterworth filter with a spatial cutoff frequency of 0.15.

Figure E.1 illustrate 2 DOF paths with increasing number of via points. These type paths were used for the complexity simulations of Figure 7.6 in Chapter 7. In particular, the first path corresponds was used to obtain the results in Figure 7.2 and Figure 7.4 (see Chapter 7).

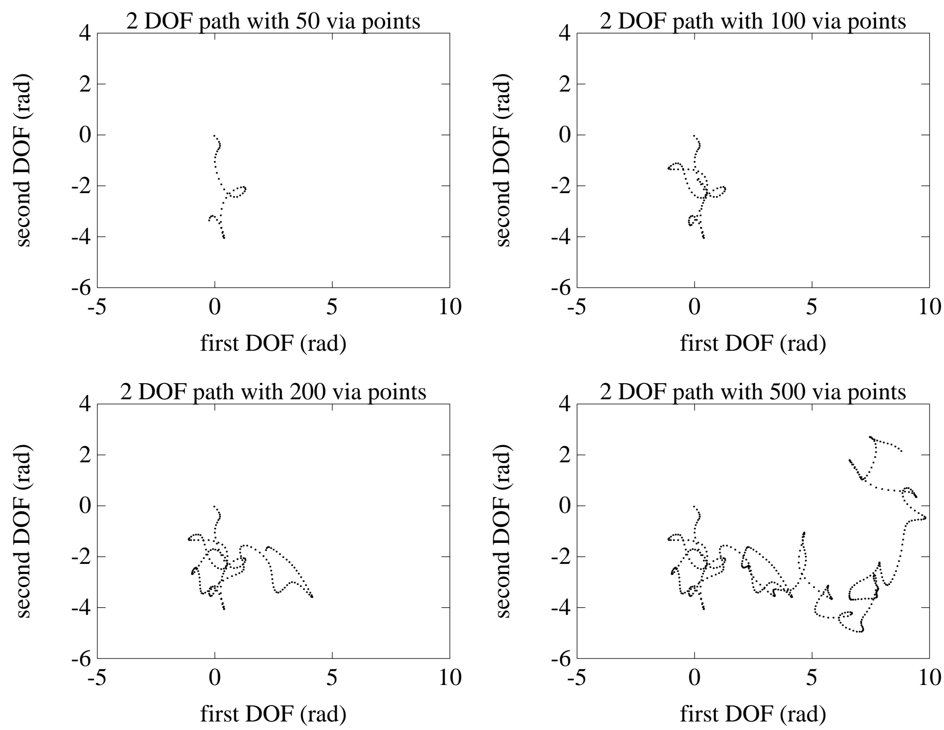


Figure E.1: **Filtered random walks of different length for 2 degrees of freedom**

These four sets of via points represent paths of increasing length. These sequences are ergodic in their geometrical properties. We have shown sets of 50, 100, 200 and 500 via points.

Appendix F

Worst-Case Paths for Time-Parameterization

Our goal is to construct a simple, analytically tractable instance of the DOTP problem for which the "classical" time-parameterization algorithms have run times that scale with the squared of the path length ($\mathcal{O}(\mathcal{L}^{\in})$). That is, we will construct paths exhibiting the behavior sketched in Figure 7.8 (Chapter 7). To this end, we will build the geometric path in 2 dimensions as a piecewise connection of $2N + 1$ circular arcs with varying radii, here N is increased to obtain paths of increasing length:

$$\text{for} \quad s \in [L_k, L_{k+1}[\quad \text{define} \tag{F.1}$$

$$q_1(s) = x_k + A_k cos(s/A_k + \phi_k) \tag{F.2}$$

$$q_2(s) = y_k + A_k sin(s/A_k + \phi(k)) \tag{F.3}$$

$$A_k \stackrel{\text{def}}{=} |N - k| + 2 \tag{F.4}$$

$$\phi_k \stackrel{\text{def}}{=} \phi_0 + \sum_{i=1}^{k} L_i(1/A_{i-1} - 1/A_i) \tag{F.5}$$

$$x_k \stackrel{\text{def}}{=} x_0 + \sum_{i=1}^{k} (A_{i-1} - A_i) \cos(\frac{L_i}{A_i} + \phi_i) \tag{F.6}$$

$$y_k \stackrel{\text{def}}{=} y_0 + \sum_{i=1}^{k} (A_{i-1} - A_i) \sin(\frac{L_i}{A_i} + \phi_i) \tag{F.7}$$

$$L_k \stackrel{\text{def}}{=} k\pi \tag{F.8}$$

$$\tag{F.9}$$

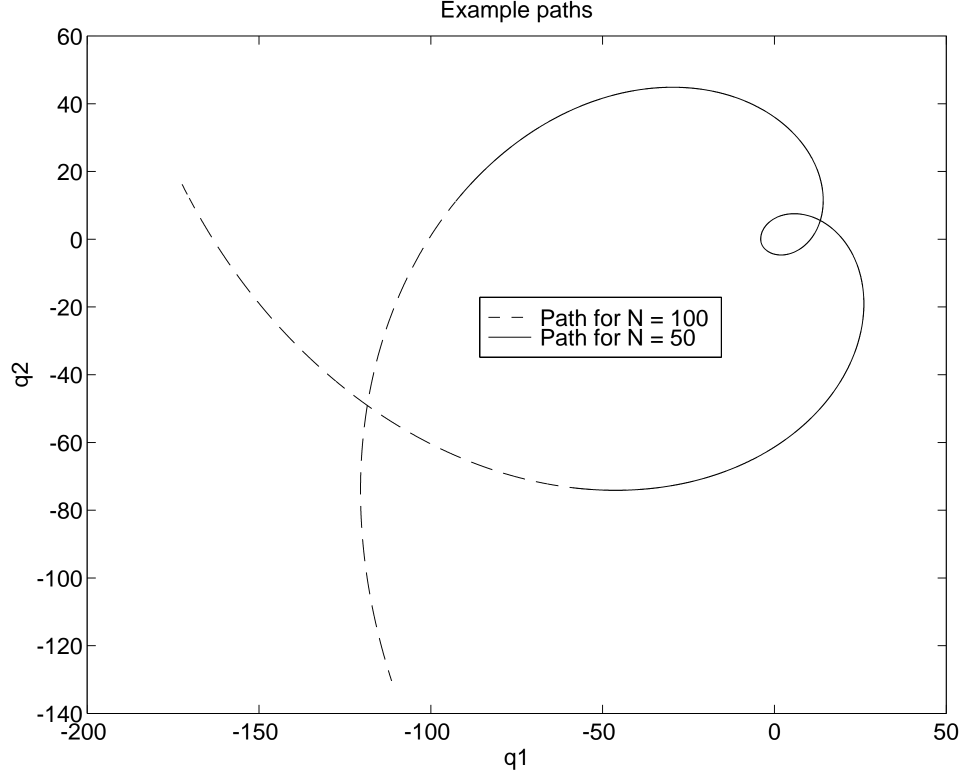


Figure F.1: **Two worst-case paths of different lengths. Each paths is composed of $2N$ circular arcs.**

Equations F.2 thru F.4 define a path composed of N arc pieces of decreasing radii followed by N pieces of increasing radii. Equations F.5 thru F.7 represent the centers and phase shifts defined so that the resulting path is continuous and has a continuous tangent. ϕ_0, x_0 and y_0 are chosen so that $q_1(N/2) = q_2(N/2) = 0$. Figure F.1 shows two example paths for different values of N. Notice that the path is are symmetrical about the $k = N + 1$ segment.

We will use maximum acceleration $|\ddot{\mathbf{q}}| \leq a \stackrel{\text{def}}{=} 0.5$ as the constraint. It is easy to show that the constraints 7.13 reduce to

$$0 \leq \dot{s} \leq \dot{s}_{max}(s) = \sqrt{aA_k} \stackrel{\text{def}}{=} \dot{s}_{max}(k) \tag{F.10}$$

$$\left|\frac{d\dot{s}}{ds}\right| \leq \frac{a}{\dot{s}}\sqrt{1 - \frac{\dot{s}^4}{\dot{s}_{max}(k)^4}}, \quad s \in [L_k, L_{k+1}] \tag{F.11}$$

For a given $N >= N_0$, we will now show that starting from $\dot{s} = 0$ the maximum acceleration curve $\dot{s} = \dot{s}_{acc}(s)$, will leave the admissible region of phase space limited by the curve $\dot{s} = \dot{s}_{max}(s)$

at a point $s_0(N) \leq L_{N/2}$. This can be proven by contradiction, assuming $s_0(N) > L_{N/2}$ we would have in the interval $0 \leq s \leq L_{N/4}$, $0 \leq k \leq N/4$:

$$\dot{s}_{acc}(s) \leq \dot{s}_{max}(N/2) = \sqrt{N/2+2} \tag{F.12}$$

$$\dot{s}_{max}(k) < \dot{s}_{max}(N/4) \tag{F.13}$$

$$\frac{d\dot{s}_{acc}}{ds} > \frac{a}{\dot{s}_{acc}}\sqrt{1 - \frac{\dot{s}_{max}(N/2)^4}{\dot{s}_{max}(N/4)^4}} \tag{F.14}$$

$$\frac{d\dot{s}_{acc}}{ds} > \frac{P(N)}{\dot{s}} \quad \Rightarrow \quad \dot{s}_{acc}(s) \geq w(s) \tag{F.15}$$

$$\text{where} \quad w(s) \quad \text{verifies } \frac{dw}{ds} = P(N)/w \tag{F.16}$$

$$w(s) = \sqrt{2P(N)(s-s_0) + w(s_0)^2} \tag{F.17}$$

$$P(N) \stackrel{\text{def}}{=} a\sqrt{1 - \frac{(N/2+2)^2}{(3N/4+2)^2}} \tag{F.18}$$

$$w(0) = 0 \quad \Rightarrow \quad w(L_{N/4}) = \sqrt{\frac{N}{2}\pi P(N)} = \sqrt{\frac{N}{2} + (\pi P(N) - 1)\frac{N}{2}} \tag{F.19}$$

Therefore, since $P(N)$ is strictly increasing and for $N = 31$, $(\pi P(N)N/2 - 1) > 2$ we see that for $N > 31$, $\dot{s}(L_{N/4}) \geq w(L_{N/4}) > \sqrt{N/2+2}$ which contradicts $\dot{s}_{acc}(s) \leq \dot{s}_{max}(N/2)$. Hence, we have proved that for $N > 31 \quad \Rightarrow \quad s_0(N) \leq L_{N/2}$.

We can see from Figure F.2 how the situation sketched in Figure 7.8 occurs in this example for $s \geq s_0(N) \in [L_{m(N)}, L_{m(N)+1}[$. Where we have just proven that $m(N) < N/2$. The subsequent forward integration along the maximum acceleration curve will leave the allowed region at each point $s = L_k$ with $m(N) < k < N$. Shin's algorithm [164] then integrates backwards from the point $\dot{s} = \dot{s}_{max}(k+1)$ until it intersects the accelerating curve. However, we will show below that for most of these points (specifically $k < r(N) = N - 80$), this integration must proceed all the way back to $s = L_{m(N)+1}$. The reason is shown in Figure F.2. For $k < r(N)$ Each backward integration from $(s = L_{k+1}, \dot{s} = \dot{s}_{max}(k+1))$ arrives to $s = L_k$ with a value of $\dot{s}$ smaller than $\dot{s}_{max}(k)$ being therefore unable to intersect the previously computed $\dot{s}(s)$ curve in the region in which it is decreasing, i.e. all the way back to the segment $s \in [0, s_0(N)]$. The symmetric situation is replicated in the backward integration branch that starts at the end of the path. Assuming N large enough, the number of backtracks is $(N - m(N)) - (N - r(N)) \geq N/2 - 80$ i.e. $\mathcal{O}(N)$. The computation required for each backtrack is proportional to the length of the path backtracked.

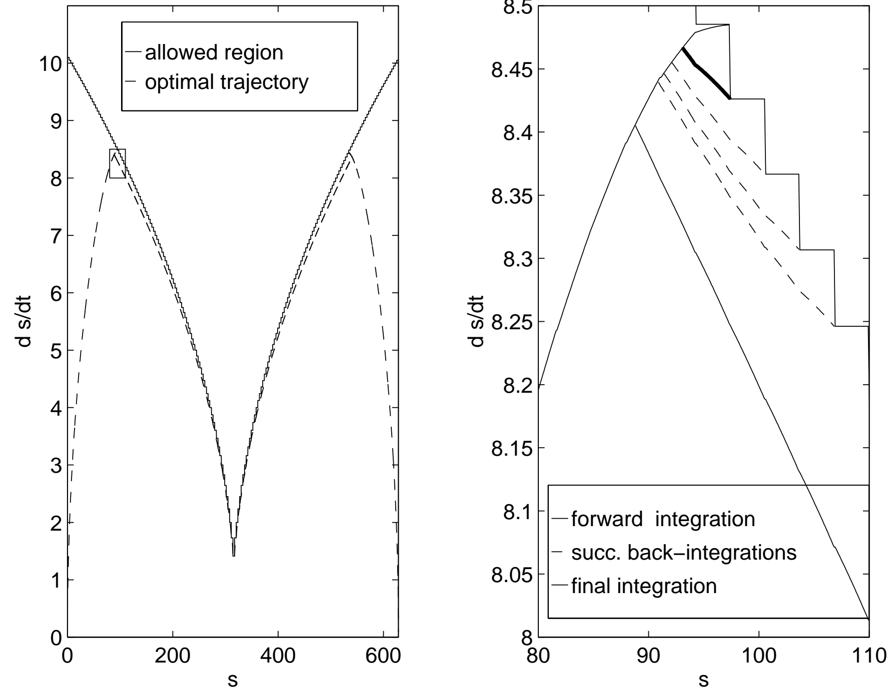


Figure F.2: **Phase space direct integration and detail.**

Since the length of each segment is constant (π), this length is $\mathcal{O}(N)$ and therefore the complexity is $\mathcal{O}(N^2) = \mathcal{O}(L^2)$.

Finally we prove that (as stated above) for $k < r(N) = N - 80$, each backward integration from $(s, \dot{s}) = (L_{k+1}, \dot{s}_{max}(k+1))$ along the maximum deceleration curve $\dot{s} = \dot{s}_{dec}(s)$, reaches $s = L_k$ with a value $\dot{s} < \dot{s}_{max}(k)$.

For $s \in [L_k, L_{k+1}[$ we have $\dot{s}_{dec}(s) \geq \dot{s}_{max}(k+1)$ and therefore:

$$\frac{d\dot{s}_{dec}(s)}{ds} \geq \frac{-1}{\dot{s}_{dec}(s)}\sqrt{1 - \frac{\dot{s}_{dec}(s)^4}{\dot{s}_{max}(k)^4}} \geq \frac{-1}{\dot{s}_{dec}(s)}\sqrt{1 - \frac{\dot{s}_{max}(k+1)^4}{\dot{s}_{max}(k)^4}} = \frac{-Q(N-k)}{\dot{s}_{dec}(s)}$$

$$Q(x) \stackrel{\text{def}}{=} a\sqrt{1 - \frac{(x+1)^2}{(x+2)^2}}$$

Using a bound similar to (F.15) and integrating backwards from L_{k+1}, to L_k we can obtain:

$$\dot{s}_{dec}(s) < \sqrt{2Q(N-k)(L_{k+1} - s) + \dot{s}_{max}(k+1)^2}$$

$$\dot{s}_{dec}(L_k) < \sqrt{2Q(N-k)\pi + \dot{s}_{max}(k+1)^2}$$

$$= \sqrt{2Q(N-k)\pi + a(N-k+1)}$$

Since $Q(x)$ is strictly decreasing and $2Q(80)\pi < a = 0.5$, we have that $N - k > 18 \quad \Rightarrow$ $\dot{s}_{dec}(L_k) < \sqrt{N-k+2} = \dot{s}_{max}(k+1)$ as stated.

Appendix G

Matrix Plotting Utilities

This appendix provides a brief description of the *matrixPlotServer* utility. The matrixPlotServer provides a low-overhead mechanism to visualize ControlShell's matrices (CSMat's). The matrix-PlotServer isn't intended to replace sophisticated matrix plotting tools such as Matlab or Xmath, rather its aim is to allow applications that use matrices to display intermediate and final results easily with minimal impact on the applications performance. The matrixPlotServer can save matrices in Matlab format for later post-processing.

G.1 Overview

The architecture of the matrixPlotServer consists of a small *communications client* linked with the application that sends the matrix information over the network to the *plot server* application. The plot server resides on a Unix workstation, and stores all the matrices sent by the client. Repeated matrices with the same name have a number appended (which represents the order in which they were received), and hence do not replace each other. The plot server allows visualization, zooming and saving operations. Both single and two dimensional matrices can be plotted. Two-dimensional matrices generate one plot per row. A typical session is illustrated in figure G.1. Multiple windows can be open, and any number of matrices can be shown on any window, allowing for easy comparison between matrices.

The matrixPlotServer allows visualization of matrices even if the application is running on platforms with no display (or even X-client) capabilities such as VxWorks. The matrix data is serialized using a machine-independent format (XDR) so that it can be displayed in any machine regardless of its architecture. In addition, since a single unacknowledged packet is sent per plot,

it has relatively small impact on the performance of the application code. This architecture allows the client to execute display functions even if no server is available. This has certain advantages if we want the application to be robust to the presence of the server. On the other hand, it is not appropriate for the collection of sequences of critical data or whenever reliable delivery of each matrix is a concern.

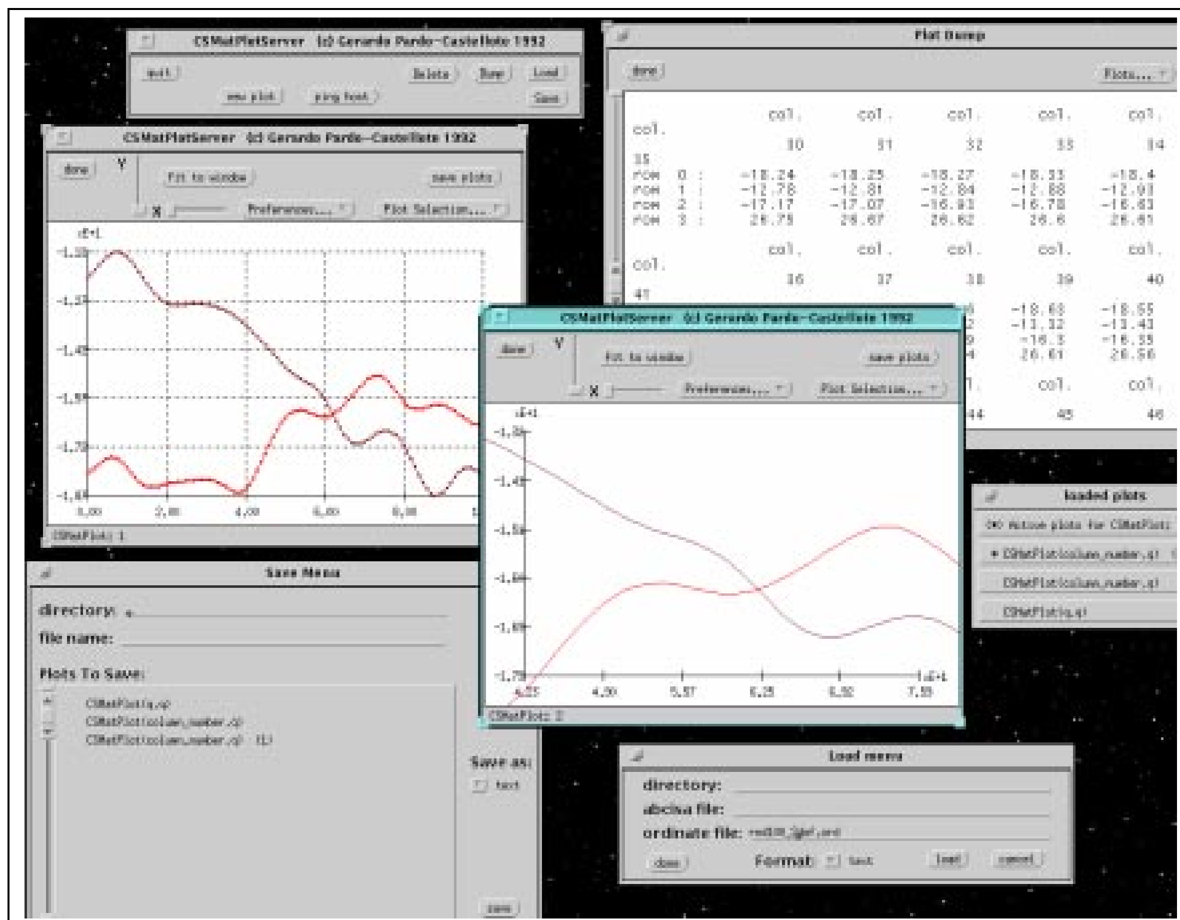


Figure G.1: **A typical session using the** `matrixPlotServer`

The **matrixPlotServer** *allows matrix visualization, zooming and load/store operations on both Control-Shell's and Matlab's formats. The plot window on the top displays a matrix with two rows (hence the two plots). In this plot the user has both selected an grid and requested each individual matrix element to be displayed. The bottom-right plot window illustrates contains is also displaying the same matrix, but the user has zoomed into the area containing the intersection of both curves.*

G.2 Application Interface

The applications interface to the matrixPlotServer consists of three functions: `CSMatPlot`, `CSMatSetVerbosity`, and `CSMatPlotSetDisplayHost`.

CSMatPlot Is the main function that serializes and sends the data to the server. This function is documented in detail in section G.3.

CSMatPlotSetDisplayHost This is a convenience function that allows global setting of the name of the (unix) host where the server will execute.

CSMatSetVerbosity This function allows turning on different levels of warning and status messaging.

G.3 Selected Interface Documentation

This section contains the manual page for the main application interface function.

```
NAME
  CSMatPlot \- Plot matrices

SYNOPSIS
  CSMatPlot(char *displayHost, CSMat m1, CSMat m2)

DESCRIPTION
  Plots m2 versus m1.
  If m1 and m2 are vectors, CSMatPlot will plot each element of m2
  versus each element of m1
  i.e. plots the points    {(m1[i],m2[i]),  i=1..n}

  If m1 is a row vector and m2 is a matrix, then CSMatPlot will
  interpret each row of m2 as a vector and plot it versus m1.
  i.e. for i=1..m plots    {(m1[j],m2[i][j]),  j=1..n}
  If m1 is a column vector and m2 is a matrix, then CSMatPlot will
  interpret each column of m2 as a vector and plot it versus m1.
  i.e. for j=1..m plots   {(m1[i],m2[i][j]),  i=1..n}

  If both m1 and m2 are matrices, then CSMatPlot will interpret
  each row as a vector and plot corresponding vectors in m1 and m2.
  i.e. for i=1..m plots  {(m1[i][j],m2[i][j]), j=1..n}

  COMPATIBILITY:
  m1 and m2 must have "compatible" that is:

  (1) for m1, m2 vectors they have to have the same
      number of elements.

  (2) for m1 vector, m2 matrix, m1 must be a row
```

```
     vector with the same number of columns as m2

 (3) for m1 matrix, m2 matrix, they must have the same size.

 (4) m1 matrix, m2 vector isn't allowed.

 (5) m1==NULL is compatible with any m2. It will plot m2 versus
     column number unless m2 is a row vector in which case it

 DISPLAY:
 displayHost==NULL has the following meaning:
 unix: use the same host where the client is running
 VxWorks: Not allowed!

 PARAMETERS: char *dysplayHost, # where the display is
             CSMat m1,  * Matrix/Vector of abscissas  *
             CSMat m2;  * Matrix/Vector of ordinates *


USAGE
     CSMatPlot(displayHost, m1, m2)
     or
     CSMatPlot(displayHost, NULL, m2)
     or
     CSMatPlot(NULL, m1, m2)
     or
     CSMatPlot(NULL, NULL, m2)

 RETURNS: m1 if no error occurred, NULL if an error
          occurred.

SEE ALSO
    CSMat(2) matrixPlotServer(2)
```

Appendix H

Selected Manual Pages

This appendix lists the manual pages of several of the software packages developed for this project:

The Network Data Delivery Service (NDDS). Included are the main page, some related utilities *(nddsgen, nddsStartDaemon, nddsRequest)*, and the accompanying *Hierarchical Hash Table* package. NDDS is now a commercially available product [145].

Via-Point Trajectory Generator. (VIP). This is a package that implements the proximate-optimal trajectory parameterization algorithm described in chapter 7. This package is now used by several other projects [112, 104, 195]. The VIP package has also been encapsulated into a ControlShell component and is available with the standard release of ControlShell [144].

The matrixPlotServer. This is a graphical utility that allows visualization (plotting, zooming etc.) of matrices (in ControlShell's CSMat format). This utility can operate remotely (over the internet) from the generating side, so that clients without displays or X-windows (such as the real-time systems) can use it to display relevant data. It also allows multiple clients to concurrently send the data to the same display server so that their data can be compared.

In addition to the manual pages presented here, each individual interface function of the above packages has its own manual page which is omitted for brevity.

```
NDDS(2)        NDDS Reference       NDDS(2)



NAME
  NDDS - Network Data Delivery Service.

SYNOPSIS
  NddsInit - Initialize NDDS optionally specify an ndds domain
  NddsPeerHostsSet - Sets hosts that receive subscription requests
  NddsTypeRegister - Register a NDDSObject.
  NddsProducerCreate - Create a new producer. Set its characteristics
  NddsProducerModify - Modify characteristics of exixting producer
  NddsProducerAddProduction - Add instance to list of productions
  NddsProducerSample - Sample all productions of the producer
  NddsConsumerCreate - Create a new consumer
  NddsConsumerModify - Modify the characteristics of existing consumer
  NddsConsumerAddSubscription - Consumer requests subscription.
  NddsConsumerPoll - Receive the updates of a specific consumer
  NddsInstanceQuery - Distributed query to all NDDS nodes
  NddsConsumerGetByName - Get a consumer from its name
  NddsProducerGetByName - Get a producer from its name
  NddsUtilityFind - Get an object given its type and name
  NddsUtilityStore - Enter a record of a given type.
  NddsDBaseListInstances - prints all NDDSObjectInstances known to NDDS
  NddsDBaseListConsumers - Print information on matching consumers
  NddsDBaseListProducers - Print information on matching producers
  NddsDBaseListProductionsRemote - Print info on matching remote productions
  NddsDBaseListSubscriptionsRemote - List matching subscription requests
  NddsDBaseListAll - Hierarchical printing of all NDDS databeses
  NddsUtilitySleep - Sleep for a specify number of micro-seconds
  NddsUtilityTimeGet - Get current time in seconds.
  NddsVerbositySet -  Sets/Gets verbosity level.
  NddsProductionsReliableParamSet - Set/Get parameters for reliable updates
  NddsProductionsAsyncParamSet - Set/Get parameters for async. productions

DESCRIPTION
  The Network Data Delivery Service (NDDS) provides transparent network  con-
  nectivity  and data ubiquity to a set of processes possibly running on dif-
  ferent machines. NDDS allows distributed processes to share data and event
  information  without concern for the actual physical location and architec-
```

ture of their peers. NDDS allows its "clients" to share data in a dual mode: subscriptions and one-time queries. NDDS especially targets applications with repetitive data flow, where low latency and high robustness to network changes and failures are required. Distributed control systems are one common example of such systems.

NDDS supports "subscriptions" as the fundamental means of communication. In the application context described, subscriptions have some fundamental advantages over other information sharing models (such as client-server or shared-memory). It cuts in half the data latency over query/response type models and it allows synchronization on the latest available information as soon as it is produced.

NDDS clients register any number of "producers" and "consumers" of named data objects. A single NDDS client may register multiple producers and consumers. Each individual producer and consumer has its own characteristics (described later) and is a producer/consumer to a set of named data objects. A NDDS client registers notify functions on each subscription. These functions are called back whenever a new update arrives. The rate at which data updates arrive and the notify function is called depends on both the rate at which data is produced and the consumer characteristics. See the man pages on NddsConsumerCreate and NddsProducerCreate.

Consumer notification can be initiated by the NDDS client---this amounts to polling to see if there are updates for the consumer's data (in this case the notify functions are called in the task context of the NDDS client) or can be asynchronously triggered by the arrival of the new data itself.

Ndds Objects: All data communicated in NDDS must be declared as an instance of an NDDS object. The user must define any object types it wishes to use by providing the methods required to perform basic operations on them. NDDS provides a basic library of objects (see the man page on NDDSobjects). The user can use this objects as models and define others that suit the application. All objects are defined using NddsTypeRegister().

Data Subscriptions: To subscribe to data you must first register yourself as a consumer. This is achieved with NddsConsumerCreate(). A consumer specifies some parameters that control things like the maximum data update rate and whether the notify routines are initiated by the user or called asynchronously on new data updates. See the man page on NddsConsumerCreate

for further information. A registered consumer can subscribe to named data objects using NddsConsumerAddSubscription(). See the man pages on NddsConsumerAddSubscription() and NddsConsumerPoll().

Data Production: To produce data you must first declare yourself as a producer. This is achieved using NddsProducerCreate(). A producer must specify some parameters that indicate the type of producer and help conflict resolution among multiple producers. Once registered, a producer can specify any number of NDDS objects it wishes to produce. This is achieved with NddsProducerAddProduction(). All data produced by the same producer is sampled synchronously when the user calls NddsProducerSample(). This will take snapshots of all the data which will be held and later distributed to all consumers.

NDDS supports reliable updates. Any producer can specify any of its updates to be delivered reliably. Reliable updates are guaranteed to be delivered in order.

NDDS maintains a database of all the instances of different objects that have been registered. You can access the database using NddsUtilityFind() and NddsUtilityStore().

BUILDING an APPLICATION: In your code include "NDDS.h". In Unix link to libNDDS.a, NddsXDR.a, libHash.a and netio.a (in that order) libNDDSobjects.a and netio.a (in that order)

In VxWorks link or preload netio.so libHash.so NddsXDR.so libNDDS.so in that order.

RUNNING AN APPLICATION

Lets assume that you have built 2 applications "producer" and "consumer" that communicate between hosts "mars" and "pluto". Here is a check list of things that will ensure proper operation:

1.- Make sure the there are ndds daemons is running on both "mars" and "pluto" on the same domain-number (say 8000) that you used in your NddsInit() call within your applications. Typically this number will either be defaulted, read from the command line by your program or taken from an environment variable.

```
  You can verify proper operation of the NDDS daemons by issuing the command:
    nddsRequest -c PING -n mars -d 8000

  If the daemons aren't running, you can start them with the command:
    nddsStartDaemon -d 8000 -v 1

  both on "mars" and "pluto"

  2.- Make sure that your environment variable NDDS_PEER_HOSTS includes  both
  mars and pluto (and any other hosts you want) for example NDDS_PEER_HOSTS =
  mars:pluto:saturn

  3.- Start the program on "mars" with the same domain-port number:
    producer 8000

  4.- Start the companion program on "pluto" (same domain-port number):
    consumer 8000

  5.- Depending on the verbosity level selected <verbosity> = 0, 1 or  2  You
  should  see messages flowing back and forth between the two programs. Refer
  to the source in the examples  directory  of  the  NDDS  distribution  for
  further details.


EXAMPLES
  Example source code using NDDS and makefiles to build it can be  found  in
  the examples directory of the NDDS distribution.


SEE ALSO
  nddsStartDaemon, nddsRequest, NddsXDR, nddsgen, PhoneMessage, and the
     individual man pages on each function.
```

```
nddsgen(1)     NDDS    nddsgen(1)



NAME
  nddsgen - Automatically generate code to describe a type to NDDS

SYNOPSIS

  No user-callable interface

USAGE
  nddsgen <filename> [-typename <name>] [-nddstype <type>]
  [-example] [-remove] [-architecture <arch>]

  <filename> -- file containing XDR description of data-type

  <name>     -- name of the type defined in <filename>,
defaults to <filename> without any extensions

  <type>     -- name of the NDDS data type that will be
generated. Defaults to p_<name>.  Cannot be
the same as the XDR type name.

  -example   -- Generate example programs that use NDDS to
communicate

  -remove    -- Do not generate 'stub' files, instead write
the actual files directly.
Warning. This will remove any changes you have
made to the files.

  <arch>     -- Architecture for the example makefile,
defaults to sun4



DESCRIPTION
  nddsgen takes a high-level specification of the  data  format  in  the  XDR
  language  and  generates  code to serialize and deserialize objects of that
  type so that they can be distributed using NDDS.
```

nddsgen can also create stub test programs and a makefile so to get a simple working NDDS application, the user only needs to fill in some initialization code (i.e., give initial values to the data) and type "make".

To use nddsgen you need to write a description of the type in the XDR language. XDR stands for eXternal Data Representation and is the industry standard machine-independent data-representation format to exchange data across machines.

Assuming you have a description for a data of type "myDataType" in the file "myFileName". All you need to do is type:

```
nddsgen  myFileName -t myDataType -example
```

nddsgen will echo several messages as it progresses and, in the end you will be left with all the necessary stub files and a makefile!

You will need to edit some files (refer to the messages printed by nddsgen) in order to give your data initial values etc. after this you can type "make -f <makefile-name>" and you'll have a working NDDS application!

THE XDR LANGUAGE

In the XDR language, data-types are described in a fashion very similar to structures in "C". Even if you are not familiar with it, looking at the following examples and more that can be found under the objectlib subdirectory on your NDDS distribution will probably be enough.

The complete description of the language can be found in "The External Data Representation Standard: Protocol Specification" chapter in the "Network Programming Manual", available from most Unix vendors.

EXAMPLES

In this example a FloatArray (named variable-sized array of floats) will be constructed.

1.- Create a file called FloatArray.xdr that contains the description in the XDR language. In this case, the complete file contains:

```
struct FloatArrayStruct {
```

```
     string name<>; / * a NULL terminated string * /
     float data<>;  / * a variable-size array * /
   };

  2.- Next run nddsgen with the command:
nddsgen FloatArray.xdr -t FloatArrayStruct
   -n FloatArray -ex

  3.- If this is the first time you tried to create the FloatArray type,  the
  files    FloatArray.c,    FloatArray_test.c  FloatArray_producer.c   and
  FloatArray_consumer.c will have been created for you. Otherwise,  you  will
  need to incorporate the changes from the corresponding stub files.

  Edit these files to initialize your data and provide a custom print routine
  for the data. Failure to initialize the data may cause the example programs
  to crash.

  4.- Now type:
 make -f makefile_FloatArray_sun4

  You  will  end up    with    3    programs: sun4/FloatArray_test,
  sun4/FloatArray_producer, and sun4/FloatArray_consumer.

  5.- First run the  FloatArray_test  to  make sure  the  serialization  and
  deserialization  routines  operate  properly.  If  you've provided a custom
  print    routine,    the    data    will   be    printed before   and after
  serialization/deserialization to show that data integrity is maintained.

  6.- Start the NDDS daemon on each machine that you want  to  run  the  NDDS
  tests  by  using nddsStartDaemon (see man nddsStartDaemon for more details)
  If two machines are named, say, "mars"  and  "pluto",  type  both  on  both
  machines:
    nddsStartDaemon -v 1

  7.- Make sure your NDDS_PEER_HOSTS environment variable is defined and con-
  tains  the  colon-separated  list of machines that will be running the NDDS
  tests. In this case, on both "mars" and "pluto", type:
     setenv NDDS_PEER_HOSTS mars:pluto

  8.- Run the tests:
    On "mars" type: sun4/FloatArray_producer
```

```
    On "pluto" type: sun4/FloatArray_consumer

  You should see both programs printing information as they communicate.
```

SEE ALSO

NDDS, NddsXDR, PhoneMessage, FloatArrayPtr

```
nddsStartDaemon(1)        NDDS Reference    nddsStartDaemon(1)



NAME
  nddsStartDaemon  - Start the NDDS daemon

SYNOPSIS

  No user-callable interface

DESCRIPTION
  Use this command to start the NDDS daemon on the local host. For  communi-
  cations to take place through NDDS, one NDDS Daemon must exist on each par-
  ticipating host in the ndds-domain used for the communications. This ndds-
  domain  can be any legal port number in your system greater than 7400. This
  must be the *same* numbers specified by the application in  the  NddsInit()
  calls. If an illegal domain-number is passed (or it is simply unspecified),
  a default value will be used. Thus starting the NDDS daemons without speci-
  fying  the domain-number and calling NddsInit(0) from the applications will
  yield the same domain for the daemon and applications.


USAGE
  On Unix systems type to the shell:
   nddsStartDaemon [-d <domain number>] [-v <verbosity>] \
   [-c <license file>] [-h]

  -d <domain number> domain number of the daemon. See man page on  NddsInit.
  The default domain number is used if left unspecified.

  -v <verbosity> select verbosity level 0 (silent), 1 (status), 2 (debug)

  -c <license file> path to the RTI license file  for  this  application.  If
  omitted, the standard places will be searched (see man page on lmcontrol)

  -h print brief help message.

  In VxWorks, type to the VxWorks shell (after loading the NDDS libraries)

    nddsStartDaemon 8000, 1, \
```

```
      "<path_to_the_license_file>/license.dat"
```

EXAMPLE

```
  UNIX>  nddsStartDaemon -d 8000  -v 1 \
 -c $(RTIHOME)/lm/license.dat
```

Will start the NDDS daemon in the local host in the domain-number 8000 with verbosity 1 (the default) using the key in the license file $(RTIHOME)/lm/license.dat

```
  VxWorks> nddsStartDaemon 8000, 1, \
   "/local/rti/lm/license.dat"
```

Aside from any message that it prints (depending on the verbosity level), you can check that the daemon is running by typing:

```
  UNIX> nddsRequest -c PING -d 8000
```

SEE ALSO

nddsRequest, NddsInit, NDDS.

```
NddsXDR(2)     NDDS    NddsXDR(2)



NAME
  NddsXDR - Library of XDR utilities for NDDS

SYNOPSIS
  NddsXDRGetSerializingStream - access the raw XDR serializing stream
  NddsXDRGetDeserializingStream - access the raw XDR deserializing
  NddsXDRStreamAlloc - Allocate an NDDS data stream
  NddsXDRStreamFree - Free an NDDS data stream
  NddsXDRSerializeInteger - Serialize an integer from the NDDSXDRStream
  NddsXDRDeserializeInteger - Deserialize an integer from the NDDSXDRStream
  NddsXDRSerializeArrayInt - Serialize array into the NDDSXDRStream
  NddsXDRDeserializeArrayInt - Deserialize a int array
  NddsXDRSerializeFloat - Serialize a float into the NDDSXDRStream
  NddsXDRDeserializeFloat - Deserialize a float from the NDDSXDRStream
  NddsXDRSerializeArrayFloat - Serialize a float array into the NDDSXDRStream
  NddsXDRDeserializeArrayFloat - Deserialize a float array
  NddsXDRSerializeDouble - Serialize an double from the NDDSXDRStream
  NddsXDRDeserializeDouble - Deserialize an double from the NDDSXDRStream
  NddsXDRSerializeString - Serialize a string into the NDDSXDRStream
  NddsXDRDeserializeString - Deserialize a string from the NDDSXDRStream


DESCRIPTION
  This library provides network utilities to  handle  serialization  and  de-
  serialization through XDR.

  To register a type with NDDS,  serialization and  deserialization  methods
  must be  provided.  These  methods  are  used  to serialize/deserialize an
  NDDSObject into/from an NDDSXDRStream.

  The serialization/deserialization methods can be *automatically*  generated
  from a high-level specification of the data-type using "nddsgen", a utility
  that comes with your NDDS distribution. With "nddsgen" you can generate the
  serialization/deserialization methods *without* any programming!

  To use "nddsgen" you must describe the data types using the XDR language.
```

You don't need to become an expert on the XDR language, or read any documentation to start using it! Looking at a couple of examples and playing with "nddsgen" is probably all most people will ever need to do. Complete examples on using "nddsgen" to generate the serialization/deserialization methods are available both in the examples and nddstypes subdirectories of your NDDS distribution. See the man pages on "FloatArray" and "PhoneMessage" for two examples that use "nddsgen".

If you want to know more about it, XDR stands for eXternal Data Representation and is the industry standard machine-independent data-representation format for data exchange across machines. For further information see the man pages on "nddsgen". A complete description of the XDR language can be found in: "The External Data Representation Standard: Protocol Specification" chapter in the "Network Programming Manual". This is available from Sun Microsystems (the original developer of the XDR language) and virtually every UNIX vendor.

There are many data types that are so simple that it may be easier to write the serialization/deserialization routines by hand. For these cases, NddsXDR provides utility functions that can make writing these extremely simple. An example can be found in the nddstypes directory of the NDDS distribution. See the man page on "IntArray".

Yet in other cases there are data structures whose semantics are very difficult to express syntactically in the XDR language (or any other syntactic language for that matter).

For example a structure such as:

```
struct StructWithSemanticInfo {
  int numchar; / * number of characters in "data" * /
  char *data; / * buffer of size "numchar" * /
  char *begin; / * begin = &data[0] * /
  char *end; / * end   = &data[numchar] * /
};
```

In this cases the best (only) approach is to write the code by hand using the utilities provided by NddsXDR.

The functions provided by NddsXDR will be sufficient to handle most data types you may come across. In the few cases in which it may be necessary to

```
  use XDR routines directly, the XDR stream can be accessed directly with the
  functions:

    NddsXDRGetSerializingStream() if serialization is intended
    NddsXDRGetDeserializingStream() if deserialization is
intended.

  See the examples below.


EXAMPLES
  To serialize the following structure:

  include "NddsXDR.h"

  struct PolygonStruct {
      char *name;    / * name of the polygon * /
      int nVertex;   / * number of vertices * /
      float *x;      / * x-coordinates of the vertices * /
      float *y;      / * y-coordinates of the vertices * /
   } *Polygon;

  (a) Using NddsXDR functions:
      -----------------------

   include NddsXDR.h

   boolean SerializePolygon(NDDSXDRStream nddsds,
    Polygon polygon)
   {
      int MAX_LEN = 10000;
      if (   (!NddsXDRSerializeString(nddsds, polygon->name,
      MAX_LEN))
  || (!NddsXDRSerializeInteger(nddsds,
       polygon->nVertex))
  || (!NddsXDRSerializeArrayFloat(nddsds, polygon->x,
  polygon->nVertex))
  || (!NddsXDRSerializeArrayFloat(nddsds, polygon->y,
  polygon->nVertex)))
       {
   return FALSE;
```

```
    }

    return TRUE;
 }
```

```
(b) Using XDR functions directly:
    ----------------------------

 include <rpc/types.h>
 include <rpc/xdr.h>

 boolean SerializePolygon(NDDSXDRStream nddsds,
  Polygon polygon)
 {
     int i, MAX_LEN = 10000;
     XDR *xdrs;

     xdrs = NddsXDRGetSerializingStream(nddsds);

     if (    (!xdr_string(xdrs, &polygon->name, MAX_LEN))
 || (!xdr_int(nddsds, &polygon->nVertex))) {
 return FALSE;
     }

     for (i=0; i<polygon->nVertex; i++) {
 if (!xdr_float(nddsds, &polygon->x[i])) {
     return FALSE;

 }
     }

     for (i=0; i<polygon->nVertex; i++) {
 if (!xdr_float(nddsds, &polygon->y[i])) {
     return FALSE;
 }
     }

     return TRUE;
 }
```

SEE ALSO

NddsTypeRegister, nddsgen, NDDS, IntArray, FloatArray, rpc, xdr, and the individual man pages on each function.

HT(2) NDDS HT(2)

NAME

HT - a hash table library

SYNOPSIS

HHTSetVerbosity - set verbosity level
HHTFindHT - Find a Hash Table Given a list of keys.
HHTFindEntry - Find the entry handle for the record
HHTFindRecord - Find a record under a list of keys
HHTEnterRecord - Enter a record (va_arg version)
HHTCreate - Create a hash table
HHTForAllMatches - Call a user-specified routines for each match
HHTListRecords - List hash tables hierarchically
HHTPrune - Recursively remove specific records and hash tables
HHTRemoveRecord - Remove a record
HHTDestroy - Destroy a hash table and all its descendants.
HTSetVerbosity - set verbosity level
HTFindEntry - Finds the entry given the key.
HTGetRecord - Gets the record given the entry
HTGetRecordUserHandle - Gets the user handle given the entry
HTSetRecordUserHandle - Sets the user handle given the entry
HTGetRecordType - Gets the record type given the entry
HTFindRecord - Find the record given the key.
HTEnterRecord - Enter a record (removes duplicates)
HTRemoveRecord - Remove a record
HTCreate - Creates a hash table
HTSetAccessType - Sets (concurrent) access type for HashTable
HTFree - Free memory associated with a hash table
HTGetHashTableUserHandle - Gets the user handle of a HashTable
HTForAllMatches - Call a user-specified routine for each match
HTForAll - Call a user-specified routine for every record
HTPrune - Remove specific records
HTDestroy - Destroy a hash table and remove all its records.
HTIsEmpty - Return TRUE if the hash table is empty

DESCRIPTION

This library implements a Hierarchical Hash Table (HHT) Abstract Data Type. A HHT is a hash table whose entries can themselves be hash tables. Charac-

ter strings are used as keys.

This package provides an efficient way of organizing data in a hierarchy. It also allows searching a HHT (and all its descendants) and executing user-provided callback routines for all records that match given templates.

Along with each record, a user-provided handle and integer type description are stored. These supply the basic functionality to build more sophisticated layers on top of this library.

The implementation gives a lot of power to the user. Beware; power carries its dangers.

SEE ALSO

Individual man pages on each function.

vip(2) CS Component vip(2)

NAME

vip.c - Via Point Trajectory Generator

SYNOPSIS

VIP_SetVerbosity - Set/Get verbosity level:
VIP_SetUserParameter - Bind a user parameter to a VIP trajectory
VIP_GetUserParameter - Bind a user parameter to a VIP trajectory
VIP_GetTrajectoryByName - Get the VipTrajectory handle from the name
VIP_SetStrictBoundEnforcement - Enforce Bounds
VIP_GetStartAndEndingTime - Get the time the trajectory will end
VIP_GetViaPoints - Get the via times of a trajectory
VIP_GetViaTimes - Get the via times of a trajectory
VIP_SetSmoothingKernel - Provide your own smoothing kernel
VIP_GetSmoothingKernel - Get the current Smoothing Kernel
VIP_Create - Create a trajectory
VIP_SetBeginNotifyFunc - Sets a notify function to call when traj. begins.
VIP_SetEndNotifyFunc - Sets a notify function to call when traj. ends
VIP_Destroy - Destroy a trejectory (Free memory)
VIP_Cancel - Cancel a trajectory
VIP_ReTimeParameterize - patch an existing trajectory
VIP_TimeParameterize - time parameterize a set of via points
VIP_GetTrajectorySample - Get the desired state at any time
VIP_SetViaPts - Sets the via points of a trajectory.
VIP_SetStartTime - Set the starting time of a trajectory
VIP_EvaluateTrajectory - Evaluate trajectory at a given time
VIP_ReallocateTrajectoryState - Reallocate space for a traj state.
VIP_AllocateTrajectoryStateHistory - Allocate space for a state history
VIP_FreeTrajectoryStateHistory - Free memory in state history
VIP_GetDesiredStateHistory - Get the trajectory's desired state history
VIP_PrintTrajectoryStateHistory - Print the state history
VIP_PlotTrajectoryStateHistory - Plot the state history
VIP_AllocateDynamicInfo - Allocate memory to Hold dynamic information
VIP_FreeDynamicInfo - Free memory used to hold dynamic information
VIP_ApproxAccelLimits - Set/Reset usage of approximate acceleration limits
VIP_SetAccelerationLimits - Sets the acceleration limits
VIP_SetVelocityLimits - Sets the velocity limits
VIP_SetTorqueLimits - Sets the torque limits

VIP_SetRobotDynamics - Specify robot dynamics

DESCRIPTION

Name: VIP - Via Point Trajectory Generation

Usage: include <vip.h> link to the following libraries: Unix: vip.a matrix.a matrix_plot.a netutils.a netio.a VxWorks : vip.so matrix.ro matrix_plot.ro netutils.so netio.so

This library implements the trajectory generation algorithm described in ARL memo: ARL-XX-92. In a nutshell, given a set of via points in any number of degress of freedom and dynamic constraints on the manipulator/robot capabilities, this algorithm will generate a trajectory that will traverse the set of via points in minimum time subject to the constraints imposed. The contraints can by any combination of (configuration independent) velocity, acceleration and torque limits. The algorithm generates a trajectory in linear time with respect to both the number of via points and the number of degreess of freedom.

Using this package requires 3 phases:

(a) Creating a trajectory ADT and setting up constraints. See VIP_Create() and VIP_AllocateDynamicInfo() VIP_SetVelocityLimits() VIP_SetAccelerationLimits() VIP_SetTorqueLimits().

(b) Time parameterizing ta set of via points. See VIP_TimeParameterize(), VIP_ReTimeParameterize()

(c) Splining the via points to a set of via times. See VIP_SetViaPoints().

(c) Getting desireds in the control loop. See VIP_GetTrajectorySample().

There is also a controlshell component that will do most of that work for you. See man VIP_GenarateState

CAVEATS

Generating trajectories in a control loop is risky business, follow this directions with care. This comments apply to a specific instance of a VIP trajectory (As returned by VIP_Create()).

```
(1) The process running VIP_GetTrajectorySample() must run at higuer prior-
ity  than  VIP_SetViaPts()  VIP_SetStartTime().  So that it is never inter-
rupted by those.

(2) It must always be  O.K.  to  zero  order hold  the  state,  passed  to
VIP_GetTrajectorySample().  In  other  words,  you  must provide reasonable
values for the parameters. The VIP_GetTrajectorySample()

Language: Standard C

Written by: Gerardo Pardo-Castellote (gpc), Stanford University, May 1991
Revised:
   May 1994, gpc  Added support for patching on-going trajectries

SEE ALSO:
   VIP_GenarateState
```

VIP_GenerateState(5) CS Component VIP_GenerateState(5)

NAME

VIP_GenerateState - Component interface to VIP Trajectory generator

SYNOPSIS

VIP_GenerateStateClass::ResetTraj - Reset trajectory
VIP_GenerateStateClass::vip - Get VIP trajectory object
VIP_GenerateStateClass::dInfo - Get VIP trajectory dynamics object

DESCRIPTION

Use this component to produce trajectories generated by the Via-Point (VIP) trajectory generator. You make regular calls to the VIP libraries to convert a set of via points into a smooth, time-parameterized set of trajectories.

You supply this component with the dynamics of your system and any limits you would like to place on the trajectory, and the component will create the DynamicInformation and VipTrajectory objects for you at instance time. You can use the vip() and dInfo() public methods to rertrieve these pointers at run time in order call the VIP libraries to calculate actual trajectories.

This component generates an output called "state" at each sample time of the position, velocity, and acceleration of each degree of freedom (DOF) in your system. Each row of "state" corresponds to a single DOF, and the columns are: 0 = position, 1 = velocity, 2 = acceleration.

The parameter "limitMask" is used to indicate which limits you would like to impose on the trajectory generation. To impose more than one type, add the values together.

```
VIP_VEL_LIMIT    = 0x1
VIP_ACC_LIMIT    = 0x2
VIP_TORQUE_LIMIT = 0x4
```

The parameters "maxVel", "maxAccel", and "maxTorque" should be matrices that contain the limits for each DOF. The limits are symmetric about zero, such thai MIN = -MAX. If the "limitMask" does not select a particular

limit, that CSMat may be a dummy matrix.

The parameter "initialState" is a matrix that contains a position that the trajectory should match when the component is enabled or when you call the ResetTraj() method. Use this to match desired and actual positions.

If you do not impose torque limits, you may ignore the "eom" parameters.

The parameter "eomParam" should contain the dynamic parameters for your system. The format of the data is completely determined by how they are used in you dynamic equations of motion. "eomParam" is passed to the "eomRtnCreate" routine to create a data object that can be utilized by the "eomRtnName_M", "eomRtnName_B", "eomRtnName_C", and "eomRtnName_G" routines.

These "eomRtnName" funtions combine to form the equatons of motion for your dynamic system. They are defined by you in your user source code. At instance time, the VIP_GenerateState component will use these String parameters to find the routines in the symbol table. If they are not found, it will generate an error.

IMPORTANT

Use extern "C" {} to enclose all the EOM functions to prevent the C++ compiler from "mangling" the function names. Otherwise, This component will not be able to find the functions at instance time.

DFE ENTRY FORMAT

```
VIP_GenerateState: <InstanceName> <HabitatName>
    state                    <CSMat_output>
    limitMask                <int_reference>
    maxVel                   <CSMat_reference>
    maxAccel                 <CSMat_reference>
    maxTorque                <CSMat_reference>
    initialState             <CSMat_reference>
    eomRtnCreate             "<String_reference>"
    eomParam                 <CSMat_reference>
    eomRtnName_M             "<String_reference>"
    eomRtnName_B             "<String_reference>"
    eomRtnName_C             "<String_reference>"
```

```
     eomRtnName_G              "<String_reference>"
```

SEE ALSO

VIP(2)

```
matrixPlotServer(2)     Reference Manual      matrixPlotServer(2)

NAME
     matrixPlotServer - CSMat plotting facility.

SYNOPSIS
     CSMatPlotSetVerbosity - Set vervosity level
     CSMatPlotSetDisplayHost - set default display host
     CSMatPlot - Plot matrices

     int CSMatPlotSetVerbosity(int verbosity)
     void CSMatPlotSetDisplayHost(char *displayHost)
     CSMat CSMatPlot(char *displayHost, CSMat m1, CSMat m2)

DESCRIPTION
     matrixPlotServer is a CSMat plotting facility based  on  the
     client-server paradigm. Any "client" program may request for
     a  matrix  to  be  plotted  in   a   in   any   host   using
     CSMatPlot(<host-name>,  mat1,  mat2) If a "matrixPlotServer"
     is running on host <host-name> it will receive the plot which
     can  then be displayed, printed, saved in matlab format etc.
     If the plot server isn't running, there are no  other  side-
     effects.

     In any case the client proceeds without interruption,  there
     is  no waiting for confirmation or checking for success. The
     protocol is unreliable and it is possible the plot will  not
     be received.

  Written by: Gerardo Pardo-Castellote (gpc), Stanford University, May 1991

  SEE ALSO:
     CSMat
```

Bibliography

[1] A. Hörmann and U. Rembold. Development of an Advanced Robot for Autonomous Assembly. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 2452–57, Sacramento, CA, May 1991.

[2] Bruno Achauer. The DOWL Distributed Object-Oriented Language. *Communications of the ACM*, 39(9):48–55, September 1993.

[3] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *IEEE Computer*, pages 26–34, August 1986.

[4] J. S. Albus, H. G. McCain, and R. Lumia. NASA/NBS Standard Reference Model for Telerobot Control System Architecture (NASREM). Technical Report 1235, NIST, April 1989.

[5] James S. Albus. System Description and Design Architecture for Multiple Autonomous Undersea Vehicles. Technical Report 1251, NIST, ARPA, September 1988.

[6] James S. Albus. Outline for a Theory of Intelligence. *IEEE Transactions on Systems, Man and Cybernetics*, 21(3):473–509, May/June 1991.

[7] J.S. Albus. RCS: a reference model architecture for intelligent control. *IEEE Computer*, 25(5):56–9, May 1992.

[8] Peter K. Allen, Aleksandar Timcenko, Billibon Yoshimi, and Paul Michelman. A least-commitment approach to intelligent robotic assembly. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1850–1856, Nice, France, May 1992.

[9] G.T. Almen, A.P. Black, E.D. Lazowska, and J.D. Noe. The Eden System: A Technical Review. *IEEE Transactions on Software Engineering*, 11(1):43–67, 1985.

[10] A.L. Ananda and B.H. Tay. A Syrvey of Asynchronous Remote Procedure Calls. *ACM Operating Systems Review*, 26(2):92–109, April 1992.

[11] R. Ananthanarayanan, S. Menon, A. Mohindra, and U. Ramachandran. Experiences in Integrating Distributed Shared Memory with Virtual Memory Management. *Operating Systems Review (ACM)*, 26(3):4–26, July 1992.

[12] R. L. Andersson. *A Robot Ping-Pong Player: Experiment in Real-time Intelligent Control*. MIT Press, Cambridge, Mass., 1988.

[13] R. C. Arkin. The Impact of Cybernetics on the Design of a Mobile Robot System: A Case Study. *IEEE Transactions on Systems, Man and Cybernetics*, 20(6):1245–1257, November 1990.

[14] Henri E. Bal, Jenniffer G. Steiner, and Andrew S. Tanenbaum. Programming Languages for Distributed Computer Systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[15] J. C. Berets, N. Cherniack, and R.M. Sands. Introduction to Cronus. Technical Report 6986, BBN Systems and Technologies, 10 Moulton Street, Cambridge MA 02138, January 1993.

[16] C. Bien. Simulation a necessity in safety engineering. *Robotics World*, 10(4):22–27, Dec. 1992.

[17] A.D. Birrell and B.J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2:39–59, February 1984.

[18] John W. Blake. *PHIGS and PHIGS PLUS*. Academic Press, 1993.

[19] Maarten Boasson. Control Systems Software. *IEEE Transactions on Automatic Control*, AC-38(7):1094–1106, July 1993.

[20] J.E. Bobrow. Optimal robot path planning using the minimum-time criterion. *IEEE Journal of Robotics and Automation*, 4(4):443–451, August 1988.

[21] R. Peter Bonasso and Marc G. Slack. Ideas on a System Design for End-User Robots. In *Cooperative Intelligent Robotics in Space III*, volume 1829, pages 352–358. SPIE, 1992.

[22] S. Bonner and K.G. Shin. A comparative study of robot languages. *IEEE Transactions on Computers*, (31):82–96, 1982.

[23] Frederick Phillip Brooks. *The Mythical man-month*. Addison-Wesley Pub. Co, Reading, Mass., 1975.

[24] Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2(1):14–23, March 1986.

[25] J. Butler and M. Tomizuka. A suboptimal reference generation technique for robotic manipulators following specified paths. *Transactions of the ASME, Journalof Dynamic Systems, Measurement, and Control*, 114:524–527, September 1992.

[26] Philip L. Butler. An Integrated Architecture for Modular Control Systems. *Robotics and Autonomous Systems*, 10(2-3):101–114, 1992.

[27] Giorgio C. Buttazzo, Benedetto Allotta, and Felice P. Fanizza. Mousebuster: A Robot System for Catching Fast Moving Objects by Vision. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 932–937, Atlanta, GA, May 1993.

[28] David J. Cannon and Larry J. Leifer. Point-and-direct robotics. In *International Conference on Intelligent Teleoperation*, Greensboro, NC, November 1991.

[29] Vincent W. Chen. *Experiments in Adaptive Control of Multiple Cooperating Manipulators on a Free-Flying Space Robot*. PhD thesis, Stanford University, Stanford, CA 94305, December 1992. Also published as SUDAAR 631.

[30] Y. Chen, S. Y.-P Chien, and A. A. Desrochers. General structure of time-optimal control of robotic manipulators moving along prescribed paths. *International Journal of Control*, 56(4):767–782, 1992.

[31] D.R. Cheriton. The V Kernel: A Software Base for Distributed Systems. *IEEE Software*, pages 19–43, January 1984.

[32] R. S. Chin and S. T. Chanson. Distributed Object-Based Programming Systems. *ACM Computing Surveys*, 23(1):91–123, March 1991.

[33] D. Clark. The structuring of systems using upcalls. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 171–180, Orcas Island, Washington, December 1985. ACM SIGOPS.

[34] Douglas E. Comer. *Internetworkning with TCP/IP*, volume 1-3. Prentice Hall, Englewood Cliffs, N.J., 2 edition, 1991-92.

[35] Eve Coste-Maniere, B. Espiau, and E. Rutten. A task-Level Robot Programming Language and its Reactive Execution. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 2751–56, Nice, France, May 1992.

[36] Eve Coste-Maniere, B. Espiau, and E. Rutten. A task-Level Robot Programming Language and its Reactive Execution. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 2732–37, Nice, France, May 1992.

[37] J.J. Craig. Robot calibration facilitates off-line programmin. *Robotics World*, 10(1):24–25, March 1992.

[38] John J. Craig. *Introduction to Robotics Mechanics and Control*. Addison-Wesley, Reading, MA, 1986.

[39] J. L. Crowley and Y. Deazeau. Principles and techniques for sensor data fusion. *Signal Processing*, 32(1-2):5–27, May 1993.

[40] P. Dasgupta, R. Ananthanarayanan ans S. Menon, A. Mohindra, and R. Chen. Distributed Programming with Objects and Threads in the CLOUDS System. *Computing Systems*, 4(3):243–275, Summer 1991.

[41] J. Denavit and R.S. Hartenberg. A kinematic notation for lower-pair mechanisms based on matrices. *Journal of Applied Mechanics*, June 1955.

[42] William C. Dickson. *Experiments in Cooperative Manipulation of Objects by Free-Flying Robot Teams*. PhD thesis, Stanford University, Department of Aeronautics and Astronautics, Stanford, CA 94305, December 1993. Also published as SUDAAR 643.

[43] H. F. Durrant-Whyte, B.Y.S. Rao, and H. Hu. Toward a Fully Decentralized Architecture for Multi-Sensor Data Fusion. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1331–1336, Cincinnati, Ohio, May 13-18 1990.

[44] Tim O'Reilly (editor). *X Toolkit Intrinsics Reference Manual*, volume 5 of *The X window system series*. O'Reilly & Associates, 1990.

[45] B. Eisenberg, B. Hine, and D. Rasmussen. Telerobotic vehicle control: Nasa preps for mars. *AI Expert*, 8(8):18–21, Aug. 1993.

[46] P. Dasgupta et al. The Design and Implementation of the Clouds Distributed Operating System. *Journal of the USENIX Association*, 3(1):11–46, Winter 1990.

[47] R. Camacho Eugenio Oliveira and C. Ramos. A Multi-agent Environment in Robotics. *Robotica*, 9(4):431–440, Oct-Dec 1991.

[48] Chris Fedor. TCX task communications. School of computer science / robotics institute report, Carnegie Mellon University, 1993.

[49] S. Fleury, M. Herrb, and R. Chatila. Design of a Modular Architecture for Autonomous Robot. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 3508–13, San Diego, CA, May 1994.

[50] Gene F. Franklin, J. David Powell, and Michael Workman. *Digital Control of Dynamic Systems*. Series in Electrical and Computer Engineering: Control Engineering. Addison-Wesley, Reading, MA, second edition, 1990.

[51] Li-Chen Fu and Yung-Jen Hsu. Fully automated two-robot flexible assembly cell. In *IEEE International Conference on Robotics and Automation*, pages 332–338, Atlanta, Georgia, USA, May 1993.

[52] Elmer G. Gilbert and Daniel W. Johnson. Distance functions and their application to robot path planning in the presence of obstacles. *IEEE Transactions on Robotics and Automation*, RA-1(1):21–30, September 1985.

[53] A. Goscinski. *Distributed Operating Systems The Logical Design*. Addison-Wesley, 1st edition, 1991.

[54] S. Graves, L. Ciscon, and J. D. Wise. A modular software system for distributed telerobotics. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 2783–85, Nice, France, May 1992.

[55] Maki K. Habib, Shooji Suzuki, Shin'ichi Yuta, and Jun'ichi Iijima. New language structure for sensor-based actions to describe the real-time behaviour of autonomous robots. *International Journal of Electronics*, 70(4):653–670, 1991.

[56] J. K. Hackett and M. Shah. Multi-sensor fusion: A perspective. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 359–66, Cincinnati, OH, May 13-18 1990.

[57] G. Hager and M. Mintz. Computational Methods for Task-directed Sensor data Fusion and Sensor Planning. *The International Journal of Robotics Research*, 10(4):285–313, August 1991.

[58] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1993.

[59] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. STATEMATE: a working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–14, 1990.

[60] S. Hayati. Hybrid position/force control of multi-arm cooperating robots. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 82–89, San Francisco, CA, April 1986.

[61] B. Hayes-Roth. A blackboard architecture for control. *Artificial Intelligence*, 26(3):251–321, 1985.

[62] J. Henshall and A. Shaw. *OSI Explained. End to End Computer Communications Standards*. Ellis Horwood, Chinchester, England, 1988.

[63] R. Hinkel, T. Knieriemen, and E.V. Puttkamer. MOBOT-III-an autonomous mobile robot for indoor applications. In R.A. Jarvis, editor, *Proceedings of the International Symposium and Exposition on Robots. Designated the 19th ISIR by the International Federation of Robotics*, Robots: Coming of Age, pages 489–504, Sydney, NSW, Australia, 6-10 Nov 1988. IFS Publications.

[64] G. Hirzinger, B. Brunner, J. Dietrich, and J. Heindl. Sensor-Based Spaced Robotics–ROTEX and its Telerobotic Features. *IEEE Transactions on Robotics and Automation*, 9(5):649–63, October 1993.

[65] Neville Hogan. Impedance control: An approach to manipulation. *Transactions of the ASME, Journal of Dynamic Systems, Measurement, and Control*, 107:1–24, March 1985.

[66] Michael G. Hollars. *Experiments in End-Point Control of Manipulators with Elastic Drives*. PhD thesis, Stanford University, Department of Aeronautics and Astronautics, Stanford, CA 94305, May 1988. Also published as SUDAAR 568.

[67] Andreas Hormann. On-Line Planning of Action Sequences for a Two-Arm Manipulator System. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1109–1114, Nice, France, May 1992.

[68] S. Ims. Inertia measurement system. ARL Memo 75, Stanford University, Department of Aeronautics and Astronautics, Stanford, CA 94305, July 1991.

[69] IONA Technologies Ltd., 8-34 Percy Place, Dublin 4, Ireland. *Orbix - A Technical Overview*, August 1993.

[70] IONA Technologies Ltd., 8-34 Percy Place, Dublin 4, Ireland. *The Orbix Architecture*, August 1993.

[71] C. Jacquemot, P Gautron, H.G. Baumgarten, F. Herrmann, J. Mukerji, H. Hartlage, and P.S. Jensen. COOL: The CHORUS CORBA Complient Framework. Technical report, CHORUS Systemes, 1983.

[72] S. Dubowsky J.E. Bobrow and J.S. Gibson. Time-optimal control of robotic manipulators along specified paths. *International Journal of Robotics Research*, 4(3), Fall 1985.

[73] J. P. Jones, P. L. Butler, S. E. Johnston, and T. G. Heywood. Hetero Helix: synchronous and asynchronous control systems in heterogeneous distributed networks. *Robotics and Autonomous Systems*, 10(2-3):85–99, 1992.

[74] Judson P. Jones, Alex L. Bangs, and Philip L. Butler. A system for simulating shared memory in heterogeneous distributed-memory networks with specializations for robotics applications. In *IEEE International Conference on Robotics and Automation*, pages 2738–2744, Nice, France, May 1992.

[75] M. Jourdan and F. Marianinchi. A Modular State/Transition Approach for programming Reactive Systems. Mouriel.Jourdan@imag.fr, 1994.

[76] E. Jul, H. Levy, N. Hutchingson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6:105–133, February 1988.

[77] L. Kavraki, J.C. Latombe, and R.H. Wilson. On the Complexity of Assembly Partitioning. *Information Processing Letters*, 48:229–235, 1993.

[78] A. Kemper and M. Wallrath. An analysis of geometric modeling in database systems. *ACM Computing Surveys*, 19(1):47–91, March 1987.

[79] O. Khatib. Augmented Object and Reduced Effective Inertia in Robot Systems. In *American Control Conference*, volume 3, pages 2140–7, Atlanta, GA, June 15-17 1988.

[80] Oussama Khatib. Real time obstacle avoidance for manipulators and mobile robots. *International Journal of Robotics Research*, 5(1), 1986.

[81] Khashayar Khorasani and M. W. Spong. Invariant manifolds and their application to robot manipulators with elastic joints. In *Proceedings of the International Conference on Robotics and Automation*, pages 978–983, St. Louis, MO, March 1985. IEEE, IEEE Computer Society.

[82] Yoshihito Koga. *On Multi-Arm Trajectory Planning*. PhD thesis, Stanford University, Department of Mechanical Engineering, Stanford, CA 94305, September 1994. Also published as STAN-CS-TR-94-1530.

[83] Petar V. Kokotovic, R. E. O'Malley, Jr., and P. Sannuti. Singular perturbations and order reduction in control theory—an overview. *Automatica*, 12:123–132, 1976.

[84] John Koza. *Genetic Programming : on the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[85] E. Krotkov and R. Hoffman. Terrain Mapping for a Walking Planetary Rover. *IEEE Transactions on Robotics and Automation*, 10(6):728–39, December 1994.

[86] J. C. Latombe, C. Laugier, J. M. Lefebvre, E. Mazer, and J. F. Miribel. The LM robot programming system. In H. Hanafusa and H. Inoue, editors, *Second International Symposium on Robotics Research*, chapter 7, pages 377–391. MIT Press, Cambridge, MA, 1985.

[87] J.C. Latombe. *Robot Motion Planning*. Kluger Academic Publishers, Boston, MA, 1991.

[88] W.G. Levelt, M.F. Kaashoek, H.E. Bal, and A.S. Tanenbaum. Distributed Programming with Shared Data. *Computer Languages*, 16(2):129–46, 1991.

[89] W.G. Levelt, M.F. Kaashoek, H.E. Bal, and A.S. Tanenbaum. A Comparison of Two Paradigms for Distributed Shared Memory. *Software–Practice and Experinece*, 22(11):985–1010, Novemeber 1992.

[90] H.M. Levy and E. Tempero. Modules, Objects and Distributed Programming: Issues in RPC and Remote Object Invokation. *Software–Practice and Experinece*, 21(1):77–90, January 1991.

[91] T.-Y. Li. *On-Line Robot Motion Planning in Dynamic Environments*. PhD thesis, Stanford University, Department of Mechanical Engineering, Stanford, CA 94305, June 1995. To be Published.

[92] Tsai-Yen Li and Jean-Claude Latombe. On-Line Motion Planning for Two Robot Arms in a Dynamic Environment. In *Proceedings of the IEEE International Conference on Robotics and Automation*, volume 1, Nagoya, Japan, May 1995.

[93] B. Liskov and Shira. Promises: Linguistic Support for Efficirnt Asynchronous Procedure Calls in Distributed Systems. In *Proceedings of the SIGPLANN'88 Conference on Programming Language Design and Implementation*, pages 260–267, Atlanta, GA, June 22-24 1988.

[94] THK Co. LTD. THK Ball Screw-Spline type BNS. Catalog No. 107-3E, 1992.

[95] T. C. Lueth and U. Rembold. Extensive Manipulation Capabilities and Reliable Behavior at Autonomous Robot Assembly. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 3495–500, San Diego, CA, May 1994.

[96] J. Y. S. Luh, W. B. Fisher, and R.P. Paul. Joint torque control by direct feedback for industrial robots. *IEEE Transactions on Automatic Control*, AC-28(2), Febuary 1983.

[97] J.Y.S. Luh and C.S. Lin. Optimum path planning for mechanical manipulators. *Transactions of the ASME*, 102:142–151, June 1981.

[98] R. Lumia, J. Fiala, and A. Wavering. The NASREM Robot Control System and Testbed. *International Journal of Robotics and Automation*, 5(1):20–26, 1990.

[99] R. Lumia, J. L. Michaloski, R. Russel, T. E. Wheatley, P. G. Backes, S. Lee, and R. D. Steele. Unified Telerobotic Architecture Project (UTAP) Interface Document. Technical report, NIST, Intelligent Systems Division, NIST, Gaithersburg, MD, June 18 1994.

[100] R. C. Luo and M. G. Kay. Multisensor Integration and Fusion in Intelligent Systems. *IEEE Transactions on Systems, Man, and Cybernetics*, 19(5):901–29, September/October 1989.

[101] Martti Mantyla. *An Introduction to Solid Modeling*. Computer Science Press, Rockville, MD, 1988.

[102] Richard L. Marks. *Experiments in Visual Sensing for Automatic Control of an Underwater Robot*. PhD thesis, Stanford University, Stanford, CA 94305, (August) 1995. To be Published.

[103] MBARI (Monterrey Bay Aquarium Research Institute. Data manager user's guide. Internal Documentation, 1991.

[104] T. W. McLain. *Experiments in the Coordinated Control of an Underwater Arm/Vehicle System*. PhD thesis, Stanford University, Department of Mechanical Engineering, Stanford, CA 94305, (December) 1995. To be Published.

[105] David W. Meer. *Experiments in Cooperative Manipulation of Flexible Objects*. PhD thesis, Stanford University, Department of Mechanical Engineering, Stanford, CA 94305, 1994.

[106] David W. Meer and Stephen M. Rock. Experiments in object impedance control for flexible objects. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1222–1227, San Diego, CA, May 1994.

[107] Sun Microsystems. RPC: Remote Procedure Call Protocol Specification Version 2. Internet Network Working Group Requets for Comments RFC 1057, Network Information Center, SRI International, June 1988.

[108] Sun Microsystems. XDR: External Data Representation Standard. Internet Network Working Group Requets for Comments RFC 1014, Network Information Center, SRI International, June 198t.

[109] D.J. Miller and R.C. Lennox. An Object-Oriented Environment for Robot System Architectures. *IEEE Control Systems Magazine*, 11(2):14–23, February 1991.

[110] D. L. Mills. Internet Time Synchronization: the Network Time Protocol. *IEEE Transactions on Communications*, 10(39):1482–93, 1991.

[111] C. Mirolo and E. Pagello. A Solid Modeling System for Robot Action Planning. *IEEE Computer Graphics and Applications*, 9(1):55–69, Jan. 1989.

[112] D. Morse. *Experiments in Multi-Rate, Multi-Bandwidth Control of a Flexible-Drive Manipulator With a Mini-Manipulator*. PhD thesis, Stanford University, Department of Aeronautics and Astronautics, Stanford, CA 94305, (August) 1995. To be Published.

[113] S. Mullender, editor. *Distributed Systems*. ACM Press, Frontier. Addison Wesley, 1989.

[114] Y. Nakamura, K. Nagai, and T. Yoshikawa. Mechanics of coordinative manipulation by multiple robotic mechanisms. In *Proceedings of the IEEE International Conference on Robotics and Automation*, volume 2, pages 991–998, Raleigh, NC, April 1987. IEEE, IEEE Computer Society.

[115] Yoshihiko Nakamura and Yingti Xu. An Architecture of Inteligent Controller for Multi-Sensor Robotic Systems. In *Proceedings of the International Symposium and Exposition on Robots. Robots: Coming of Age.*, Robots: Coming of Age, pages 550–92, Sydney, NSW, Australia, 6-10 Nov 1988. IFS Publications.

[116] F. Nashashibi and M. Devy. 3-D Incremental Modeling and Robot Localization in a Structured Environment using Laser Range Finder. In *Proceeding IEEE International Conference on Robotics and Automation*, volume 1, pages 20–7, Atlanta, GA, May 2-6 1993.

[117] B. Nitzberg and V. Lo. Distributed shared memory: A survey of uses and algorithms. *IEEE Computer*, 24(8):52–60, June 1991.

[118] F. R. Noreils. Toward a robot architecture integrating cooperation between mobile robots. *The International Journal of Robotics Research*, 12(1):79–98, February 1993.

[119] F. R. Noreils and R. G. Chatila. Plan Execution Monitoring and Control Architecture for Mobile Robots. *IEEE Transactions on Robotics and Automation*, 11(2):255–266, April 1995.

[120] F.R. Noreils and R.G. Chatila. Control of Mobile Robot Actions. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 701–7, Scottsdale, AZ, April 1989.

[121] The Object Manager Group, Framingham Corporate Center, 492 Old Connecticut Path, Framingham, MA 01701-4568. *The Common Object Request Broker: Architecture and Specification*, revision 1.2 edition, December 1993.

[122] Michel Occello and Marie-Claude Thomas. A Distributed Blackboards Methodology for Designing Robotic Control Softwares. In *IEEE International Conference on Systems Engineering*, pages 147–150. IEEE, 1992.

[123] G. Pardo-Castellote and S. A. Schneider. The Network Data Delivery Service: A Real-Time Data Connectivity System. In *Proceedings of the AIAA/NASA Conference on Intelligent Robots in Field, Factory, Service and Space*, volume II, pages 591–7, Houston, TX, March 1994. AIAA, AIAA.

[124] G. Pardo-Castellote and S. A. Schneider. The Network Data Delivery Service: Real-Time Data Connectivity for Distributed Control Applications. In *Proceedings of the International Conference on Robotics and Automation*, San Diego, CA, May 1994. IEEE, IEEE Computer Society.

[125] G. Pardo-Castellote, S. A. Schneider, and R. H. Cannon Jr. System Design and Interfaces for Intelligent Manufacturing Workcell. In *Proceedings of the International Conference on Robotics and Automation*, Nagoya, Japan, May 1995. IEEE, IEEE Computer Society.

[126] Gerardo Pardo-Castellote and Henrique A. S. Martins. Real-Time Motion Scheduling for a SMALL Workcell. In *Proceedings of the International Conference on Robotics and Automation*, volume 1, pages 810–817, Sacramento, California, April 1991.

[127] D. W. Payton, J. K. Rosenblatt, and D. M. Keirsey. Plan Guided Reaction. In *IEEE Transactions on Systems, Man and Cybernetics* [128], pages 1370–82.

[128] D.W. Payton and T.E. Bihari. Intelligent real-time control of robotic vehicles. *Communications of the ACM*, 34(8):48–63, August 1991.

[129] Lawrence Pfeffer, Oussama Khatib, and John Hake. Joint torque sensory feedback in the control of a PUMA manipulator. In *Proceedings of the American Control Conference*, pages 818–824, Seattle, WA, June 1986.

[130] Lawrence E. Pfeffer. The RPM toolbox: A system for fitting linear models to frequency response data. In *Proceedings of the 1993 MATLAB User's Conference*, Cambridge, MASS, October 1993.

[131] Lawrence E. Pfeffer. *The Design and Control of a Two-Armed, Cooperating, Flexible-Drivetrain Robot System*. PhD thesis, Stanford University, Stanford, CA 94305, December 1993. Also published as SUDAAR 644.

[132] Lawrence E. Pfeffer and Robert H. Cannon Jr. Experiments with a Dual-Armed, Cooperative, Flexible-Drivetrain Robot System. In *Proceedings of the International Conference on Robotics and Automation*, pages 601–608, Atlanta, GA, May 1993. IEEE, IEEE Computer Society. Flex Primarily in Drive NOT Links, Tight Inner Loop.

[133] Lawrence E. Pfeffer, Oussama Khatib, and John Hake. Joint torque sensory feedback in the control of a PUMA manipulator. *IEEE Transactions on Robotics and Automation*, 5(4):418–425, August 1989.

[134] Judson P. Jones Philip L. Butler. A Modular Control Architecture for Real-Time Synchronous and Asynchronous Systems. In *Proceedings of the SPIE - Applications of Artificial Intelligence: Machine Vision and Robotics*, volume 1964, pages 287–298. SPIE, 1993.

[135] J. Postel. User datagram protocol. RFC 768, June 1980.

[136] J. M. Purtilo. The POLYLITH Software Bus. *ACM Transactions on Programming Languages and Systems*, 16(1):151–73, January 1994.

[137] S. Quinlan and O. Khatib. Elastic Bands: Connecting Path Planning and Control. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 802–7, San Diego, CA, May 1994.

[138] Sean Quinlan. *Real-Time Modification of Collision-Free Paths*. PhD thesis, Stanford University, Department of Computer Science, Stanford, CA 94305, 1994. Also published as STAN-CS-TR-94-1537.

[139] R. Quintero and A.J. Barbera. A Real-Time Control System Methodology for Developing Intelligent Control Systems. Technical Report NISTIR 4936, NIST, October 1992.

[140] R. K. Raj, E. Tempero, and H.M. Levy. Emerald: A General-Purpose Programming Language. *Software–Practice and Experinece*, 21(1):91–118, January 1991.

[141] S.C.V. Raju and A.C. Shaw. A Prototyping Environment for Specifying, Executing and Checking Communicating Real-Time State Machines. *Software–Practice and Experinece*, 24(2):175–95, February 1994.

[142] A. Ramadorai, T.J. Tarn, A. K. Bejczy, and N. Xi. Task-Driven Control of Multi-Arm Systems. *IEEE Transactions on Robotics and Automation*, 2(3):198–206, September 1994.

[143] Bahram Ravani. World Modeling for CAD-based Robot Programming and Simulation. *International Journal of Robotics and Automation*, 4(2):96–105, 1989.

[144] Real-Time Innovations, Inc., 954 Aster, Sunnyvale, CA 94086. *ControlShell: Object-Oriented Framework for Real-Time System Software User's Manual*, 4.0a edition, June 1991.

[145] Real-Time Innovations, Inc., 954 Aster, Sunnyvale, CA 94086. *NDDS: The Network Data-Delivery Service User's Manual*, 1.7 edition, November 1994.

[146] Eberhardt Rechtin. *Systems Architecting: Creating and Building Complex Systems*. Prentice-Hall, first edition, 1991.

[147] E. Rimon and D.E. Koditschek. Exact Robot Navigation Using Artificial Potential Functions. *IEEE Transactions on Robotics and Automation*, 8(5):501–18, Oct. 1992.

[148] Paul S. Rosenbloom, John E Laird, Allen Newell, and Robert McCarl. A Preliminary Analysis of the Soar Architecture as a Basis for General Intelligence. *Artificial Intelligence*, 47(1-3):289–325, January 1991.

[149] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Hermann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. Overview of the CHORUS Distributed Operating System. Technical report, CHORUS Systemes, February 1991.

[150] J. Russakow, O. Khatib, and S. M. Rock. Extended Operational Space Formulation for Serial-to-Parallel Chain (Branching) Manipulators. In *Proceedings of the IEEE International Conference on Robotics and Automation*, Nagoya, Japan, May 1995. IEEE, IEEE Computer Society.

[151] J. Russakow and S. M. Rock. Generalized Object Control and Assembly for Space Robots. In *Proceedings of the ASCE: Robotics for Challenging Environments*, Albuquerque NM, February 28 - March 3 1994.

[152] Jeff Russakow. *Experiments in Cooperative Assembly by Multiple Free-Flying Multi-Arm Robots*. PhD thesis, Stanford University, Department of Aeronautics and Astronautics, Stanford, CA 94305, (December) 1995. To be Published.

[153] A. Schill, editor. *DCE - The OSF Distributed Computing Environment Client/Server Model and Beyond*, International DCE Workshop, Karlsruhe, Germany, October 7-8 1993. Springer-Verlag.

[154] S. Schneider. *Experiments in the Dynamic and Strategic Control of Cooperating Manipulators*. PhD thesis, Stanford University, Stanford, CA 94305, September 1989. Also published as SUDAAR 586.

[155] S. Schneider and R. H. Cannon. Object Impedance Control for Cooperative Manipulation: Theory and Experimental Results. *IEEE Transactions on Robotics and Automation*, 8(3), June 1992. Paper number B90145.

[156] S. Schneider and L. Pfeffer. Inertia measurement pendulum documentation. ARL Memo 28, Stanford University, Department of Aeronautics and Astronautics, Stanford, CA 94305, Sep. 1987.

[157] S. A. Schneider and R. H. Cannon. Experimental Object-Level Strategic Control With Cooperating Manipulators. *The International Journal of Robotics Research*, 12(4):338–350, August 1993.

[158] S. A. Schneider, V. Chen, and G. Pardo-Castellote. Object-Oriented Framework for Real-Time System Development. In *Proceedings of the IEEE International Conference on Robotics and Automation*, Nagoya, Japan, May 1995. IEEE, IEEE Computer Society.

[159] S. A. Schneider, V. W. Chen, and G. Pardo-Castellote. ControlShell: A Real-Time Software Framework. In *Proceedings of the AIAA/NASA Conference on Intelligent Robots in Field, Factory, Service and Space*, volume II, pages 870–7, Houston, TX, March 1994. AIAA, AIAA.

[160] S. A. Schneider, M. A. Ullman, and V. W. Chen. ControlShell: A Real-Time Software Framework. In *Proceedings of the 1991 IEEE International Conference on Systems Engineering*, Dayton, OH, August 1991.

[161] Alan C. Shaw. Communicating Real-Time State Machines. *IEEE Transactions on Software Engineering*, 18(9):805–816, September 1992.

[162] Z. Shiller and S. Dubowsky. Robot path planning with obstacles, actuator, gripper and payload constraints. *International Journal of Robotics Research*, 8(6), December 1989.

[163] Z. Shiller and S. Dubowsky. On computing the global time-optimal motions of robotic manipulators in the presence of obstacles. *IEEE Transactions on Robotics and Automation*, 7(6):785–797, December 1991.

[164] Kang G. Shin and Neil D. McKay. Minimum-time control of robotic manipulators with geometric path constraints. *IEEE Transactions on Automatic Control*, AC-30(6):531–541, June 1985.

[165] Kang G. Shin and Neil D. McKay. A dynamic programming approach to trajectory planning of robotic manipulators. *IEEE Transactions on Automatic Control*, AC-31(6):491–500, June 1986.

[166] Kang G. Shin and Neil D. McKay. Minimum-time trajectory planning for industrial robots with general torque constraints. In *IEEE International Conference on Robotics and Automation*, pages 412–415, San Francisco, California, April 6-10 1986.

[167] Kang G. Shin and Neil D. McKay. Selection of near-minimum time geometric paths for robotic manipulators. *IEEE Transactions on Automatic Control*, AC-31(6):501–511, June 1986.

[168] Michael D. Sidman, Franco E. DeAngelis, and George C. Verghese. Parametric system identification on logarithmic frequency response data. *IEEE Transactions on Automatic Control*, 36(9):1065 – 1070, Sept. 1991.

[169] R. Simmons, L.-J. Li, and C. Fedor. Autonomous task control for mobile robots. In *Proceedings of the IEEE International Simposium on Intelligent Control*, pages 663–8, Philadelphia, PA, Sept 1990.

[170] R. G. Simmons. Concurrent planning and execution for autonomous robots. *IEEE Control Systems Magazine*, 12(1):46–50, Feb. 1992.

[171] R. G. Simmons. Structured control for autonomous robots. *IEEE Transactions on Robotics and Automation*, 10(1):34–43, Feb. 1994.

[172] D. Simon, B. Espiau, E. Castillo, and K. Kapellos. Computer-aided design of a generic robot controller handling reactivity and real-time control issues. Research Report 1801, INRIA, Le Chesnay, France, December 1992.

[173] B. Simons and A. Spector. *Fault-tolerant distributed computing*, chapter Argus (Distributed Program Language and System), pages 108–14. Springer-Verlag, 1990.

[174] S. Singh and M.C. Leu. Optimal trajectory generation for robotic manipulators using dynamic programming. *Transactions of the ASME*, 109:88–96, June 1987.

[175] Dale Skeen. An Information Bus Architecture for Large-Scale, Decision-Support Environments. In *Proceedings of the Winter 1992 USENIX Conference*, pages 183–95, San Francisco, CA, January 1992.

[176] J. E. Slotine and H. S. Yang. Improving the efficiency of time-optimal path-following algortihms. *IEEE Transactions on Robotics and Automation*, 5(1):118–124, April 1989.

[177] Sparta, Inc., 7926 Jones Branch Drive, McLean, VA 22102. *ARTSE product literature*.

[178] Hal Stern. *Managing NFS and NIS*. O'Reilly & Associates, Inc., 1991.

[179] H.D. Stevens, E.S. Miles, S. M. Rock, and R.H. Cannon. Object-Based Task-Level Control: A Hierarchical Control Architecture for Remote Operation of Space Robots. In *Proceedings of the AIAA/NASA Conference on Intelligent Robotics in Field, Factory, Service, and Space*, pages 264–73, Houston TX, March 1994.

[180] SunSoft Inc., 2550 Garcia Avenue, Mountain View, CA 94043. *Introduction to the ToolTalk Service*, September 1991.

[181] SunSoft Inc., 2550 Garcia Avenue, Mountain View, CA 94043. *Migrating to DOE*, May 1994.

[182] A. S. Tanenbaum. *Computer Networks*. Prentica Hall, Englewood Cliffs, New Jersey, 2nd edition, 1989.

[183] A.S. Tanenbaum and R. Van Renesse. Distributed Operating Systems. *ACM Computing Surveys*, 17(4):419–470, December 1985.

[184] S. C. A. Thomopoulos. Sensor Integration and Data Fusion. *Journal of Robotic Systems*, 7(3):337–72, June 1990.

[185] Aleksandar Timcenko, Steven Abrams, and Peter K. Allen. APHRODITE, Intelligent Planning, Control and Sensing in a Distributed Robotic System. In *IAS-3, International Conference on Intelligent Autonomous Systems*, pages 561–71, Amsterdam, Netherlands, 1993. IOS Press.

[186] M. M. Trivedi, M. A. Abidi, R. O. Eason, and R. C. Gonzalez. Developing Robotic Systems with Multiple Sensors. *IEEE Transactions on Systems, Man, and Cybernetics*, 20(6):1285–1300, December 1990.

[187] M. Uchiyama and T. Yamashita. Assymettric hybrid control of positions and forces of a dual arm robot to share loads. In *Experimental Robotics I: The First International Symposium*, pages 100–115, Philadelphia, Canada, June 1989.

[188] Christopher R. Uhlik. *Experiments in High-Performance Nonlinear and Adaptive Control of a Two-Link, Flexible-Drive-Train Manipulator*. PhD thesis, Stanford University, Department of Electrical Engineering, Stanford, CA 94305, May 1990. Also published as SUDAAR 592.

[189] M. A. Ullman. *Experiments in Autonomous Navigation and Control of Multi-Manipulator Free-Flying Space Robots*. PhD thesis, Stanford University, Stanford, CA 94305, March 1993. Also published as SUDAAR 630.

[190] R. L. Vasquez. *Experiments in Two-Cooperating-Arm Manipulation from a Platform with Unknown Motion*. PhD thesis, Stanford University, Department of Aeronautics and Astronautics, Stanford, CA 94305, January 1992. Also published as SUDAAR 617.

[191] M.I. Vuskovic, A.L. Riedel, and C. Q. Do. The Robot Shell. *International Journal of Robotics and Automation*, 3(3):165–76, 1988.

[192] E. F. Walker, R. Floyd, and P. Neves. Asynchronous Remote Operation Execution in Distributed Systems. In *Proceeding of the 10th International Conference on Distributed Computing Systems*, pages 253–259, Paris, France, May 28 - June 1 1990.

[193] C. Wang and D. Cannon. A Virtual End-Effector Pointing System in Point-and-Direct Robotics for Inspection of Surface Flaws Using a Neural Network Based Skeleton Transform. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 784–9, Atlanta, GA, May, 2-6 1993.

[194] Fei-Yue Wang and Geroge N. Saridis. Task translation and integration specification in intelligent machines. *IEEE Transactions on Robotics and Automation*, 9(3):257 – 271, June 1993.

[195] H. H. Wang, R. L. Marks, S. M. Rock, and M. J. Lee. Task-Based Control Architecture for an Untethered, Unmanned Submersible. In *Proceedings of the 8th Annual Symposium of Unmanned Untethered Submersible Technology*, pages 137–147. Marine Systems Engineering Laboratory, Northeastern University, September 1993.

[196] J. T. Wen and A. Desrochers. Sub-time-optimal control strategies for robotic manipulators. In *IEEE International Conference on Robotics and Automation*, San Francisco, California, April 6-10 1986.

[197] T. Westbrook. CAD software. *IEEE Transactions on Robotics and Automation*, 60(19):76–7,80–1,84,87–8, Aug 1990.

[198] T. Williams. Fiber network supports distributed real-time systems. *Computer Design*, 29(17):60–62, September 1990. dd.

[199] R. Wilson and J.C. Latombe. Geometric Reasoning About Mechanical Assembly. *accepted for publication in Artificial Intelligence Journal*, 1994.

[200] Wind River Systems, Inc., 1351 Ocean Ave., Emeryville, CA 94608. *VxWorks User's Manual*, 1988-1993.

[201] J. D. Wise and Larry Ciscon. *TelRIP Distributed Applications Environment Operating Manual*. Rice University, Houston Texas, 1992. Technical Report 9103.

[202] J.D. Wise, L.A. Ciscon, and S. Graves. A distributed telerobotics system for space operations. In *Proceedings of the SPIE - The International Society for Optical Engineering*, volume 1829 of *Cooperative Intelligent Robotics in Space III*, pages 359–7, Boston, MA, 1992. November 16-18.

[203] L. Zahn, T.H. Dineen, P.J. Leach, E.A. Martin, N.W. Mishkin, J.N. Pato, and G.L. Wyant. *Network Computing Architecture*. Prentice-Hall, 1990.

[204] F. Zanichelli, S. Caselli, A. Natali, and A. Omicini. A Multi-Agent Framework and Programming Environment for Autonomous Robotics. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 3501–7, San Diego, CA, May 1994.

[205] Y. F. Zheng and J. Y. S. Luh. Optimal load distribution for two robots handling a single object. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 344–349, Philadelphia, PA, April 1988.

[206] K. R. Zimmerman and R. H. Cannon Jr. GPS-Based Control for Space Vehicle Rendezvous. In *Proceedings of the ASCE: Robotics for Challenging Environments*, Albuquerque NM, February 28 - March 3 1994.

[207] Kurt Zimmermann. *Experiments in the use of Differential GPS for Space Vehicle Rendez-Vous*. PhD thesis, Stanford University, Stanford, CA 94305, (June) 1996. To be Published.