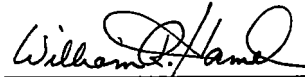


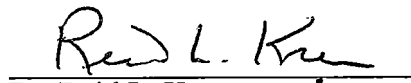
To the Graduate Council:

I am submitting herewith a thesis written by Veronika L. Gospodareva entitled "Finite-State Control of the Hybrid Telerobotic System". I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Mechanical Engineering.



Dr. William R. Hamel, Major Professor

We have read this thesis and
recommend its acceptance:


Dr. Gary V. Smith
Dr. Reid L. Kress

Accepted for the Council:



Associate Vice Chancellor and
Dean of the Graduate School

FINITE-STATE CONTROL OF THE HYBRID TELEROBOTIC SYSTEM

A Thesis
Presented for the
Master of Science
Degree
The University of Tennessee, Knoxville

Veronika L. Gospodareva
December 1999

Copyright ©, 1999 Veronika L. Gospodareva
All rights reserved.

Acknowledgements

I am very glad to have an opportunity to acknowledge some of the people who have supported and helped me in the research leading to the completion of this thesis.

I need to recognize that this work was performed under contract DE-AR26-97FT34314 for the Department of Energy. I would like to thank Dr. Hamel whose guidance and support professionally and personally is greatly appreciated. I would like to recognize Dr. Steve Everett who was the main force behind this research. I would like to show my gratitude to Matt Smith who helped me with all the C++ programming and all the ControlShell complications. Special thanks are required for my new friend Claudell Hatmaker for her friendship and help during some rocky times in my life. Most of all I would like to thank my family and my friends on two continents who are always with me in my dreams and my prayers.

Abstract

Summarizing the past history of robotics, we find on one hand, the unsuccessful search to give autonomy to robots, and on the other hand, the people who need this autonomy for their applications but are obligated to let man into the control loop while they wait for the successful outcome. The goal of the telerobotics is not only to construct the human - robot systems, but also to find the right balance of their cooperation. The hybrid system theory is one of the methods in modeling and optimization of the telerobotic systems. This thesis describes and analyzes a hybrid telerobotic system, constructs the hybrid system controller using the finite-state automata, and implements the theoretical findings into commercial program as part of the real-time application.

Contents

1	Introduction	1
2	Background	3
2.1	History of robotics.....	3
2.2	Teleoperation and Human - Machine cooperation.....	4
2.3	Telerobotics.....	5
2.4	Robot Task Space Analyzer.....	6
2.5	Task Planner.....	8
2.6	Telerobotic System Schematics.....	8
2.7	Finite-State control of the Hybrid system.....	8
2.8	ControlShell Implementation.....	10
3	Hybrid telerobotic system	11
3.1	Introduction.....	11
3.2	A general Hybrid System Model.....	11
3.3	Hybrid Telerobotic System.....	12
4	Finite-State Machine as a high level controller of the hybrid telerobotic system	17
4.1	Introduction.....	17
4.2	Basic model of the FSM.....	18
4.3	High-level controller for the hybrid telerobotic system.....	22
4.3.1	Main FSM.....	22
4.3.2	Autoexecution submachine.....	24
4.3.3	Mode changing Hybrid Controller Interface.....	26
5	ControlShell implementation	33
5.1	Introduction.....	33
5.2	ControlShell Structure.....	34
5.3	Controller Components.....	36
5.3.1	Controller App.....	36
5.3.2	DESController.....	39
5.3.3	AUTOEXEC composite state.....	41
5.3.4	Fetching and parsing the Task Planner instructions.....	43

5.3.5	RobotController COG.....	44
5.4	Robot closed loop control.....	47
5.5	Controller Interface.....	53
5.5.1	ControlShell menus interface.....	53
5.5.2	ControlShell messages.....	53
5.5.3	Minimaster menu screens.....	53
6	Conclusions	57
7	Future work	59
	Bibliography	61
	Appendices	64
1	Selected code for the Fetch_Parse ControlShell component.....	65
2	ControlShell menus line.....	70
3	Selected code for the miniMaster component.....	71
4	Selected code for the Communication component.....	76
	Vita	80

List of Tables

4.1	States, state transitions, modes, and system status.....	32
-----	--	----

List of Figures

2.1	Functional Architecture of a Robust Telerobotic System with RTSA.....	7
2.2	Architecture of the Telerobotic System with Task Plan.....	9
3.1	General Hybrid System Architecture.....	13
3.2	Hybrid Telerobotic System.....	14
4.1	Branch labeling.....	21
4.2	Types of branches.....	21
4.3	Transition Diagram of FSM for the Teleoperation/Autonomous System.....	22
4.4	Main FSM.....	23
4.5	Autoexecution submachine.....	25
4.6	Mode Interface of the Hybrid System.....	29
4.7	FSM diagram Part 1.....	30
4.8	FSM diagram Part 2.....	31
5.1	ControlShell Interface Structure.....	36
5.2	ControlShell application (APP) of the controller.....	37
5.3	Controller COG.....	38
5.4	Main FSM.....	40
5.5	AUTOEXEC.....	42
5.6	Example of the Task Planner text file.....	44
5.7	RobotController COG: Teleop mode selected.....	46
5.8	RobotCC COG: DirControl mode selected.....	48
5.9	RobotCC COG: ImpControl mode selected.....	49
5.10	DirControl COG.....	50
5.11	ImpControl COG.....	51
5.12	Robot COG.....	52
5.13	MasterCommunication COG.....	54
5.14	Welcome screen.....	55
5.15	Idle screen.....	56
5.16	Autoexecution screen.....	56

List of Acronyms

RTSA	Robot Task Space Analyzer
OS	operating system
HPU	hydraulic power unit
FSM	finite-state machine
DES	discrete event system
App	application
COG	composite object group
STC	state-transition component
DFC	data flow component
ATC	atomic component

Chapter 1

Introduction

"I see the world, and human outposts on other worlds and in space, filled with cousin-intelligence of two entirely different types. I see the silicon-intelligence (robots) that can manipulate numbers with incredible speed and precision and that can perform operations tirelessly and with perfect reproducibility; and I see carbon-intelligence (human beings) that can apply intuition, insight, and imagination to the solution of problems on the basis of what would seem insufficient data to a robot."

Isaac Asimov
(1920 - 1992)

In the past *Robotic* and *Teleoperation* technologies were two different fields of the research interest. The main goal of *Robotics* was the development of the full autonomous system and artificial intelligence. The main goal of *Teleoperation* was to achieve the maximum approximation of the teleoperation system performance in speed and accuracy to the autonomous system. *Telerobotics* combines these two fields together in aspiration to achieve the optimal cooperation between human and robot intelligences. The performance of the telerobotic system depends on the system architecture, the complexity of the given task, and the teleoperator experiences. In recent years there has been growing interest in how to eliminate the telerobotic system functional reliance on the teleoperator

CHAPTER 1. INTRODUCTION

skills during the task execution. One of the directions that can be taken is to create the hybrid teleoperator/computer controller. This controller can be possible only if the considerable information of the work environment, and the task can be planned before the actual execution.

The next chapter will give the background of the special device to collect geometrical and other information about the unstructured work environment Robot Task Space Analyzer, and its Task Planner expansion.

The integration of the Task Planner into the telerobotic system allows the system to operate not only manually, but also automatically, when it is needed and possible. The telerobotic system with preplanned task executions can be defined as hybrid. Chapter three analyzes this system in the concept of hybrid systems.

Many different approaches to modeling and controlling of the hybrid system are known. The proposed method is to control of the hybrid telerobotic system with the finite automata as a high-level controller. The introduction of the finite-state theory and the development of the finite-state controller for the hybrid telerobotic system is given in chapter four.

The challenge of the integration of the controller into the actual system is well known. The choice of the right software can make this integration uncomplicated and more efficient.

Chapter five will show the ControlShell implementation of the developed controller in detail.

The sixth chapter will present the conclusions. Lastly, chapter seven will discuss future work necessary to fulfill the analysis of hybrid telerobotic system architecture.

Chapter 2

Background

2.1 History of Robotics

From the pyramid-building pharaohs of Egypt through all of the history of technology, there has been concern about how best to extend the capability of human workers and in the future to exclude them from heavy or dangerous operations. The enigma of human mechanical slaves has been around for more than three thousand years. From the first simple machines and their functions by Archimedes of Syracuse to Rene Descartes Cartesian coordinate system and Robert Willis introduction of word "kinematics" each of known mathematicians and engineers in history of civilization had his contribution of solving the mystery of human mechanical double [1]. But each new creation always had produced a new mystery.

Twentieth century science fiction gave this human mechanical double its name – robot and its identity – force through intelligence. The fiction became reality and the new science of robotics was born somewhere in the late 50's when George Devol and Joseph Engelberger formed the first industrial robot production company, Unimation and organized the first laboratory of robotics research [2].

The explosive progress of computers in the second half of the century boosted the robotics technology and it looked like mankind could succeed in transforming the

CHAPTER 2. BACKGROUND

everlasting dream of the human double into reality by year 2000. But ironically, the science revolution gave us an understanding of the originality of the human brain and the importance of human-machine cooperation.

Consequently, by the end of the eighties, the human-robot cooperation became an emerging research area in robotics and machine intelligence.

2.2 Teleoperation and Human - Machine cooperation

Teleoperator [the term "teleoperator" was introduced by Edwin G. Johnsen in 1967] is a manipulator that requires a direct human operator control [3]. Many of what seems to be new developments in teleoperator technology actually had their beginnings long ago. For many centuries people have been using teleoperators in the form of simple arm extensions. "This process became more important since the start of the atomic physics and radioactivity researches in the beginning of the century. The engineering implications of production-scale nuclear weapon manufacture resulted in a new generation of remote handling technology. During this time, remote engineering projects occurred throughout the U.S. Atomic Energy Commission (AEC) complex, in Europe, and in the former Soviet Union" [4]. The research group led by Ray Goertz around 1945 had developed the first mechanical telemanipulator at Argonne National Laboratory [3]. This device was the ancestor of the whole family of manipulators known as master-slave manipulator. In present time, applications for teleoperators are numerous. They typically involve work conditions dangerous for humans, and when use of intelligent autonomous robot is impossible or inappropriate.

There are some reasons for the decision to involve the human operator in the robotic system control. The main reason is the human ability to recognize "situations" and make rapid judgments to deal with them. Also the cost and labor consumption of design and manufacturing of completely autonomous robotic system are much higher than that of corresponding remote control system.

CHAPTER 2. BACKGROUND

Modern control theory allows the teleoperators to solve most of the existing practical problems in robotics, but not always by using the most effective method. In teleoperation the two main characteristics of any system performance, accuracy and time execution are inversely proportional. Thus, the improvement of one of these criteria leads to the decrease of the other [5].

2.3 Telerobotics

One of the techniques to improve the efficiency of the teleoperation system is to combine the teleoperation with autonomous executions whenever it is possible and needed. This allows the system to use the human intelligence and autonomous robot speed and accuracy to achieve the optimal cooperation between human and robot. "A system that beneficially combines human interaction and automation in a single robot is defined as telerobot. Telerobot can be considered as modified robot or modified teleoperator. These two perspectives provide fundamentally different results. Symbolically, this can be summarized as

$$Tr = Teleoperator \cup Robot$$

$$tR = Robot \cup Teleoperator$$

$$Tr \neq tR$$

These simple expressions mask the complexity of the underlying differences. A Tr system is one that is foremost a teleoperator but that can be "programmed" to perform specific subtasks automatically. A tR system is a programmable robot that has been augmented with control features that allow it to be operated manually in teleoperator mode" [4]. In our research the Tr systems will be under consideration. One of the ways to improve the performance of the telerobotic system is to build a robust telerobotic system.

According to W. R. Hamel [4], "Robust" telerobot is defined as one that (1) provides effective teleoperability, (2) permits efficient in-situ subtask automation (3) is able to monitor and detect fault conditions reliably, (4) is able to facilitate smooth or bumpless transfer between manual and atomic control modes, and (5) performs these functions reliably under realistic task conditions."

2.4 Robot Task Space Analyzer

To create robust telerobotic systems, complete information about the environment is needed. A system to collect the needed geometrical data and other information about the unstructured work environment has been carried out through research activity between the University of Tennessee - Knoxville and the Carnegie Mellon University as the Robot Task Space Analyzer, RTSA, project. "RTSA is a tool for remote equipment operations that combines infrared laser and visible stereo imaging, human-interactive modeling and computer-based object recognition to build 3-D models of the immediate work zone in which a robot system is operating" [6]. The idea of RTSA is that a robot can be under teleoperator control during some subtasks, and under computer control during some other subtasks. As a result of the use of the RTSA, the operator creates the model of the task execution with all the information about the location and orientation of the parts of the environment, which are necessary for the autonomous execution. This complete model allows the operator to understand the degree of complexity of the future execution plan and to divide the overall task into subtasks to be performed by manual or autonomous control. The information about the total geometry of the environment allows autonomous subtasks to be accurately defined in an acceptable time. Figure 2.1 shows the functional architecture of a robust telerobot that includes the RTSA.

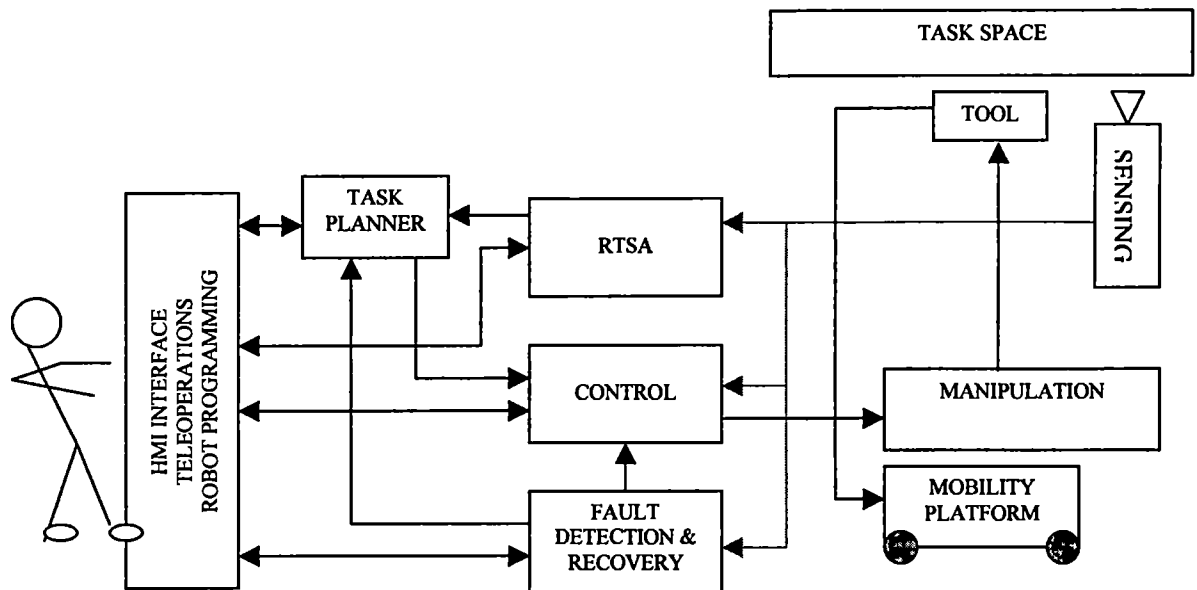


Figure 2.1: Functional Architecture of a Robust Telerobotic System with RTSA

2.5 Task Planner

Within the RTSA project the Task Planner is a special program, which uses the task space models of the work zone to assist the operator in defining the plan of the disassembling of the modeled environment. The planner allows the operator to choose the parts that are to be removed, tools to be used, the modes to be performed, and methods of the teleoperational assistance. To make the future control of this kind of robotic system possible the task planner follows the principle that any complex task may be broken down into a set of simple subtasks, called atomic tasks. Four types of subtasks have been defined: move, toggle gripper, toggle tool power, and teleoperation [7]. After the task planner completes the task plan file, this file is downloaded to the control computer for task execution.

2.6 Telerobotic system schematic

The overall hardware architecture of the telerobotic system is shown in Figure 2.2. After the completion the RTSA and task plan procedures the task plan file is downloaded to the real-time computer. The real-time computer is a controller for the task execution. This computer is connected with the robot through the Schilling C-30 interface [8]. During a manual execution the minimaster receives commands from the teleoperator and manipulates the slave through the real-time computer. During an automatic execution the task plan file commands the slave through the real-time computer.

2.7 Finite-State control of the Hybrid system

"Hybrid means, in general, heterogeneous in nature or composition" [9]. A system that optimizes the capability of both machines and humans as intelligent and cooperative entities is defined as a hybrid system. A telerobotic system with balanced choice of the

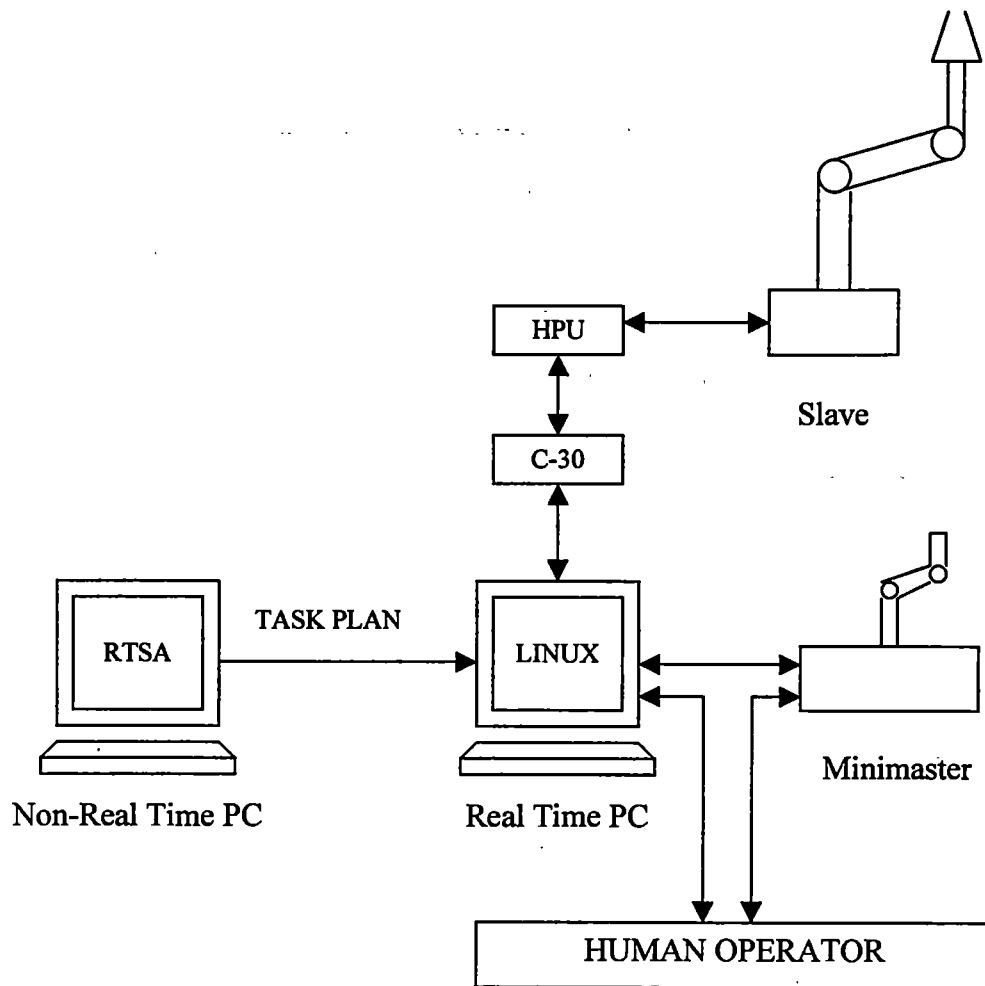


Figure 2.2: Architecture of the Telerobotic System with Task Plan

CHAPTER 2. BACKGROUND

autonomous or teleoperational task executions is a classical example of the hybrid discrete-continuous control system. Traditionally, in robotics, hybrid system control assumes the hybrid force-position control. Just recently a new hybrid concept involving discrete-continuous robotic systems has been considered. B.J. McCarragher and H. Asada applied hybrid system structures to formulate transitions between constrained motions for a peg-in-hole task in a robotic manufacturing system and used Petri nets to highlight the key components [10]. T.J. Tarn and N. Xi proposed a hybrid approach based on a Max-Plus Algebra model for method of control of the multiple segment robotic operations [11].

In this work planned tasks are characterized as discrete event systems and modeled using a finite state machine. The idea to use a finite-state machine as a high level controller for a set of tasks is not new in the telerobotics. M. Brooks proposed a pattern-matching task controller for a telerobot, which is taught by a human operator's demonstration of skills. Brooks visualized any given manipulation task as a network of a small number of component motions such as "grasp a tool", "move tool until touches workpieces", etc. Each component motion had starting and stopping states [12]. The proposed finite-state controller is very similar to the Brooks' architecture. The main difference is an addition of a task planner. The proposed control is a finite state control, with state changing rules dictated by a human operator, task planner, and a computer.

2.8 Control Shell implementation

"ControlShell is a development tool from Real-Time Innovations used for designing and implementing real-time applications. The program can create flow diagrams and link them with finite-state machines to configure the behavior of a system" [13]. ControlShell software has been selected to implement the control of the hybrid system being discussed here, because it provided graphically oriented, high-level control system programming.

Chapter 3

Hybrid telerobotic system

3.1 Introduction

Traditionally, robotics hybrid system control assumes hybrid force-position control. Just recently new definitions of hybrid system in robotics are being considered: hybrid teleoperator/robot systems [2] and hybrid discrete-continuous robotic systems [10, 11]. The teleoperator/autonomous system with preplan task scheduling can be classified as hybrid teleoperator/robot system and, also, as hybrid discrete-continuous robotic system, because of the discrete nature of the task scheduling. This kind of telerobotic system can be called a double hybrid. The control of the teleoperation/autonomous system with preplan task scheduling is very complicated. It requires the integration of the task scheduling control and teleoperator control to form a double hybrid system structure.

3.2 A general Hybrid System Model.

Hybrid control systems typically arise from the interaction of discrete planning algorithms and continuous processes, and, as such, they provide the basic framework and methodology for the analysis and synthesis of teleoperation/autonomous systems. The term hybrid refers to a mixing of two fundamentally different types of objects or

CHAPTER 3. HYBRID TELEROBOTIC SYSTEM

methods, usually discrete and continuous, which interact with each other. A look at literature clearly shows that there are many approaches to modeling, analysis, and synthesis of hybrid systems. They can be characterized and described along several dimensions. Approaches differ with respect to the importance or the complexity of the continuous and discrete subsystems. The architecture that can be applied to particular case of the hybrid telerobotic system is shown in Figure 3.1 [14]. This hybrid system includes three blocks: DISCRETE EVENT SYSTEM, CONTINUOUS SYSTEM and INTERFACE. Different levels are used at each layer of the hierarchy. In the bottom layer the model is usually described by means of differential and /or difference equations. This layer contains the actual system and any conventional controllers working at the same level of abstraction. In the top layer the model description is more abstract. Typical choices of description language at this level are finite state machines, fuzzy logic, Petri nets, etc. Usually the controllers designed at this level are discrete event supervisory controllers. The two levels communicate by means of an interface that plays the role of a translator between signals and symbols.

3.3 Hybrid Telerobotic System

Integration of an autonomous mode into an otherwise teleoperation system is useful in a variety of situations where the teleoperation system alone would not be sufficient. This integration allows both the human and the robot to perform tasks for which each is best. Figure 3.2 is a diagram for the hybrid telerobotic system.

Autonomous and manual operations are separated into two different operating modes. At each time instant, only one of the modes is active. The vertical dashed line shows the two subsystems of the hybrid telerobotic system: teleoperator control subsystem and autonomous control subsystem. Each subsystem represents one of the operating modes: teleoperation or autonomous. Based on logical conditions, the operating modes can be switched. In our control strategy the operator represents the "toggle",

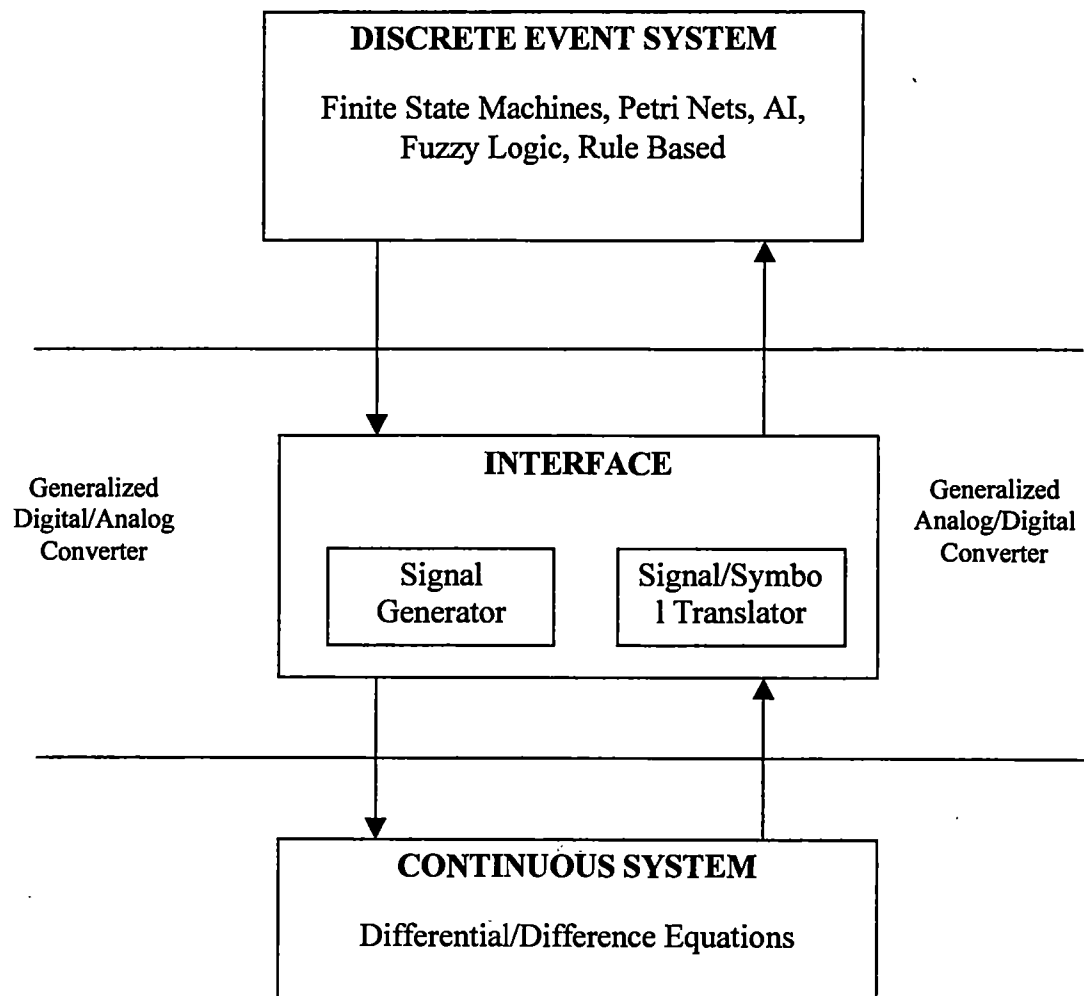


Figure 3.1: General Hybrid System Architecture

TELEOPERATION

AUTONOMOUS

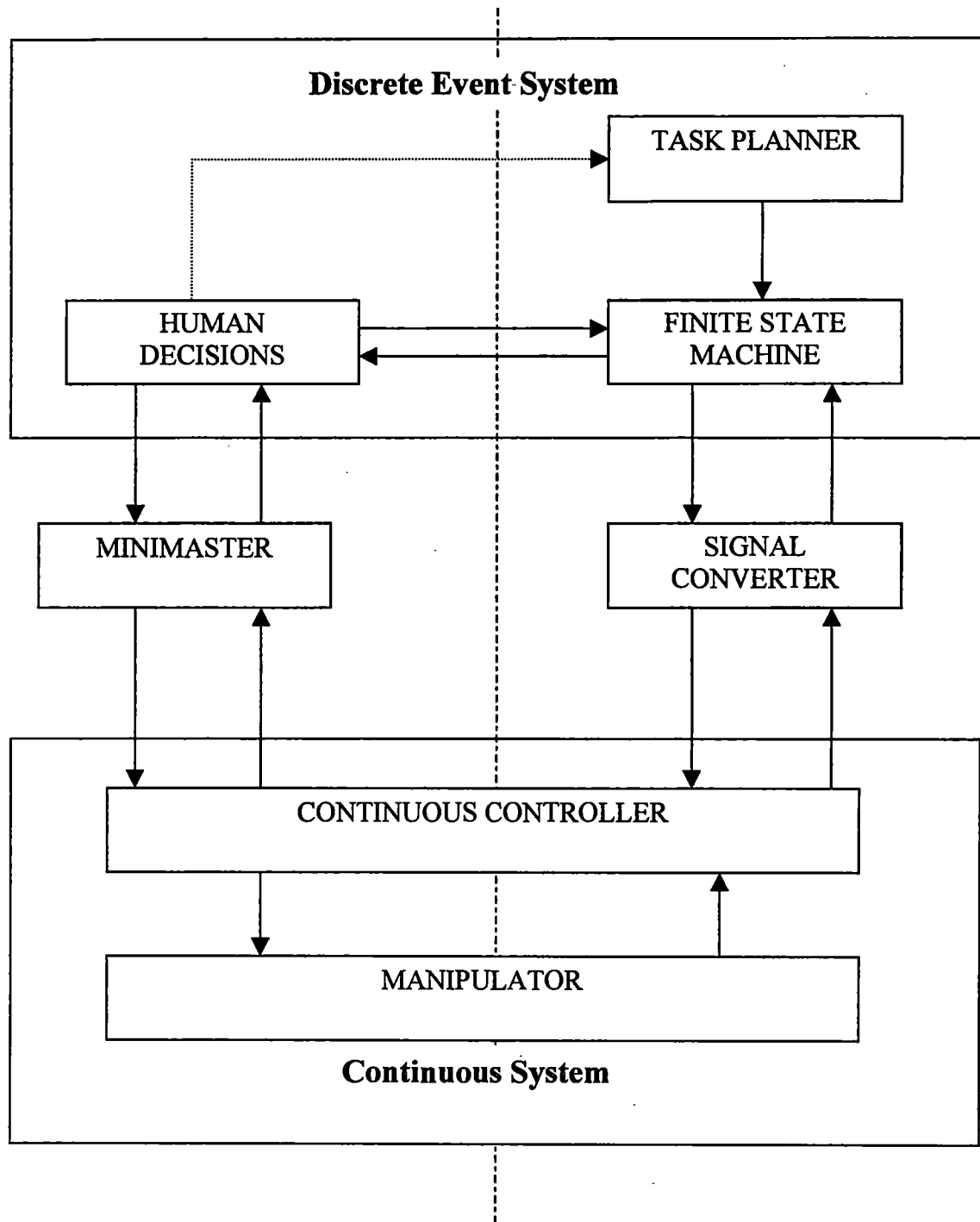


Figure 3.2: Hybrid Telerobotic System

CHAPTER 3. HYBRID TELEROBOTIC SYSTEM

therefore the HUMAN OPERATOR is a part of the Event Discrete System. If the execution switches to the autonomous mode, the TASK PLANNER takes control of the system. However, if some unexpected event, such as the occurrence of obstacles, happens in the autonomous operation, the operator can switch system back to the manual mode. After proper manual operation, replanning of the autonomous operation is required in order to switch the system back into the TASK PLANNER control. The dashed arrow in the diagram represents the replanning operation. The TASK PLANNER can switch execution back to the teleoperation mode, upon completion of the planning process. In this case, the TASK PLANNER notifies the operator when to switch back to the autonomous execution, and is responsible for the replanning strategy. During the autonomous execution the system can be asked to perform many different tasks that require different actions of the system. For example, to make a simple cut, the robot has to pick up the tool (system: open/close gripper), move to the cutting location (system: move) and do the cut (system: move and turn on/off the tool). The task plan is a text file containing a set of simple subtasks (atomic tasks). Each atomic task requires particular mode of the execution. The task plan is inherently a discrete process, which is represented by the higher layers of the hierarchy whereas the continuous time control laws that execute each atomic action represent the lower layers. The task plan is the other "toggle" of the system, and, therefore, the TASK PLANNER block is the other part of a Discrete Event System. The interface between the teleoperator, the task planner and the actual manipulator is very complicated. The finite state machine (FSM) was chosen as a high level controller of the system. The more detail explanation of the FSM is given in chapter 4.3. The FINITE STATE MACHINE is the third block of the Discrete Event System. The Discrete Event System interfaces with the Continuous System through two different interfaces. During autonomous task execution the task planner commands are sent to the CONTINUOUS CONTROLLER through the FINITE STATE MACHINE and SIGNAL CONVERTER. During teleoperation the MINIMASTER is the converter between the HUMAN OPERATOR and the CONTINUOUS CONTROLLER.

CHAPTER 3. HYBRID TELEROBOTIC SYSTEM

CONTINUOUS CONTROLLER and MANIPULATOR are two components of the Continuous System of the hybrid telerobotic system.

Chapter 4

Finite-State Machine as a high level controller of the hybrid telerobotic system

4.1 Introduction

Adding task-planned control to teleoperation is a key to improve the speed and accuracy of the telerobotic system. However, the control of this type of system is a very complicated issue. Because of its discrete nature the task planner requires the telerobotic system to behave as a discrete event system. But, robot is a real-time continuous system that operates in the time domain, and requires a different continuous control for each of task execution.

For example, for a given execution of the disassembling of the modeled environment the following atomic tasks have to be done: teleoperation (planned and unplanned), move from one point to another, open or close gripper to pick up or put down the tool, and turn on or turn off the tool. Each atomic task has different functionality requirements, and has to be accurately synchronized in time. In addition, the controller must be consistent with and easily extendable to any other possible operational atomic tasks.

One of the techniques to model and to control continuous systems, particularly robots is to approximate them with sample-data systems with sufficiently high sampling

rate. This approximation gives the alternative to model and control continuous system using not only analog, but also, digital computers. Further on, a sample-data control with sufficiently high sampling rate will be assumed as a continuous control of the continuous system.

Because most of the conventional software for robotic control (Simulink, ControlShell) includes finite-state machine (FSM) functions, it has been selected as a high level discrete controller between the task planner and the lower level continuous controllers of the robot. Also, the FSM is responsible for the human operator and task planner interface.

4.2 Basic model of the FSM [14]

Every system representable by the basic FSM model is assumed to be controlled by an independent *synchronizing source*, in the following fashion: The system variables are not measured continuously, but only at the discrete instants of time at which a certain specified event, called *synchronizing signal*, is provided by the source. These instants of time are called *sampling times*, the v the sampling time being denoted by t_v ($v = 1, 2, \dots$). An additional assumption is that the finite-state machine at any sampling time t_v is independent of the interval between t_v and the previous sampling time t_{v-1} . Thus, the true independent quantity, against which every system variable is measured, is not time, but the ordinal number associated with the sampling times. Systems that conform to this time-discreteness assumption are said to be *synchronous*.

The next assumption to be made for the basic FSM is that each variable can assume only a finite number of distinct values. The set of values, which the variable v can assume is called the v *alphabet* and denoted by V ; each element in V is called a v *symbol*.

Let A and B be two finite sets. The product set $A \otimes B$ (read " A cross B ") is defined as $A \otimes B = \{(a, b) \mid a \in A, b \in B\}$. Pair (a, b) is called an *ordering pair*, where the term "ordered" implies a distinction between the first and second element. If $A_1, A_2, \dots$,

A_n sets, then the *product set* is defined as

$A_1 \otimes A_2 \otimes \dots \otimes A_n = \{(a_1, a_2, \dots, a_n) \mid a_1 \in A_1, a_2 \in A_2, \dots, a_n \in A_n\}$. The symbol $(a_1, a_2, \dots, a_n)$ is called an *ordered n-tuple*.

If the excitation variables of a given system are $x_v^{(1)}, x_v^{(2)}, \dots, x_v^{(u)}$, the *input alphabet* or *stimulus* of this system is denoted by X and defined by

$$X = X^{(1)} \otimes X^{(2)} \otimes \dots \otimes X^{(u)} \quad (4.1)$$

Where $X^{(i)}, i=1, 2, \dots, u$, is the $x^{(i)}$ alphabet. Similarly, if the response variables of the system are $z_v^{(1)}, z_v^{(2)}, \dots, z_v^{(w)}$, the *output alphabet* or *actions* of the system is denoted by Z and defined by

$$Z = Z^{(1)} \otimes Z^{(2)} \otimes \dots \otimes Z^{(w)} \quad (4.2)$$

Where $Z^{(i)}, i=1, 2, \dots, w$, is the $z^{(i)}$ alphabet.

The concept of the *state* can be accurately defined through the role played by it in the basic FSM. This role can be described through the following two statements: (1) the output symbol and the present sampling time are uniquely determined by the input symbol and state at the present sampling time; (2) the state at the next sampling time is uniquely determined by the input symbol and state at the present sampling time.

Definition.

A *Finite State Machine* is a synchronous system with a finite input alphabet $X = \{\xi_1, \xi_2, \dots, \xi_p\}$, a finite output alphabet $Z = \{\zeta_1, \zeta_2, \dots, \zeta_q\}$, a finite state set $S = \{\sigma_1, \sigma_2, \dots, \sigma_n\}$, and a pair of *characterizing functions* f_z and f_s , giving by

$$z_v = f_z(x_v, s_v) \quad (4.3)$$

$$s_{v+1} = f_s(x_v, s_v) \quad (4.4)$$

where x_v , z_v , and s_v are, respectively, the input symbol, output symbol, and state of FSM at time t_v ($v = 1, 2, \dots$). f_s is called the *transition function* mapping each current state and input to the next state, and f_z is called *output function* mapping each current state and input to an output.

The verbal description of the system can be formalized in a diagram as an alternative form for the characterizing functions of the FSM. The transition diagram is a structure composed of vertices, drawn as circles or rectangles, and oriented branches, drawn as lines between pairs of vertices, with arrow signs pointing from one vertex to the other. A transition diagram describing an n -state machine contains n vertices, each vertex representing a different state; the label identifies the state represented by a vertex. The oriented branches are drawn and labeled according to the following rule:

let

$$X_{ij} = \{\xi_{k_1}, \xi_{k_2}, \dots, \xi_{k_r}\}$$

to be the set of x_v values for which $f_s(x_v, \sigma_i) = \sigma_j$, and let $f_z(\xi_{kh}, \sigma_i) = \zeta_{lh}$ for $h = 1, 2, \dots, r$. If X_{ij} is nonempty, a branch is drawn from the vertex labeled σ_i to the vertex labeled σ_j is shown in the Figure 4.1 Each term $(\xi_{k_h} / \zeta_{l_h})$ contained in a branch label is called an input-output pair or stimulus and system action.

A branch touching any given state, say σ_i , may be *converging* branch of σ_i , if it points toward σ_i from another state or a *diverging* branch of σ_i , if it points from σ_i toward another state, or a *reflecting* branch of σ_i if it loops around σ_i . [Figure 4.2]

The FSM has a hierarchical architecture; it can have different *submachines (levels)*. Submachines are basically transparent states that enclose other states. Transitions from a submachine apply to any state enclosed by the subsystem.

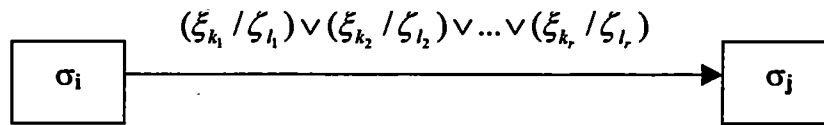


Figure 4.1: Branch labeling

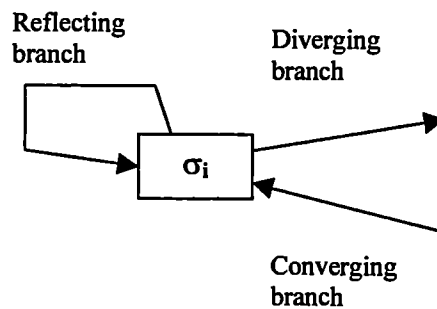


Figure 4.2: Types of branches

The diagram transforms an abstract model into a physical picture that enhances the investigator's intuition and enables him to "visualize" various processes and properties, which would otherwise remain a series of dry mathematical abstractions. As example, Figure 4.3 illustrates the simple transition diagram of finite state machine of the teleoperation and autonomous control telerobotic system. In this diagram the transition pair is stimulus expression and function that executes in response to the stimulus.

The main advantage in employing the FSM for the representation of a given system is that the prediction of the system response does not require any data concerning past performance of the system. To predict the response at any given time, it is sufficient to know the stimulus and the state at that time.

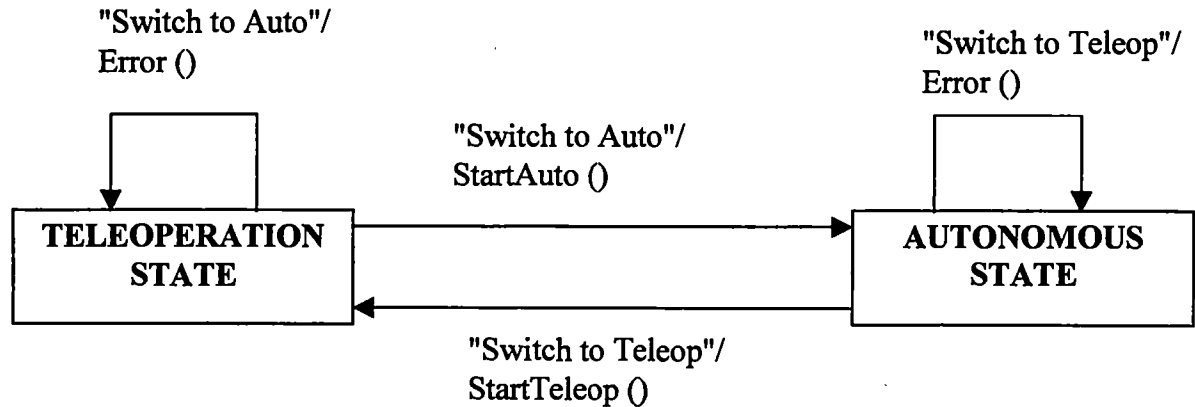


Figure 4.3: Transition Diagram of FSM for the Teleoperation/Autonomous System

4.3 High-level controller for the hybrid telerobotic system

4.3.1 Main FSM

Basic components of the telerobotic/autonomous system being considered here include human operator, computer, minimaster, and robot. During the task execution, the human operator and computer will switch the control of the robot according to some logic. The transition between the TELEOPERATION and the AUTOEXECUTION states has to be synchronized and smooth for effective operation. Therefore one additional state has to be considered: the state when robot and minimaster are immobilized. This state is an IDLE state. These three states are parts of the ON level of the execution, when all components of the system are in operation. In an emergency situation, the system can be switched off at any state enclosed in this level. "Emergency Stop" transition of the system will turn all the components off and take the FSM to the ESTOP state. Each time when transition to the other state cannot be done the system reflects back to the original state. The Main FSM of the telerobotic/autonomous system controller is showed at Figure 4.4.

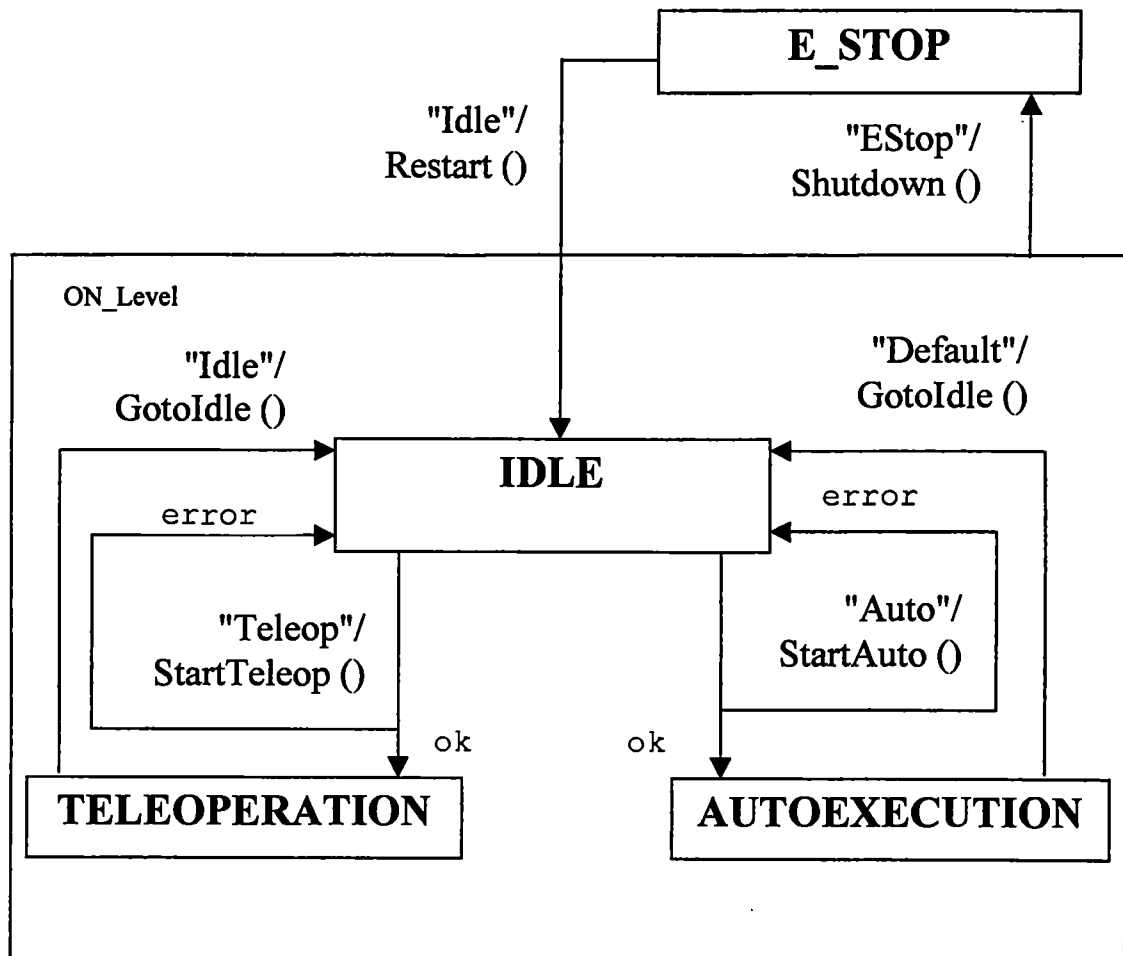


Figure 4.4: Main FSM

When the telerobotic system is turned on the controller automatically goes to the IDLE state and waits for the future commands from the human operator. If human operator selects "Teleop", the system switches to the TELEOPERATION state, and the left subsystem of the hybrid telerobotic system is active [Figure 3.2].

After completion of teleoperation, the human operator can switch the telerobotic system back to IDLE with "Idle" stimulus. In IDLE, robot and minimaster, and all of the components of the continuous controller are on, but they are in an inactive (pause) mode in the time domain. Therefore, the operator can decide what to do and send the next stimulus without turning off the system. From IDLE state the human operator can switch system back to the teleoperation or switch control to the Task Planner by selecting "Auto". The "Auto" stimulus allows the Task Planner the full control of the system in the AUTOEXECUTION submachine. When this submachine reaches the exit state [see chapter 4.3.2] the system automatically will be sent back to the IDLE state.

4.3.2 Autoexecution submachine

Typically each submachine contains an entry state and any number of exit states. The transition to the submachine takes the system directly to an entry state. The transition out of the submachine can take place only from one of the exit states. Block START represents the entry state of the AUTOEXECUTION submachine [Figure 4.5].

The two exit states are END and ERROR. After the entry state, the system on Default goes to PAUSE. The Null () function stipulates that no action has to be done. The PAUSE state represents the same inactive (pause) mode of the telerobotic system in time domain [see chapter 4.3.3], but it is not the IDLE state of the FSM. In PAUSE the human operator still has control of the system. If "Teleop" is selected by the human operator, the FSM goes to the UNPLANNED TELEOPERATION state and the system switches to teleoperation. Even though the UNPLANNED TELEOPERATION and TELEOPERATION are represented by the same teleoperation mode of the continuous system; these two states are different also by state definition. After teleoperation, the

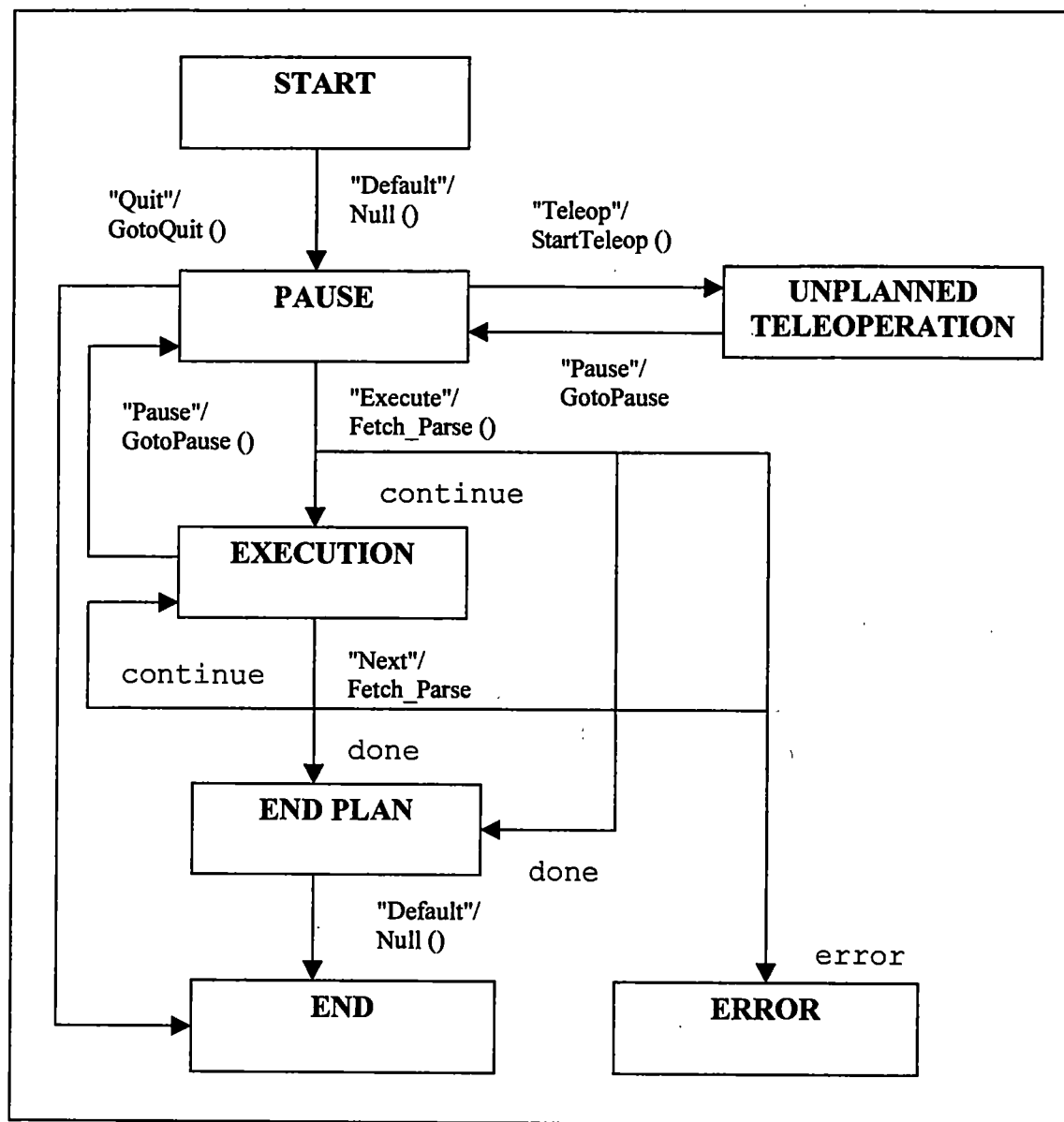


Figure 4.5: Autoexecution submachine

system can be switched back to the pause mode, which is defined by the PAUSE state of the FSM. When the AUTOEXECUTION submachine goes to the EXECUTE state, the Task Planner takes over the control of the execution. The Task Planner represents a text file (taskplan.txt) containing a set of tasks (atomic tasks). Each line in the Task Plan file is a different command. The submachine is required to open the file, if it is not yet open, to read the next line of file, to select the operating mode execution for each particular atomic task, and to send appropriate commands to the continuous controller to activate the mode. It is done by function `Fetch_Parse ()`, after the Task Planner sends the "Execute" stimulus to the FSM. If the file is empty or cannot be opened, `error` return of the function will place the FSM in the ERROR state. If the last line of the file is executed, the `done` return will send the system to the END PLAN state, and automatically to the END state of the FSM, and out of the AUTOEXECUTION submachine (END and ERROR are exit states). Return `continue` will send the FSM to the EXECUTION state, and asks the continuous system to complete the atomic task. When an atomic task is completed, the stimulus "Next" will order the `Fetch_Parse ()` to complete the next command from the Task Planner, and so on. A more detailed explanation of the mode selection is done in chapter 4.3.3. The construction of the `Fetch_Parse ()` function is described in chapter 5.3.4. During the autonomous task execution, the human operator acts as a supervisor, who can interrupt in the execution. The "Pause" stimulus will send the FSM to the PAUSE state, and then if human operator decides not to continue the autoexecution, the stimulus "Quit" will ask the system to close the `Fetch_Parse ()` file, switch to the END state go directly to the IDLE state in the Main FSM.

4.3.3 Mode changing Hybrid Controller Interface.

Because of its complexity, the telerobotic hybrid system can have many different operating modes. By defining different modes, the run-time continuous system can switch between different control paths easily and quickly at run time without halting execution. Each mode corresponds to a set of active components of the system, and

requires different system functionality. With functionalities defined, mode switching is performed under the direction of the high –level controller for the hybrid system, in our case the FSM. Mode changing operation is an interface between the discrete event and the continuous system characteristics of the hybrid model [Chapter 3]. The interface must be synchronized and smooth so that run time mode switches can occur quickly and efficiently providing a "bumpless transfer" between operating modes. Later in chapter 5 it will be demonstrated how this interface could be done in the ControlShell implementation.

FSM components interact with the robot controller changing the controller operating modes. The robot controller, in turn, feeds sensor data into the FSM components to allow them to detect changes. For pipe cutting teleoperation/autonomous operations, the FSM does the following:

- Processes the Task Planner commands: do planned teleoperation, move from point to point, pick up or put down tool, complete the cut.
- Responds to "emergency" situations to turn off the system.
- "Freezes" the system during IDLE or PAUSE states.
- Asks the system to do teleoperation during TELEOPERATION and UNPLANNED TELEOPERATION states.

Six different operating modes are required:

- *Off mode* to switch the telerobotic system off.
- *Teleoperation mode* to perform any teleoperation execution.
- *Trajectory mode* to perform “move from point to point” command of the Task Planner.
- *Gripper mode* to open or close gripper during the “pick up or put down tool” operation.
- *Tool mode* to turn on, or turn off, the tool to “complete the cut”.
- *Pause mode* to “freeze” the system during the switching between states operations.

CHAPTER 4. FSM AS A HIGH LEVEL CONTROLLER OF THE HYBRID TELEROBOTIC SYSTEM

Each of the five modes will be turned on only if the corresponding task has to be done. In “emergency” situations, all five modes will be deactivated. Figure 4.6 demonstrates the interaction between the high-level controller (FSM) and the continuous controller of the hybrid telerobotics system. Each step, only one of five modes: teleoperation, trajectory, gripper, tool or pause can be activated. When the continuous controller finishes atomic task execution the signal "Next" is sent back to the FSM. Figures 4.7 and 4.8 show the complete FSM diagram and its interface with continuous controller in ControlShell expressions. Complete relations between states, state transitions, modes, and system status are shown in Table 4.1.

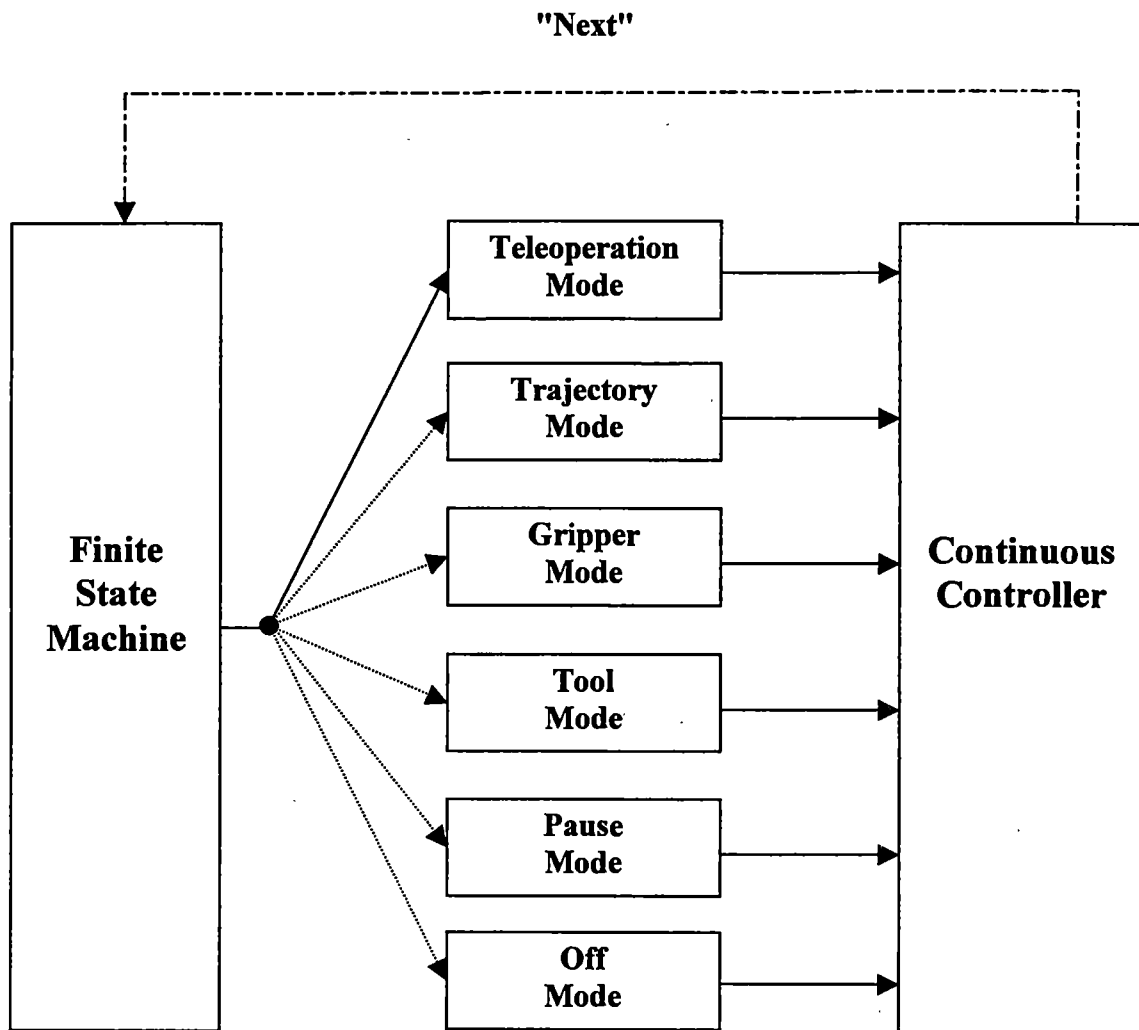


Figure 4.6: Mode Interface of the Hybrid System

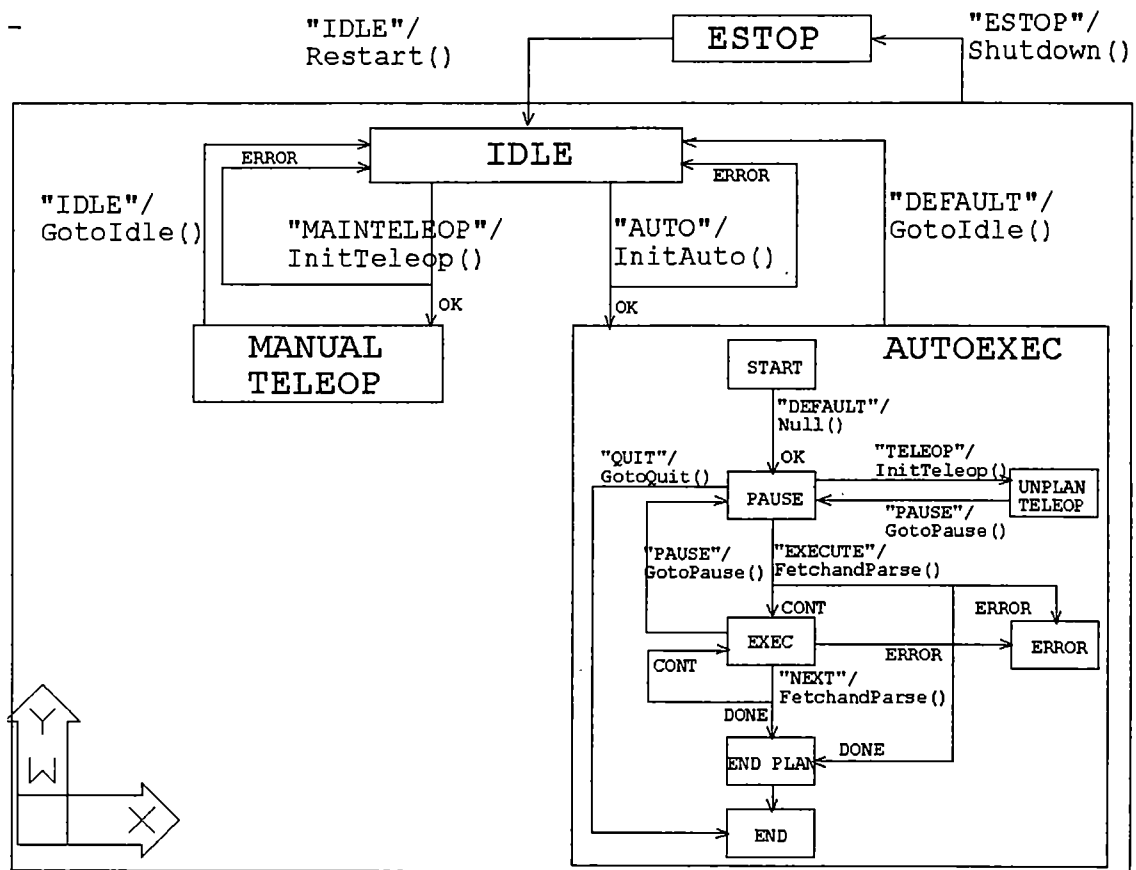


Figure 4.7: FSM diagram Part 1

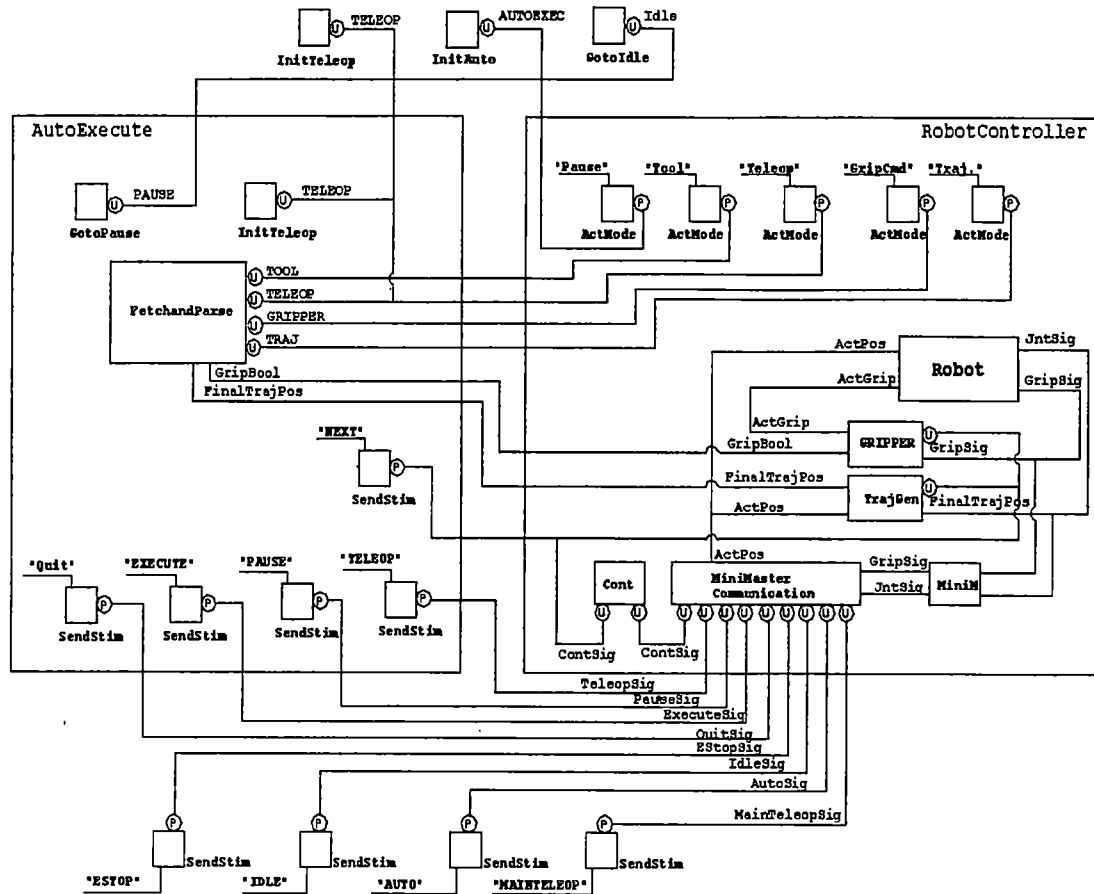


Figure 4.8: FSM diagram Part 2

Table 4.1: States, state transitions, modes, and system status.

FSM state	timulus	FSM Transitions	Controller mode	System status
E_STOP			Off	off
	Idle	go to IDLE	change to Pause	restart
IDLE			Pause	on/inactive
	Teleop	OK: go to TELEOP ERROR: back to IDLE	change to Teleop Pause	start teleop on/inactive
	Auto	OK: go to AUTOEXEC ERROR: back to IDLE	Pause Pause	on/inactive on/inactive
TELEOP			Teleoperation	teleoperation
	Idle	go to IDLE	change to Pause	stop teleoperation
AUTOEXEC:				
START	Default	go to PAUSE	Pause	on/inactive
PAUSE			Pause	on/inactive
	Execute	cont : go to EXECUTE done : go to END PLAN	change to Traj. change to Gripper change to Tool change to Teleop	move from point to point open/close gripper on/off tool start teleoperation
		error: go to ERROR	Pause	on/inactive
	Teleop	go to UNPL.TELEOP	change to Teleop	start teleop
	Quit	go to END	Pause	on/inactive
UNPLANNED TELEOP			Teleoperation	teleoperation
	Pause	go to PAUSE	change to Pause	deactivate
EXECUTION	Next	cont : go to EXECUTE done : go to END PLAN	change to Traj. change to Gripper change to Tool change to Teleop	move from point to point open/close gripper on/off tool start teleoperation
		error: go to ERROR	any	any
	Pause	go to PAUSE	change to Pause	deactivate
END PLAN	Default	go to END	any	any
END	Idle	go to IDLE	change to Pause	deactivate
ERROR	Idle	go to IDLE	change to Pause	deactivate
any	Estop	go to E_STOP	change to Off	turn off

Chapter 5

ControlShell implementation

5.1 Introduction

"The ControlShell software design system from Real-Time Innovations, Inc. is a development tool set for designing and implementing real-time applications.

ControlShell's graphical component-based design tools, component repository management tools, code generators, run-time execution engines, and run-time debugging tools allow building and maintaining the real-time applications quickly and easily.

ControlShell is an end-to-end software development system. ControlShell's design methodology is targeted specifically at the class of real-time systems that include sampled-data and event-driven subsystems. These include most electro-mechanical systems, particularly robots. Because of its practical focus – rather than a theoretical software approach – ControlShell is able to provide specific tools and services that directly translate designs into efficient, run-time code.

ControlShell components are hierarchical. At the lowest level, they are implemented in C++ code. At the higher level, they are composed – graphically – by combining other components. Whereas the low-level components typically provide simple functions, higher-level components can model physical objects by combining functional components that provide capability with finite state machine components" [13].

A hybrid telerobotic system is a real-time application system that requires a multi-threading support from the operating system. In the past the most suitable operating system for robotic models was VxWorks OS. But VxWorks is an expensive operating system. Just recently two newer operating systems have been considered: QNX (hard real-time) and LINUX (soft real-time). In 1999 Real-Time Innovations, Inc. introduced a new version of the ControlShell application executable under LINUX. To evaluate the ControlShell performance under LINUX, ControlShell has been selected for the hybrid telerobotic system controller.

5.2 ControlShell Structure [13]

To control the hybrid real-time system two different controls have to be built. First is the continuous low-level controller for the robot that uses sampled-data processing. This controller has to run periodically at some sufficiently high sampling rate that provides essentially continuous control. Second is the discrete event controller of human and computer cooperation that uses event-driven processing. This controller reacts to asynchronous commands and processes logic. ControlShell provides specific tools to design and build both types of controllers. Data-flow diagrams are used to build the sample-data processing parts of the control system and state-transition diagrams to build the event-driven processing parts of the system. The coupling of event and continuous processing defines the hybrid system.

An *application* (App) represents a top-level ControlShell object that translates directly into a running execution. The role of an application is to map operating-system services to a composite diagram. After the mapping, the application can be built and run.

A *composite object group* (COG) is a graphically built composite component that contains both event-driven and cyclic elements with a set of defined connectors that interface with other COG or FSM components.

Finite state machine (FSM) is also a composite component that is built graphically. Its contents form a state-transition diagram that represents an event-driven program. The basic elements within a state-transition diagram are *states*, *stimuli*, and *state-transition components* (STC).

STC is one of three primitive components in ControlShell. STC provides actions taken by a finite state machine in response to a stimulus. Other two primitives are *data flow component* (DFC) and *atomic component* (ATC). DFC is a building block for the sampled-data part of the system that executes routines run at every sample period. An ATC may be inserted into a COG or FSM diagram. ATC provides generic utilities or functions or services to other components in the same diagram.

Connections between components in ControlShell take one of three forms. *Pin* connections are inputs and outputs of data from components. *Bubble* connections provide utility of functions from one method to another. *Interfaces* allow pins and bubbles to be bundled in a unified connection.

When the primitive STC, DFC or ATC is inserted in the ControlShell diagram and all the connectors are selected, the C++ code for the block can be generated. The code can be modifying as the operator requests.

Another important feature of ControlShell is the ability to define *operating modes*. Each operating mode can represent in a ControlShell diagram as a list of active DFC, COG, and FSM components. An active DFC or COG means that it will run at every sample period; an active FSM means that it is processing stimuli.

ControlShell uses a *habitat* object to manage the sample rate and to trigger sampled-data processing. A habitat is the execution environment for the system. Different habitats are used for different rates. *Clock* is a primitive component that represents a clock – a timing source – for the sampled-data part of an application. Its role is to convert timing services provided by the operating system or hardware into a trigger for habitat object. Relationship of ControlShell elements is described in Figure 5.1

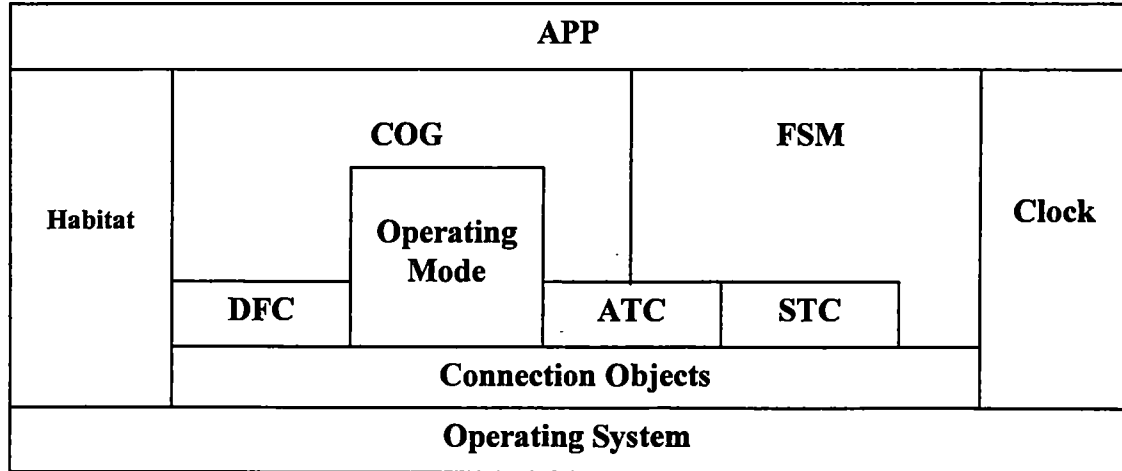


Figure 5.1: ControlShell Interface Structure

5.3 Controller Components.

5.3.1 Controller App.

The ControlShell application of the hybrid telerobotic system controller is shown in Figure 5.2. There is one COG Controller, which contains all the components of the controller. In the habitat and the clock the SampleRate is set at 60 Hz, all other parameters are set on default.

The Controller COG includes two primary components of the hybrid telerobotic system – the FSM block DESController, which models the discrete event subsystem and the COG block RobotController, which represents the continuous subsystem [Figure 5.3]. The interaction between these two blocks simulates the hybrid telerobotic system interface.

CHAPTER 5. CONTROLSHELL IMPLEMENTATION

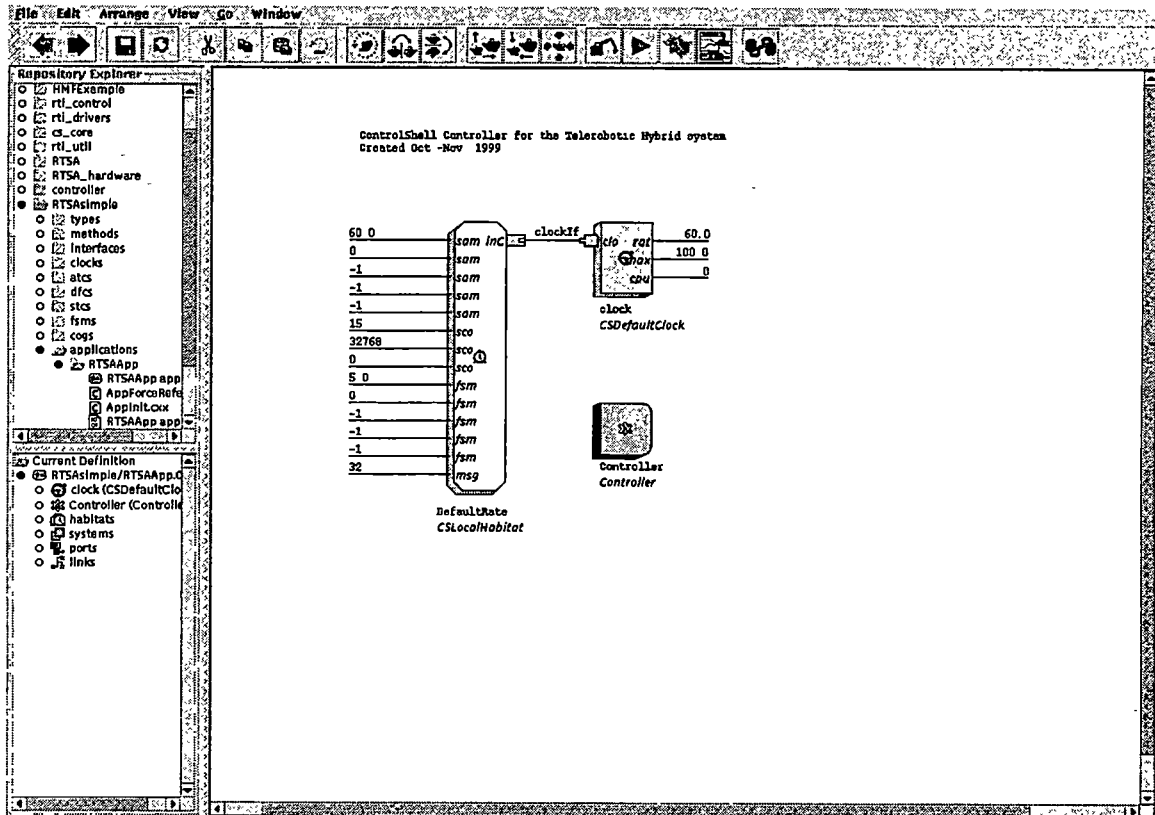


Figure 5.2: ControlShell application (APP) of the controller.

CHAPTER 5. CONTROLSHELL IMPLEMENTATION

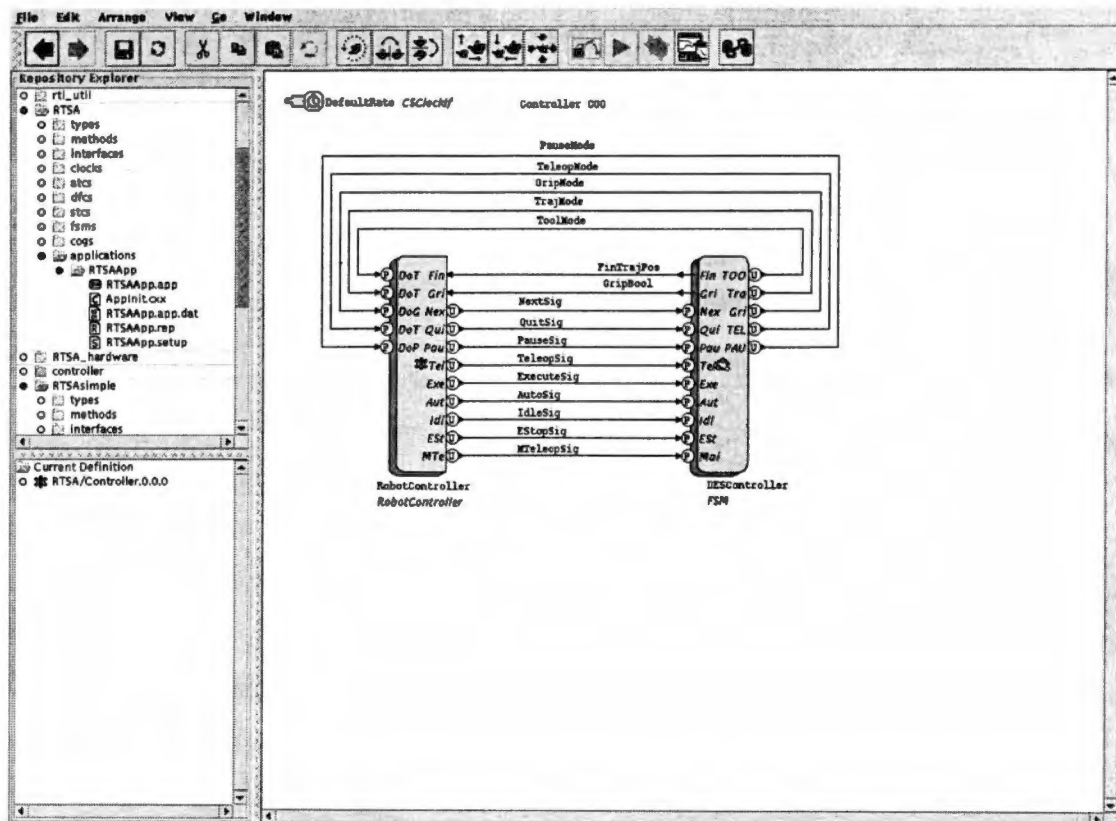


Figure 5.3: Controller COG

5.3.2 DESController

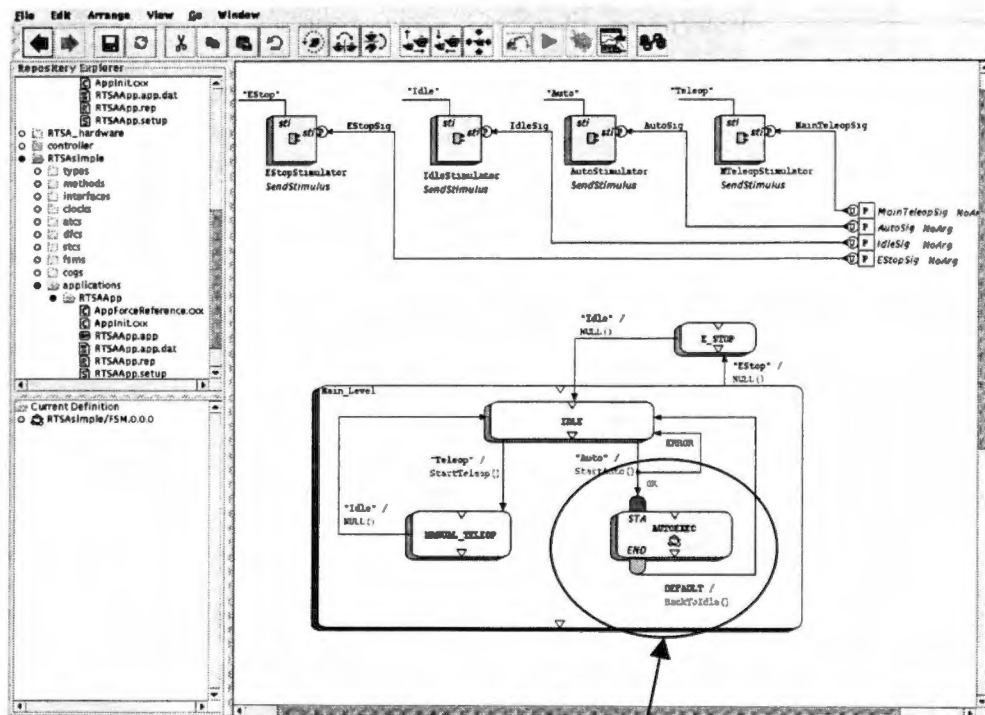
DESController implements the Main FSM of the hybrid telerobotic system [Chapter 4.3.1]. In this finite-state diagram the same labels for the stimulus, states, and state-transition functions are used. The diagram in Figure 5.4 (a) is the complete replica of the FSM diagram in Figure 4.4.

The entire transition stimuli in this diagram are sent by the teleoperator through the minimaster communication interface [Chapter 5.5.3]. The teleoperator can send one of four commands: E_Stop, Idle, Auto, or Teleop. Each command is transmitted to the build-in atomic component SendStimulus ACT, which converts this command to the corresponding stimulus. Each stimulus triggers the actual state-transition. Each state-transition function is represented as a STC block below the state-transition graph [Figure 5.4 (b)]. Each of the STC corresponds to a C++ class. When the STC block is inset in the diagram, and all connections are specified, the C++ code is generated automatically. This code can be used as framework for the complete code required for the function.

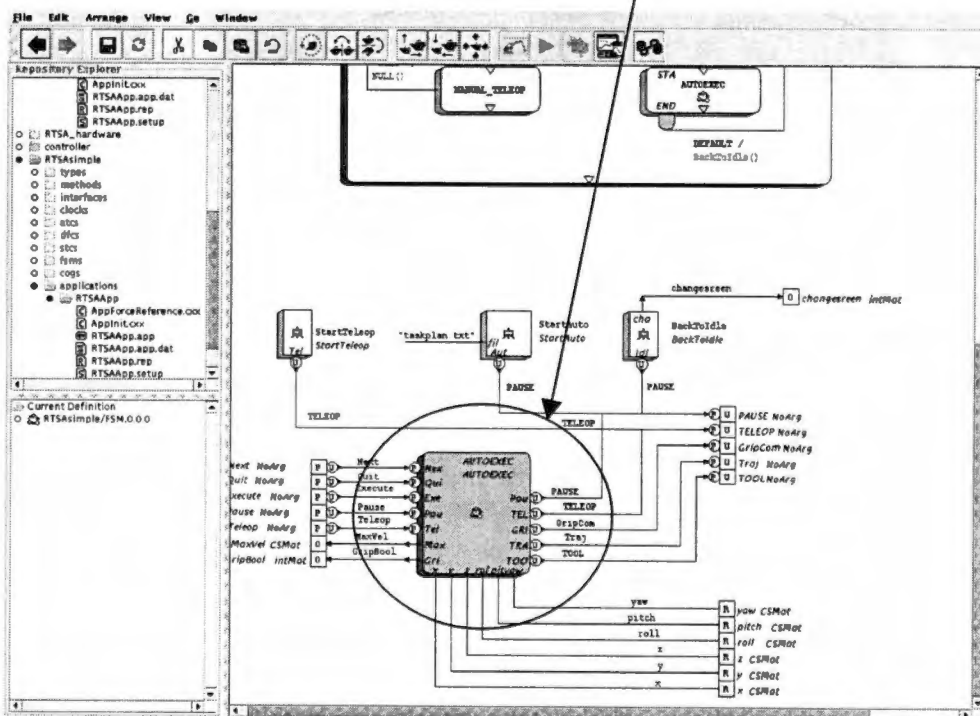
At the presented hardware structure of the hybrid telerobotic system, the C-30 and minimaster are independent blocks of the controller. These will be turned off automatically if the communication cannot be established or in "emergency" situations. The arrangement of the starting or restarting the system from the FSM has not been investigated yet. The E_STOP state is placed in the diagram just for future references.

The IDLE state is the initial state of the system. The transition "Teleop" triggers the STC StartTeleop (), which activates the Teleop Mode of the continuous RobotController, and sends the "Start Teleoperation" message to the teleoperator. The transition "Idle" takes the system back to the IDLE state. The STC component Idle () sends the Bubble PAUSE to the continuous RobotController and "System back to Idle" message to the operator. When teleoperator selects to switch the control to the Task Planner and to start the autonomous task execution the stimulus "Auto" sends the system to the AUTOEXEC submachine. StartAuto () checks the existence of the taskplan.txt file, and makes sure that

CHAPTER 5. CONTROL SHELL IMPLEMENTATION



(a): state-transition diagram



(b): connections

Figure 5.4: Main FSM

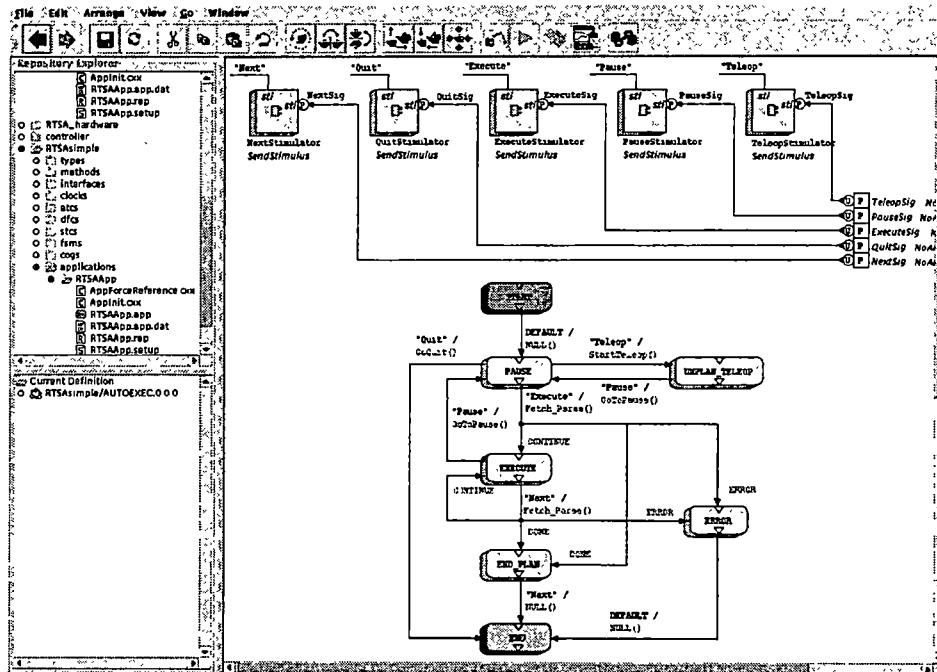
the system is still in Pause Mode. If the taskplan.txt does not exist or cannot be open the transition to AUTOEXEC is not carrier out. The ERROR return of the function StartAuto () takes the system back to the IDLE state. The OK return sends the system to the AUTOEXEC subsystem. The transition out of the AUTOEXEC submachine to the IDLE state is automatic. In this case the STC BackToIdle () activates the Pause Mode, and sends the "change screen" request to the Communication block, which is responsible about the minimaster menu screens [see chapter 5.5.3].

5.3.3 AUTOEXEC composite state

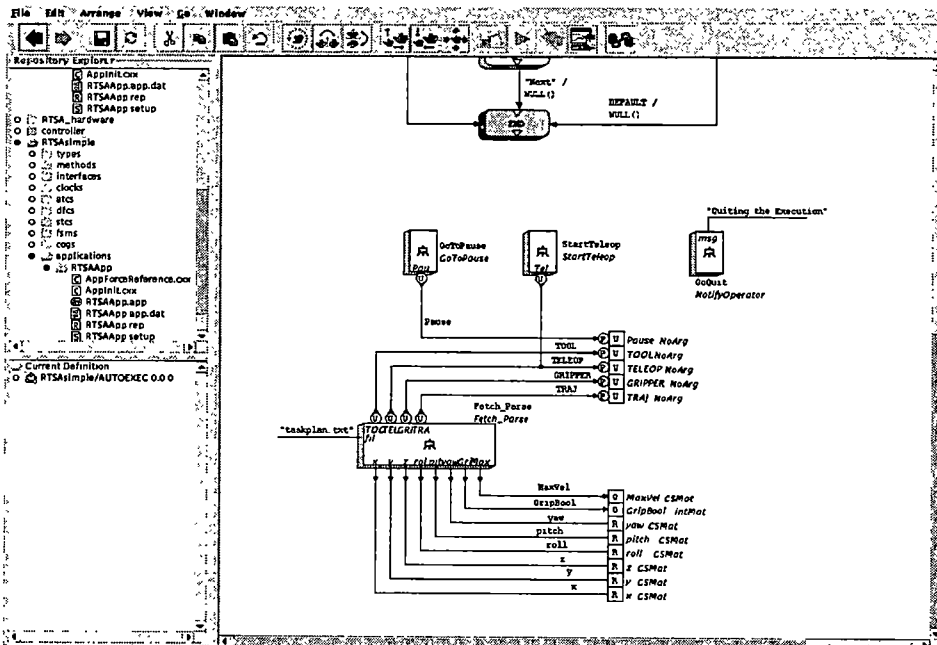
"Any finite-state diagram in ControlShell may be used directly in another diagram as a *composite state*, allowing the design of hierarchical diagrams. To reduce visual clutter, ControlShell represents a composite state using two distinct bodies within the parent finite-state diagram: a special state body and an FSM component body. The special state body allows transition attachments while the component body allows link connections for connection-type objects – pins, bubbles, and interfaces. In addition to reducing visual clutter, this separation clearly delineates the two roles played by the finite-state diagram. The state transitions define the logical program. The component connections determine how to perform the actions or computations and how to route signals and method calls into and out of the diagram." [13]

Two bodies of the AUTOEXEC submachine are shown in Figure 5.5 (a) and (b). The body in Figure 5.4 (a) corresponds to the state-transition diagram in Figure 5.5 (a). The body in Figure 5.4 (b) corresponds to the component connections in Figure 5.5 (b). The state-transition diagram is the identical to the submachine in Figure 4.5 with exactly the same labels. The teleoperator and the continuous controller do transitions in this submachine. Five different stimulus can be sent from five corresponding build-in SendStimulus atomic components: "Quit", "Execute", "Pause", "Teleop", and "Next".

CHAPTER 5. CONTROL SHELL IMPLEMENTATION



(a): state-transition diagram



(b): connections

Figure 5.5: AUTOEXEC

The first four commands come from the teleoperator. The RobotController sends the "Next" signal, when it completes each atomic task. For "Teleop" command the same STC block StartTeleop () is used as for the same operation in diagram Figure 5.4 (a). Stimulus "Quit" sends the system out of the submachine, and notifies teleoperator about quitting the autonomous execution, using the ControlShell build-in NotifyOperator STC block. GoToPause () activates the Pause Mode of the robot continuous controller, and lets teleoperator know about it through the ControlShell text interface [Chapter 5.5.2].

5.3.4 Fetching and parsing the Task Planner instructions.

After the Task Planner of the RTSA system completes the list of execution instructions, the list is downloaded to the controller computer as a text file (taskplan.txt). Figure 5.6 shows an example of this file.

The Fetch_Parse () state transition component is responsible for opening the taskplan.txt file on its first recall, retrieving the first unexecuted action from the file, calling the appropriate execution mode of the RobotController [TELEOP, TOOL, GRIPPER, or TRAJ], and sending values of the gripper in the Gripper Mode or final coordinates of the manipulator for the MOVE action. The function has three possible returns: CONTINUE, when Fetch_Parse () fetches the taskplan.txt line and parses the mode, gripper Boolean signal or position values to the RobotController; DONE, when the parsing line is the last line of the text; and ERROR, when an error occurs during the reading of the file.

The CONTINUE return takes the diagram back to the EXECUTE state. DONE and ERROR send the system out of the AUTOEXEC submachine [chapter 4.3.2] The C++ code of the Fetch_Parse block is shown in Appendix 1. In the PAUSE state teleoperator has to check if some replanning is required. During the replanning operation the teleoperator can access the task plan file and modify it if necessary. The replanning operation is still in discussion.

1. MOVE	x 1.000 y 1.000 z 1.000 roll 40 pitch 40 yaw 40 Vmax 0.100
2. MOVE	x 3.000 y 3.000 z 3.000 roll 40 pitch 40 yaw 40 Vmax 0.100
3. GRIPPER	open
4. MOVE	x 3.000 y 3.000 z 8.000 roll 40 pitch 40 yaw 40 Vmax 0.050
5. GRIPPER	close
6. MOVE	x 3.000 y 3.000 z 3.000 roll 40 pitch 40 yaw 40 Vmax 0.100
7. MOVE	x 10.000 y 10.000 z 10.000 roll 50 pitch 40 yaw 50 Vmax 0.500
8. TOOL	saw on
9. MOVE	x 10.000 y 40.000 z 10.000 roll 50 pitch 40 yaw 50 Vmax 0.050
10. TOOL	saw off
11. MOVE	x 10.000 y 10.000 z 10.000 roll 50 pitch 40 yaw 50 Vmax 0.100
12. TELEOP	assist method 3 N1 N3 N4
13. TELEOP	manual

Figure 5.6: Example of the Task Planner text file

5.3.5 RobotController COG

The hybrid telerobotic system continuous controller is shown in Figure 5.7 as COG RobotController. The teleoperation mode is selected.

The RobotController COG is the data flow diagram, which includes the different components of the robot control architecture, and has various capabilities for modes of operation of the manipulator. The COG contains three COGs and three data flow components DFCs. The explanation of the RobotCC and MiniMasterCommunication COGs will be done in chapters 5.4 and 5.5 respectively. The TrajGen COG block builds up an atomic task execution to move from point to point. This COG gets the desired final Cartesian coordinates of the manipulator move from the Task Planner file, and the feedback information of actual joint positions from the manipulator, converts its to the Cartesian coordinates, constructs the desired Cartesian trajectory with some velocity profile, converts it point by point to the joint space, and, finally, sends the desired joints commands to the low-level robot controller. When the execution of the path is completed, the COG calls the function to send the "Next" stimulus.

The Gripper DFC block is responsible of the gripper commands execution. It receives the Boolean signal from the Task Planner – 1 to open the gripper, and 0 to close the gripper, and the feedback open percentage information from the manipulator, plans the balanced opening or closing execution. When the atomic task is completed the signal "Next" is sent to the AUTOEXEC finite-state machine.

The MiniMasterCommunication and the RobotCC components are always activated during operation of the system. Therefore to activate Teleoperation mode two additional component are required - Minimaster and OnOff. These DFCs are just pass through components.

The system has five implemented modes: Pause, Teleoperation, Gripper, Trajectory, and Tool, and two predefined modes of the ControlShell: Off and Default. Chapter 4.3.3 explains how these modes are selected. In the ControlShell modes can be activated using the ActivateMode build-in atomic components. Each time just one mode is operating. Each operating mode is represented in the diagram as a list of active DFCs, and COGs. Following modes are provided:

- OFF – This means that all blocks will not run anything at each sample period (predefined mode of the ControlShell);
- DEFAULT – MiniMasterCommunication and RobotCC components are active (predefined and modified mode of the ControlShell);
- PAUSE - MiniMasterCommunication and RobotCC components are active;
- TELEOP – MiniMasterCommunication, RobotCC, Minimaster, and OnOff components are active [Figure 5.7];
- GRIPPER – MiniMasterCommunication, RobotCC, and Gripper components are active;
- TRAJ – MiniMasterCommunication, RobotCC, and TrajGen components are active;
- TOOL – MiniMasterCommunication, RobotCC, and Tool components are active.

CHAPTER 5. CONTROLSHELL IMPLEMENTATION

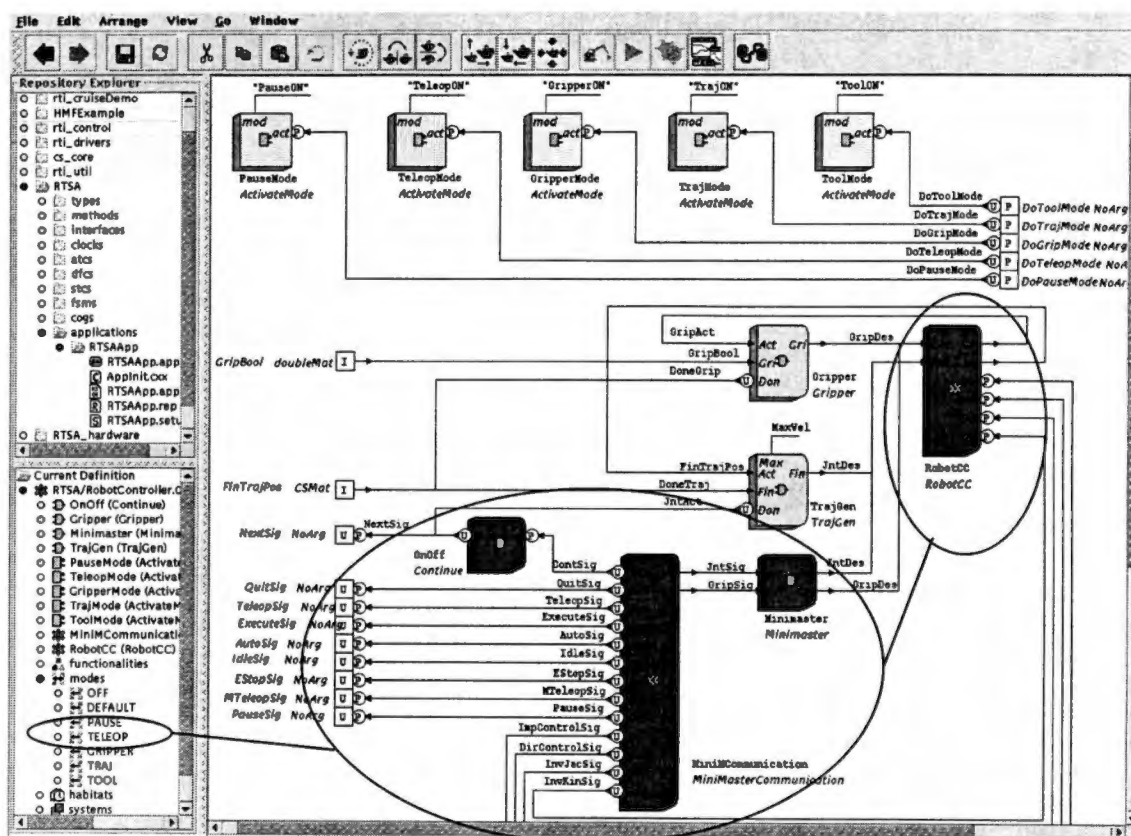


Figure 5.7: RobotController COG: Teleop mode selected

5.4 Robot closed loop control

RobotCC COG is shown in Figures 5.8 and 5.9. The RobotCC COG is the robot low-level close loop controller. Four different continuous closed loop control schemes are considered: direct inverse kinematics control, direct inverse Jacobean control, impedance inverse kinematics control, and impedance inverse Jacobean control. From the beginning of the task execution the operator selects one of these four controls and pushes the corresponding button on the master console [Chapter 5.5.3]. That button sends the command to activate one of the following modes of the control execution: “DirControl”, “ImpControl”, “InvKin”, and “InvJac” by using build-in ActivateMode components in the robot closed loop control diagram. Each of these modes requires a different set of the components to be running. ControlShell allows placement of all components needed the same COG then map modes to corresponding sets of components. For example, if direct control is selected inside RobotCC COG following blocks will be activated: ForKin, which converts the input joint angles to the Cartesian coordinates; Diff differential block, which outputs the velocities; AssistFunctions, which implies the assist functions strategy during teleoperation; Robot, which is the controller interface with the C-30 card; and DirControl, the direct low-level robot controller [Figure 5.8] Analogous, Figure 5.9 represents the selection of the impedance control scheme. In the DirControl block or in ImpControl block two controlled modes are possible: inverse kinematics [Figure 5.10 (a) or Figure 5.11 (a)], and inverse Jacobian [Figure 5.10 (b) or Figure 5.11 (b)]. Selections of different control scheme for each experiment will demonstrate the best low-level controller for the hybrid telerobotic system. Impedance control is introduced as our future research area and will be selected when a force feedback sensor will be available.

Robot component of the RobotCC COG is the communication with the robot hardware block [Figure 5.12]. DFC vmeC30 is the communication block between the Schilling C-30 and ControlShell.

CHAPTER 5. CONTROL SHELL IMPLEMENTATION

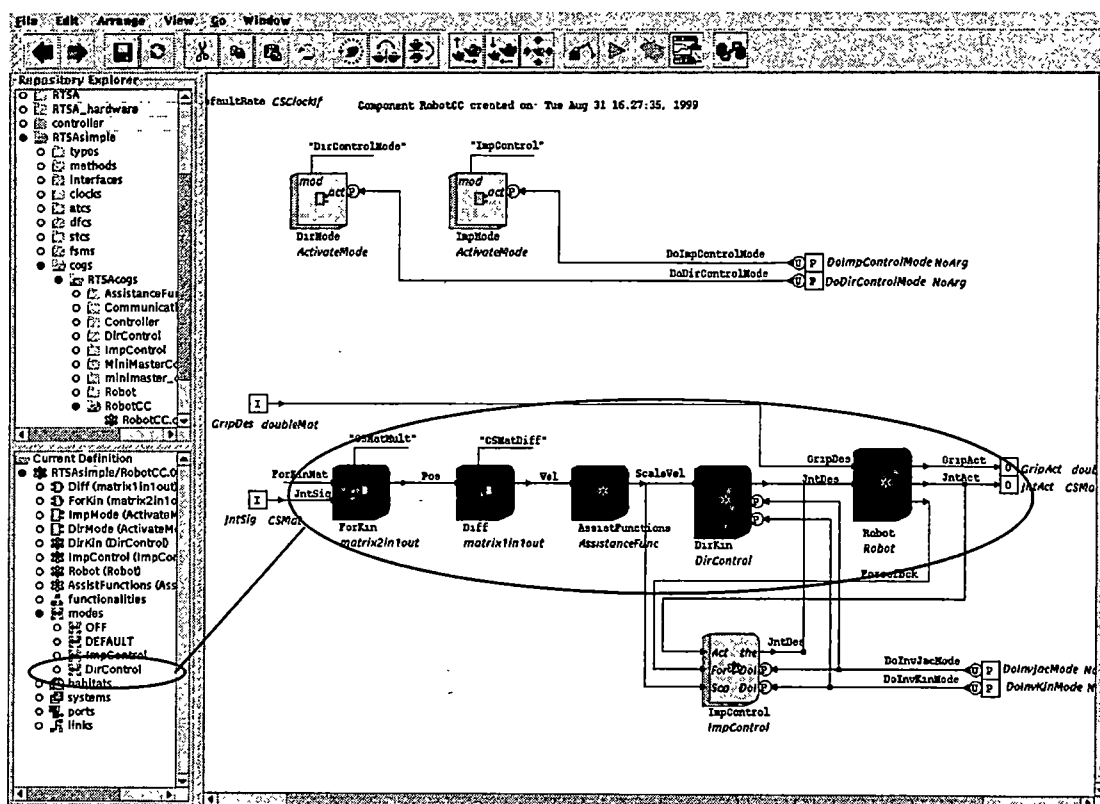


Figure 5.8: RobotCC COG: DirControl mode selected

CHAPTER 5. CONTROL SHELL IMPLEMENTATION

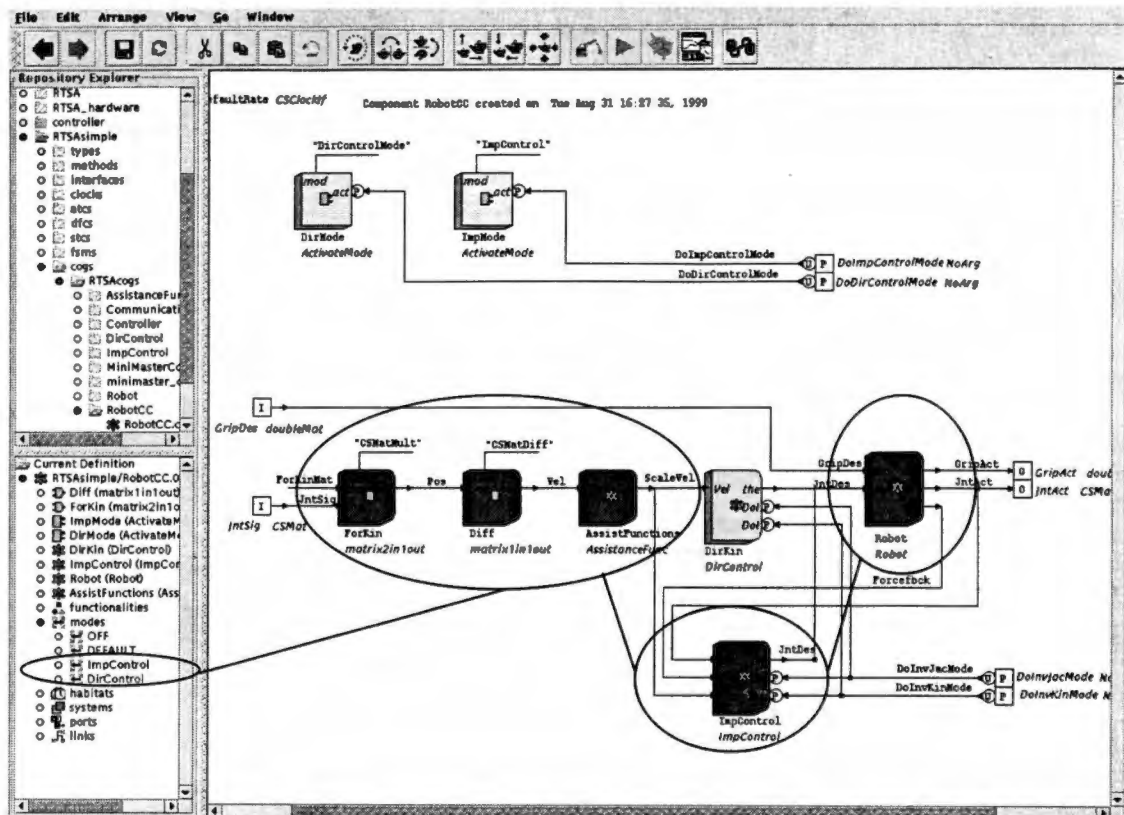
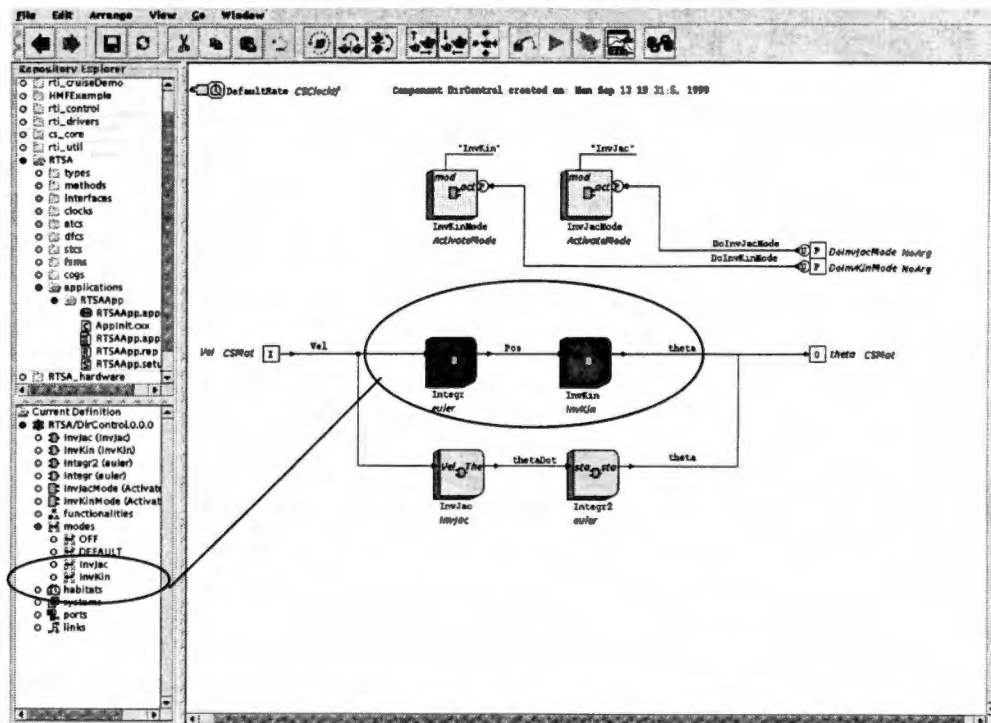
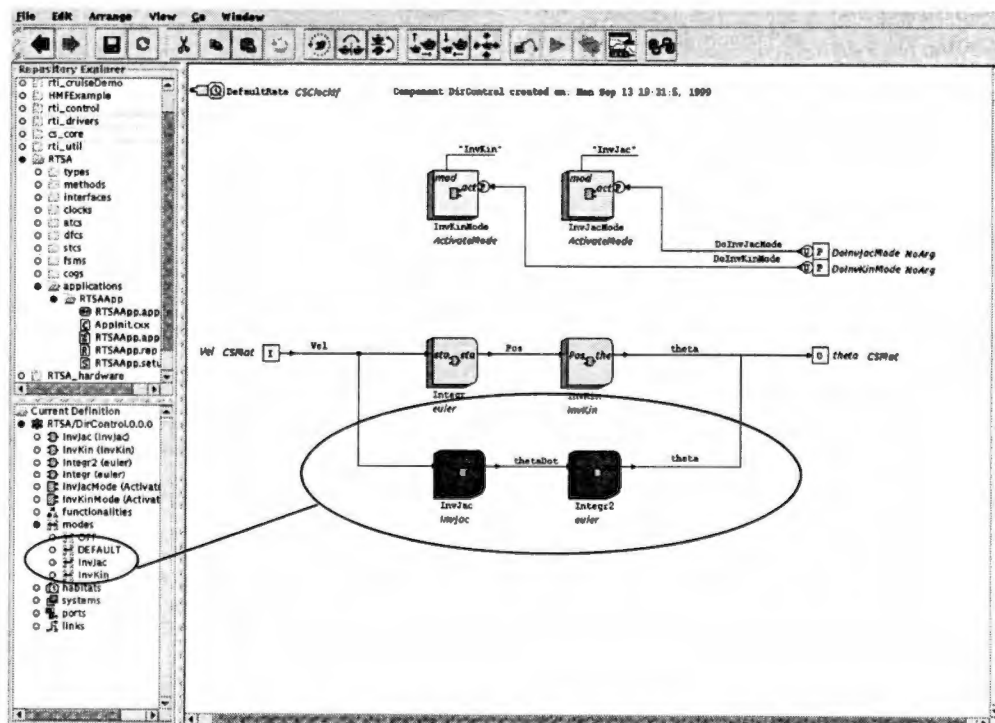


Figure 5.9: RobotCC COG: ImpControl mode selected

CHAPTER 5. CONTROL SHELL IMPLEMENTATION



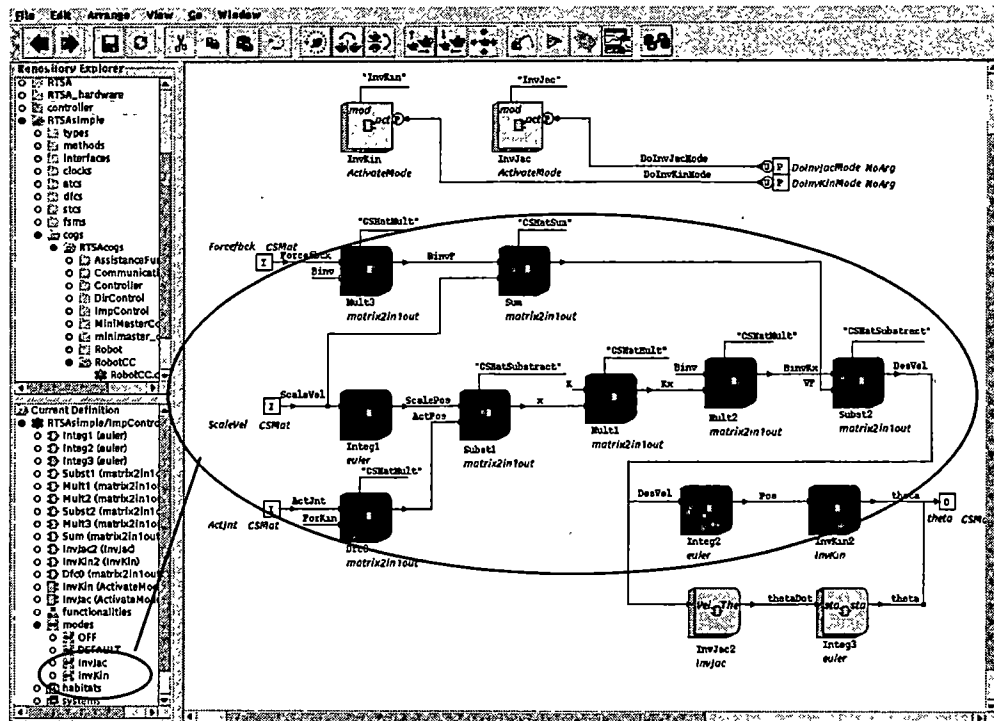
(a): InvKin mode selected



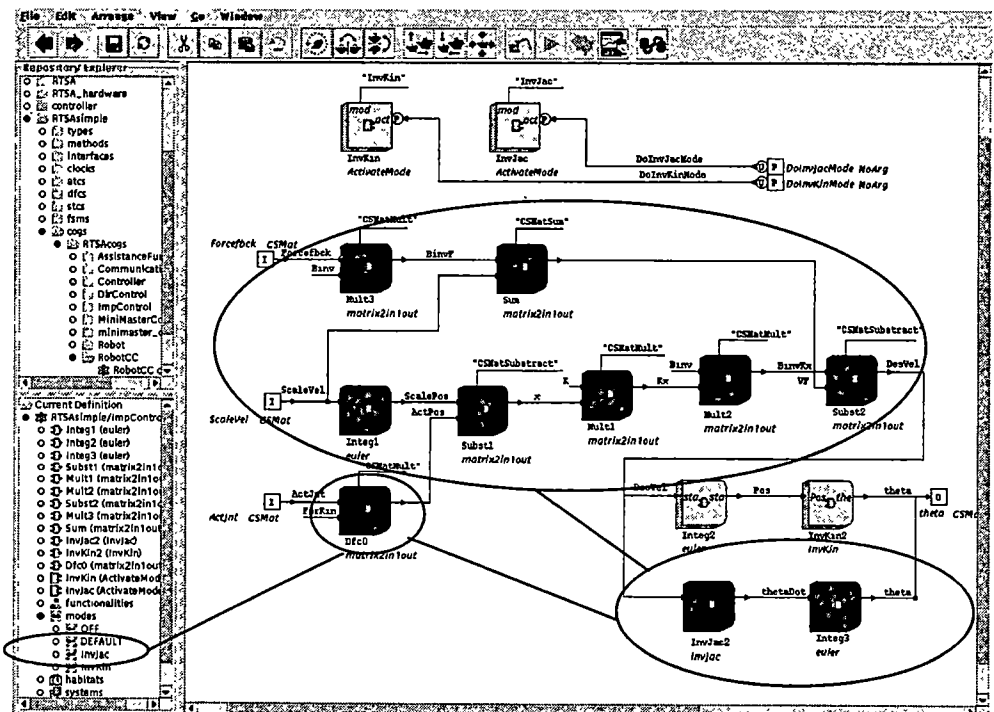
(b): InvJac mode selected

Figure 5.10: DirControl COG

CHAPTER 5. CONTROL SHELL IMPLEMENTATION



(a): InvKin mode selected



(b): InvJac mode selected

Figure 5.11: ImpControl COG

CHAPTER 5. CONTROL SHELL IMPLEMENTATION

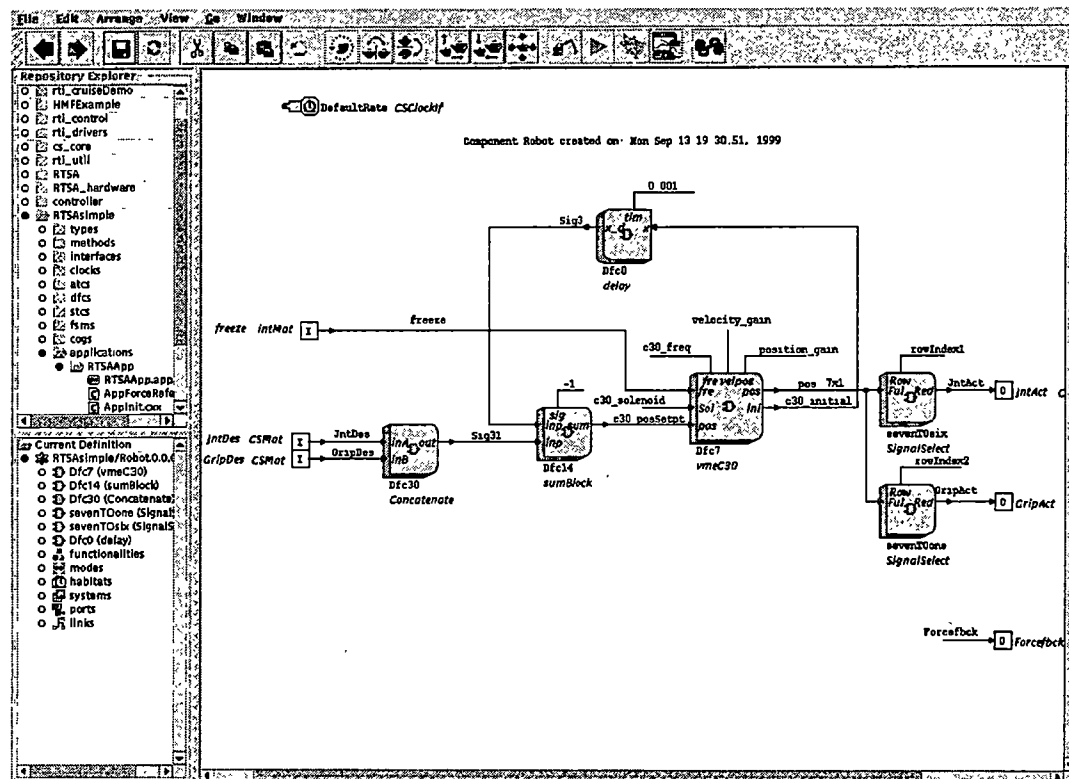


Figure 5.12: Robot COG

5.5 Controller Interface

5.5.1 ControlShell menus interface

The ControlShell run-time menu system allows exploring and controlling a running ControlShell application. The run-time menus can be seen, accessed and edited during the program execution [Appendix 2]. ControlShell text interface with the current Task Planner will be added as a new menu option in the main directory. Each time when the operator wants to see or to edit the Task Plan a Task Plan menu option can be invoked by typing 'task plan' as a command string.

5.5.2 ControlShell messages

Other option of the ControlShell text interface with the operator is C++ print messages. If the C++ code for the corresponding ControlShell function includes any print message commands this message will be printed on the display each time function is executed. For example, if ControlShell unavailable to open the taskplan.txt file during the Fetch_Parse () function execution message "Task Plan file cannot be open. Back to IDLE state" will be displayed. The message communication between ControlShell and the operator allows the operator to always be aware of what's going on during the program execution.

5.5.3 Minimaster menu screens.

The operator can communicate with ControlShell by using the MiniMasterCommunication interface. Master has twelve different buttons, which can be used to activate different stimuli in the ControlShell. With different submenu screens of the Minimaster the same buttons can call different functions. The MiniMasterCommunication COG is shown in Figure 5.13.

CHAPTER 5. CONTROL SHELL IMPLEMENTATION

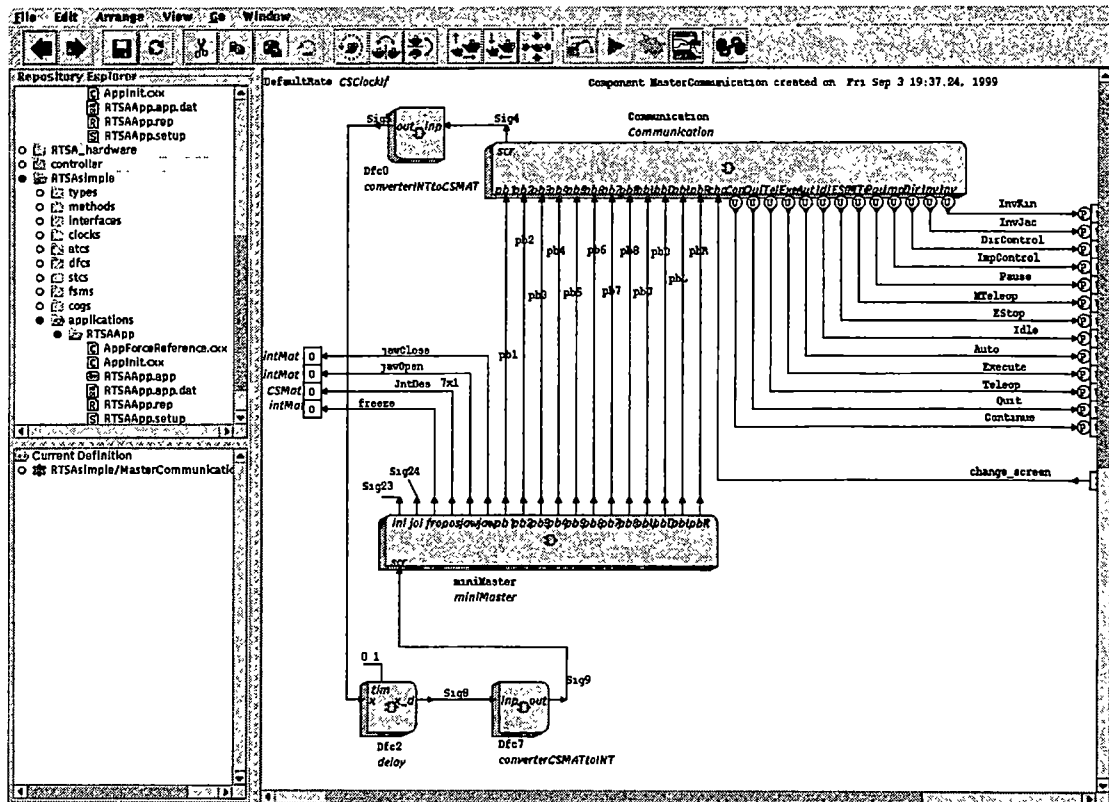


Figure 5.13: MasterCommunication COG

CHAPTER 5. CONTROL SHELL IMPLEMENTATION

Each channel (pb) in the miniMaster block [Appendix 3] corresponds to the different buttons on the minimaster. When a button is activated the Boolean signal is sent to the Communication block [Appendix 4]. Depending on the submenu screen number in the Communication block each signal is converted to the corresponding command and sent as a “bubble” to the main Controller. Each time the screen has to be changed the Communication block sends the “screen” Boolean signal to the miniMaster. Signal “change_screen” is sent from the DESController when the system leaves the autonomous execution. Three different submenu screens are used. Figure 5.14 shows the first screen corresponding to the beginning of the program or each time the program has to be restarted.

```
SELECT THE LOW-LEVEL CONTROLLER
-----
> DIRECT INV. KINEMATICS CONTROL
> DIRECT INV. JACOBIAN CONTROL
> IMPEDANCE INV. KINEMATICS CONTROL
> IMPEDANCE INV. JACOBIAN CONTROL
```

Figure 5.14: Welcome screen

After the operator makes a selection and pushes the corresponding button on the minimaster the screen will be changed to the next one with the Boolean “screen” signal from the Communication block. The next submenu screen is IDLE SCREEN. [Figure 5.15]

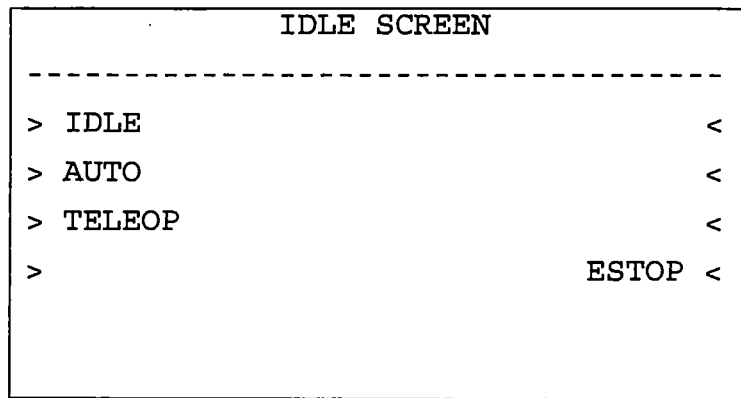


Figure 5.15: Idle screen

Each time at any screen if "ESTOP" button is selected screen will be changed to the first screen. Buttons "TELEOP" and "IDLE" will leave the same screen on. Selection of the "AUTO" button will change the current screen to the PAUSE SCREEN [Figure 5.16]. This screen will be changed back to the IDLE SCREEN automatically after the completing of the planned task or by pushing "QUIT" button.

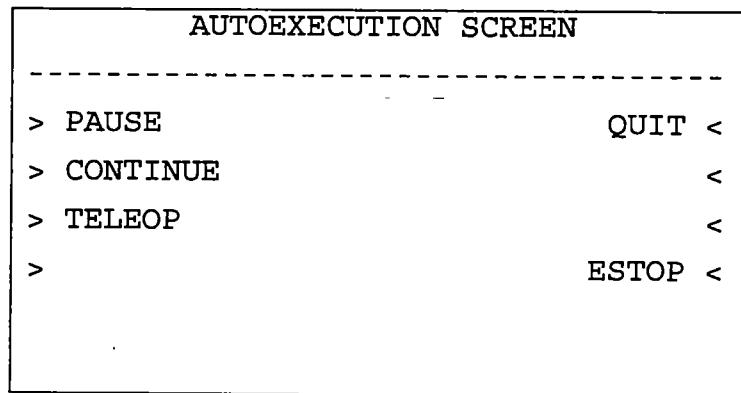


Figure 5.16: Autoexecution screen

Chapter 6

Conclusions

In this thesis a new telerobotic system controller with preplanned task scheduling was developed. An analytical approach has been proposed to design an interface between discrete commands and continuous commands, such that the major difficulty in designing a hybrid telerobotic system that involves both discrete event and continuous dynamics.

The development of the controller includes the following postulations:

- The telerobotic system with preplanned task scheduling can be recognized as a double hybrid system and can be analyzed using the hybrid system theory.
- The appropriate controller for the hybrid telerobotic system can be a hybrid discrete-continuous controller, with the discrete controller at the high level and the continuous controller at the low level.
- The discrete part of the hybrid controller can be modeled as a finite-state machine, using the theory of the finite automata.
- The interface between the discrete and continuous levels of the hybrid controller can be represented as a mode changing operation.

As a result, it has been shown that design of the hybrid telerobotic system controller can be resolved without use of complicated mathematical methods of Petri nets, Max-Plus algebra, or Fuzzy Logic. The proposed finite automata method appears to be efficient, comprehensive, and general. Also it has been shown that practical implementation of the introduced controller does not require any additional programming, and can easily be realized through built-in units of commercial software.

CHAPTER 6. CONCLUSIONS

As example, the ControlShell implementation of the controller has been developed. The graphical user interface of the ControlShell allows the designed controller to be not only comprehensive, but also flexible to any extension of the system. The ControlShell ability to define operating modes allows the implementation of the modeled interface between the discrete event and continuous dynamics of the hybrid telerobotic system to be synchronized and smooth in order to achieve an accurate performance of the system in near real-time applications. The ControlShell execution under LINUX OS allows the expanding use of LINUX in engineering applications beyond computer networking.

The finite-state machine of the controller is completed, and running efficiently. At the moment the control system can operate only in the teleoperation mode, because not all components required for the autonomous execution have been built. Therefore, the desired proof of the accuracy of the finite automata method for the modeling of the hybrid telerobotic system controller with preplanned task scheduling cannot yet be done experimentally. Also, the quality of the ControlShell implementation of the system controller cannot be fully analyzed at this time.

Chapter 7

Future work

To fully achieve the objective of this research the ControlShell controller implementation has to be completed, and the accuracy of the controller performance has to be analyzed. If the controller performance will prove to meet the requirements, as a continuation of the hybrid telerobotic system with preplanned task planner analysis the following steps are suggested.

- The system can be tested using different sets of the low-level controls to select the best control or the controls combination for different tasks of the plan execution. As an addition to the already built-in direct kinematics and impedance controls any other low-level control can be implemented as a new set of the ControlShell components.
- The system can be tested using different assist functions to select the best strategy for the teleoperation execution.
- An investigation of the system task performance with and without a task planner can be completed to emphasize the difference in accuracy and speed of the teleoperation system versus hybrid telerobotic system.
- The design of the same controller using different mathematical method (for example, Petri nets) as a high level discrete controller of the hybrid system can be done to recognize advantages and disadvantages of the finite automata approach.
- Some other techniques of the task planner integration in the hybrid telerobotic system can be studied. One of the new approaches can use the theory of event-

CHAPTER 7. FUTURE WORK

based planning and control. Another possible focus of the investigation can be a combination of the high level controller and the expert advisory system. In this case the computer has to be implemented as one of the decision-making components of the FSM.

Bibliography

Bibliography

- [1] C. Dorf, editor. *Concise International Encyclopedia of Robotics*. John Wiley & Sons, Inc., 1990
- [2] S. Y. Nof, editor. *Handbook of industrial robotics*. John Wiley & Sons, Inc., second edition, 1999
- [3] J. Vertut and P. Coiffet. *Teleoperation and robotics: evolution and development, Robot Technology series Vol. 3a*, Prentice-Hall, Inc, 1964.
- [4] W. R. Hamel. Sensor-based planning and control in telerobotics. *Control in Robotics and automation. Sensor-based integration*, pp. 285-308. Academic Press, 1999.
- [5] L. I. Slutski. *Remote Manipulation Systems*. Kluwer Academic Publishers, 1998.
- [6] W. R. Hamel and J. Osborn. Human-Interactive and Multi-Spectral Robot Task Space Modeling. *American Nuclear Society 8th International Topical Meeting on Robotics and Remote Systems*, Pittsburgh, Pennsylvania, USA, Apr. 25-29, 1999.
- [7] *RTSA Technical Overview*. Robotics and Electromechanical Systems Laboratory University of Tennessee, Knoxville. To be publish in 2000.
- [8] *Titan II Manipulator System technical manual*. Version 1. GEC Alsthom Schilling Robotic Systems, Inc. 1996. sales.schilling@ind.alstom.com
- [9] P. J. Antsaklis. Hybrid Control Systems: An Introductory Discussion of the Special issue. *IEEE Transactions on Automatic Control* 48(4): 457-459, Apr. 1998.
- [10] B. J. McCarragher and H. Asada. The Discrete Event Control of Robotic Assembly Tasks. *ASME Journal of Dynamic Systems, Measurement, and Control*, 117: 384-393, Sept. 1995.

- [11] N. Xi and T. J. Tarn. Intelligent Planning and Control for Multirobot Coordination: an Event-Based Approach. *IEEE Transactions on Robotics and Automation*. 12(3): 439-452. 1996.
- [12] T. B. Sheridan. *Telerobotics, Automation, and Human Supervisory Control*. The MIT Press, Cambridge, Massachusetts, 1992.
- [13] *ControlShell User's Manual*, Version 6.0, Real-Time Innovations, 1999.
email: sales@rti.com
- [14] J. Lygeros and S. Sastry. Hybrid system control. *Hybrid Control II*, Volume 999: 166-190, Springer Verlag, 1995
- [15] A. Gill. *Introduction to the theory of finite-state machine*. McGraw-Hill Book Company, Inc., 1962.

Appendices

Appendix 1

Selected code of the Fetch_Parse () ControlShell component

/* Fetch_Parse \- Transition component short description
modification history

rtj,01jan1998,xxx Skeleton generated

DESCRIPTION:

This file implements the class for the Fetch_Parse transition component.
Generated automatically by ControlShell.
User modifications should be done in Fetch_Parse.cxx, not in
Fetch_Parse_stubs.cxx.

#include <stdio.h>

#include "rtilib/makeheader.h"

#include "cs.h"

// Included methods

#include "cs_core/NoArg.h"

//-----

.....
Public

```
class Fetch_Parse : public CSTransComponentClass {
protected:
    CSRealMat      *_x; /* output */
    CSRealMat      *_y; /* output */
    CSRealMat      *_z; /* output */
    CSRealMat      *_roll; /* output */
    CSRealMat      *_pitch; /* output */
    CSRealMat      *_yaw; /* output */
    intMat          *_GripBool; /* output */
    char            *_fileName; /* reference */
    CSRealMat      *_MaxVel; /* output */
}
```

APPENDIX 1. SELECTED CODE OF THE FETCH_PARSE () CONTROL SHELL COMPONENT

```

public:
    enum RETCODE {
        RETCODE_CONTINUE,
        RETCODE_DONE,
        RETCODE_ERROR,
        numRETCODE
    };

    virtual ~Fetch_Parse();
    // The method called by the FSM engine when this STC is triggered

    virtual int OnExecute();
    // Custom Component Methods
    ///
    virtual int TOOL( );
    ///
    virtual int TELEOP( );
    ///
    virtual int GRIPPER( );
    ///
    virtual int TRAJ( );

    // Class-level initialization
    static void OnClassInitialize();

.....

#define BUFSIZE (1024)

//-----

Fetch_Parse::Fetch_Parse( CSFsmClass *fsm, const char *name,
    const char *className )
    : CStansComponentClass(fsm, name, className)
{
    _fileName = NULL;
    // Additional initialization code goes here.
    // veronika
    _fp = NULL;
}
// End of Fetch_Parse::Fetch_Parse

//-----

Fetch_Parse::~~Fetch_Parse()
{
    if ( _fileName != NULL ) { free( _fileName ); _fileName = NULL; }
    // Your destructor code goes here
    // veronika
    if ( _fp != NULL ) { fclose( _fp ); _fp = NULL; }
}

```

APPENDIX 1. SELECTED CODE OF THE FETCH_PARSE () CONTROL SHELL COMPONENT

```

}
// End of Fetch_Parse::~Fetch_Parse
//-----
//-----
// veronika
bool Fetch_Parse::ProcessCommand(char *varName, CSReal value)
{
    // mode: move = 1; gripper = 2; tool = 3; teleop = 4;
    if (strcmp(varName, "mode") == 0) {
        switch ((int) value)
        {
            case 1: // mode == MOVE
                // FinalTrajPos = [x,y,z,roll,pitch,yaw]
                // MaxVel = cruise velocity
                if (strcmp(varName, "x") == 0) { _x->El(0) = value;
                } else if (strcmp(varName, "y") == 0) { _y->El(0) = value;
                } else if (strcmp(varName, "z") == 0) { _z->El(0) = value;
                } else if (strcmp(varName, "roll") == 0) { _roll->El(0) = value;
                } else if (strcmp(varName, "pitch") == 0) { _pitch->El(0) = value;
                } else if (strcmp(varName, "yaw") == 0) { _yaw->El(0) = value;
                } else if (strcmp(varName, "cr_vel") == 0) { _MaxVel->El(0) = value; }
                TRAJ();
                fprintf(stderr, "MOVE execution.\n");
                break;

            case 2: // mode == GRIPPER
                // if open send 1, if close send 0
                if (strcmp(varName, "open") == 0) { _GripBool->El(0) = (int) value;
                if (value == 1) { fprintf(stderr, "Gripper is OPEN.\n");
                } else if (value == 0) { fprintf(stderr, "Gripper is CLOSE.\n");
                } else { fprintf(stderr, "Gripper information is insufficient. Check your plan
file!\n");
                } }
                GRIPPER();
                break;

            case 3: // mode == TOOL
                // type: if saw send 1, if drill send 2...
                // power: on == 1, off == 0
                // if ( strcmp(varName, "type") == 0 { _ToolType->El(0) = value;
                // } else
                if (strcmp(varName, "power") == 0) { _ToolPower->El(0) = 1;
                if (value == 0)
                { fprintf(stderr, "Turn OFF the tool.\n");
                } else if (value == 1)
                { fprintf(stderr, "Turn ON the tool.\n");
                } else { fprintf(stderr, "TOOL information is insufficient. Check your
plan file!\n");
                } }
                TOOL();

```

APPENDIX 1. SELECTED CODE OF THE FETCH_PARSE () CONTROL SHELL COMPONENT

```

        break;

        case 4:                                // mode == TELEOP
            // type: manual == 1, assist == 2
            // method: 1,2,3 ???
            // if ( strcmp(varName, "type") == 0 { _TeleopType->El(0) = value;
            // } else if (strcmp(varName, "method") == 0) { _TeleopMethod->El(0) = value;}
            TELEOP();
            fprintf(stderr, "Do Teleoperation.\n");
            break;
            default: return -1;
        }
    } else {
        fprintf(stderr, "*** %s (%s) : Bad command \"%s\"\n",
            Name(), ClassName(), varName);
        return false; }
    return true;
}

//-----

// parses one line with the format:
//
// mode 1 x 1 y 1 z 0...
//
bool Fetch_Parse::ExecuteCommand(char *command)
{
    char *lasts;
    char *endptr;
    char *SEPARATOR_TOK = "\t";
    char *varName;
    char *varValueStr;
    char *durationStr;
    CSReal varValue;
    bool executed = false;
    durationStr = strtok(command, SEPARATOR_TOK, &lasts);
    // check for failure
    if ( endptr == durationStr ) {
        //printf("*** did not find first number\n");
        return false; // skip the command
    }
    // each time around the loop it should parse a name/value pair
    while ( (varName = strtok(NULL, SEPARATOR_TOK, &lasts))
        != NULL ) {
        varValueStr = strtok(NULL, SEPARATOR_TOK, &lasts);
        if ( varValueStr == NULL ) { // missing value!! skip rest
            printf("*** returned from varValueStr == NULL\n");
            return executed;
        }
        varValue = (CSReal)strtod(varValueStr, &endptr);
        // check for failure

```

APPENDIX 1. SELECTED CODE OF THE FETCH_PARSE () CONTROL SHELL COMPONENT

```

        if ( endptr == varValueStr ) {
            printf("**** returned from endptr == varValueStr\n");
            return executed; // skip the rest of the line
        }
        executed = ProcessCommand( varName, varValue);
    }
    return executed;
}
//-----
/**
The main action routine for Fetch_Parse State-Transition component.
{\bf Description}
This is the Fetch_Parse transition component's main routine.
It is called by the Finite State Machine engine when a transition
associated with this component is triggered.
This should return the appropriate RETCODE return code.
*/
int Fetch_Parse::OnExecute()
{
    char buffer[BUFSIZE];
    char *ch;
    // open the file:
    if (_fp == NULL) {
        _fp = fopen(_fileName, "r");
    }
    if (_fp == NULL) {
        fprintf(stderr, "**** %s (%s) : Cannot open file \"%s\"\n",
            Name(), ClassName(), _fileName);
        return RETCODE_ERROR;
    }
    // fetch the next command from the file, skipping comment lines:
    do {
        ch = fgets(buffer, BUFSIZE, _fp);
    } while ( ( ch != NULL) &&
        ((buffer[0] == '#')
            || (ExecuteCommand(buffer) == false)) );
    // close the file, if we are at the end:
    if (feof(_fp)) {
        fclose(_fp);
        _fp = NULL;
        return RETCODE_DONE;
    }
    return RETCODE_CONTINUE;
}
// End of Fetch_Parse::OnExecute

```

Appendix 2

ControlShell menus interface

```

ControlShell version 6.0s.
Copyright 1991-9, Real-Time Innovations, Inc. All rights reserved.

Functionality (? for options ; to abort) =? [UNCATEGORIZED] Implementation (? f
or options ; to abort) =? [ALL_OFF] Force callback of OnMode (y/[n]) ---> Imple
mentation "UNASSIGNED_DEFAULT" activated for functionality "UNCATEGORIZED" in co
g "RUN_FSM (TheSystem)".
Sample habitat to start (? for options ; to abort) =? [DefaultRate] --> Default
Rate (CSSampleHabitat) : Habitat started.
Control loop executing as process "DefaultRate".
Install all signals through which depth (-1 == all) =? [-1] Available scope in
dices are:
    Habitat          Scope Index
    DefaultRate      1
StethoScope Index (-1 to abort) =? [1] New state =? [no change] Enter time to pa
use (sec) =? [0] FSM (8158218) DESCController: [ "DEFAULT" -> DESCController ] ==>
[ "NULL"() -> -100002 ] ==> E_STOP
Stimulus to send =? [ ] Enter (void *) parameter =? [0] send as high priority sti
mulus (y/[n]) Enter time to pause (sec) =? [0] FSM "DESCController" stimulus: "Id
le"
CSFsmState (815fc60) DESCController: [ "Idle" -> E_STOP ] ==> [ {}() -> -100002 ]
==> IDLE
ControlShell-> Stimulus to send =? [Idle] Enter (void *) parameter =? [0] send a
s high priority stimulus (y/[n]) FSM "DESCController" stimulus: "Auto"
*** StartAuto (StartAuto) : Cannot open file "home/vgospoda/RTSAsimple/fsms/RTSA
fsm/FSM/taskplan.txt"
home/vgospoda/RTSAsimple/fsms/RTSAfsm/FSM/taskplan.txt: No such file or director
y
CSFsmState (815f760) DESCController: [ "Auto" -> IDLE ] ==> [ {StartAuto}() -> 1
] ==> IDLE

ControlShell-> ls

Options for "/RUN_FSM/RUNfsmCOG" are:
Menu options:
scope          - Scope manager
functionality   - Functionality managment and activation
mode           - Mode control
comp           - Component management
habitat        - Habitat execution control
xecute         - BEING DEPRECATED: please use "habitat" option instead
signal         - signal manager
registry        - CSRegistry management
help*          - Help facility
menu*          - Menu manager control
ccomp*         - Change current component
pause          - pause menu input
!             - Escape to shell
quit           - Quit Application

Components at this level:
DESCController [FSM]
ControlShell-> mode

Options for "/RUN_FSM/RUNfsmCOG mode" are:
Menu options:
display        - Display a mode
activate       - Activate a mode

/RUN_FSM/RUNfsmCOG mode: Selection-> █

```

Appendix 3

Selected code for the miniMaster component

```
/* miniMaster \- Data-flow component short description
modification history
```

```
-----
rti,01jan1998,xxx Skeleton generated
```

DESCRIPTION:

This file implements the class for the miniMaster data-flow component.
Generated automatically by ControlShell.
User modifications should be done in miniMaster.cxx, not in
miniMaster_stubs.cxx.

```
#include <miniMaster.h>
```

```
/* matt */
```

```
*/
```

```
#include <stdio.h>
```

```
#include "rtilib/makeheader.h"
```

```
#include "cs.h"
```

```
//-----
```

```
Public
```

```
class miniMaster : public CSDfcClass {
```

```
protected:
```

```
    CSRealMat    *_initPos; /* output */
    CSRealMat    *_jointPos; /* output */
    intMat        *_frozen; /* output */
    CSRealMat    *_posSetPt; /* output */
    intMat        *_jawOpen; /* output */
    intMat        *_jawClose; /* output */
    intMat        *_pb1; /* output */
    intMat        *_pb2; /* output */
    intMat        *_pb3; /* output */
    intMat        *_pb4; /* output */
    intMat        *_pb5; /* output */
    intMat        *_pb6; /* output */
    intMat        *_pb7; /* output */
```

APPENDIX 3. SELECTED CODE FOR THE MINIMASTER COMPONENT

```

intMat      *_pb8; /* output */
intMat      *_pbUp; /* output */
intMat      *_pbDown; /* output */
intMat      *_pbLeft; /* output */
intMat      *_pbRight; /* output */
intMat      *_screen; /* input */

public:
virtual ~miniMaster();
// Standard Component Methods
///
virtual int OnExecute( int rateIdx );
///
virtual int OnActivate();
///
virtual int OnStartup( int rateIdx );
///
virtual int OnShutdown( int rateIdx );
///
virtual int OnTerminate();

.....

// Additional user-defined protected members and methods

private:
// Additional user-defined private members and methods
masterAccess *ma; /* matt */
int last_screen;

.....

//-----
miniMaster::miniMaster( CSSystemClass *system,
const char *compName, const char *className,
CSCogClass *parent, int nRates, bool feedthrough)
: CSDfcClass(system, compName, className,
nRates, feedthrough, parent, CSDfcClass::CLASS_NAME)
{
// Additional initialization code goes here.
ma = new masterAccess;
memset(ma, 0, sizeof(masterAccess)); /* matt */
last_screen = 0;
}
// End of miniMaster::miniMaster
//-----
miniMaster::~miniMaster()
{
// Your destructor code goes here
delete ma;
}

```

APPENDIX 3. SELECTED CODE FOR THE MINIMASTER COMPONENT

```
// End of miniMaster::~miniMaster
//-----
.....
*/
int miniMaster::OnInstance( int verbosity )
{
    CSUseParametersAndLocals(verbosity); // do not remove
    if (MenusCreate() == false) {
        fprintf(stderr, "*** Error %s (%s): Could not create menu "
            "options.\n", Name(), ClassName());
    }

    // Your instantiation code goes here (return negative value for error)
    if (!ma)
    {
        fprintf(stderr, "*** Error %s (%s): Cannot allocate storage for "
            "internal registers.\n", Name(), ClassName());
        return -1;
    }
    if (masterOpen(ma)) return -1;
    /* check for matrix size */
    if (
        (_initPos->m() == 0) || (_initPos->n() == 0) ||
        (_jointPos->m() == 0) || (_jointPos->n() == 0) ||
        (_posSetPt->m() == 0) || (_posSetPt->n() == 0))
    {
        fprintf(stderr, "*** Error %s (%s): Output not initialized\n",
            Name(), ClassName());
        return -1;
    }

    if (masterGetInitPos(ma)) return -1;
    return 0;
}
// End of miniMaster::OnInstance
/* =====
* Below are the standard method(s) that you have selected for this component
* ===== */
//-----
/**
Main sample-loop routine for miniMaster.
{\bf Description}
This is the miniMaster component's OnExecute routine.
It is called each sample "loop".
@return If error, return a negative value.
*/

int miniMaster::OnExecute( int /* rateIdx */)
{
    register int i;
    // If Error, return negative value
    /* actually get the feedback */
    if (masterGetFdbk(ma) != 0)
    {
```

APPENDIX 3. SELECTED CODE FOR THE MINIMASTER COMPONENT

```

        fprintf(stderr, "MasterGetFdbk failed!!!!!!!!!!!!\n");
        return -1;
    }
    /* now, put the data in control shell variables */
    for (i=0;i<MASTERJNTS;i++)
    {
        _initPos->El(i) = ma->fdbk.initPos[i];
        _jointPos->El(i) = ma->fdbk.jointPos[i];
        _posSetPt->El(i) = ma->fdbk.posSetPt[i];
    }
    _frozen->El(0) = ma->fdbk.frozen;
    _jawOpen->El(0) = ma->fdbk.opnJaw;
    _jawClose->El(0) = ma->fdbk.clsJaw;
    _pb1->El(0) = ma->fdbk.fk1;
    _pb2->El(0) = ma->fdbk.fk2;
    _pb3->El(0) = ma->fdbk.fk3;
    _pb4->El(0) = ma->fdbk.fk4;
    _pb5->El(0) = ma->fdbk.fk5;
    _pb6->El(0) = ma->fdbk.fk6;
    _pb7->El(0) = ma->fdbk.fk7;
    _pb8->El(0) = ma->fdbk.fk8;
    _pbUp->El(0) = ma->fdbk.fkup;
    _pbDown->El(0) = ma->fdbk.fkdn;
    _pbLeft->El(0) = ma->fdbk.fklf;
    _pbRight->El(0) = ma->fdbk.fkrt;
    fprintf(stderr, "screen = %s\n", _screen->El(0));
    if (_screen->El(0) != last_screen)
    {
        switch (_screen->El(0))
        {
            case 1:                /* WELCOME screen 1 */
                masterPrint(ma, "WELCOME----select controller----", 1, 0, CURSOFF, 0);
                masterPrint(ma, "<-Dir Inv Kin", 2, 0, CURCURS, 0);
                masterPrint(ma, " ", 3, 0, CURCURS, 0);
                masterPrint(ma, " ", 4, 0, CURCURS, 0);
                masterPrint(ma, " ", 5, 0, CURCURS, 0);
                masterPrint(ma, "<-Dir Inv Jac", 6, 0, CURCURS, 0);
                masterPrint(ma, " ", 7, 0, CURCURS, 0);
                masterPrint(ma, " ", 8, 0, CURCURS, 0);
                masterPrint(ma, " ", 9, 0, CURCURS, 0);
                masterPrint(ma, "<-Imp Inv Kin", 10, 0, CURCURS, 0);
                masterPrint(ma, " ", 11, 0, CURCURS, 0);
                masterPrint(ma, " ", 12, 0, CURCURS, 0);
                masterPrint(ma, " ", 13, 0, CURCURS, 0);
                masterPrint(ma, "<-Imp Inv Jac", 14, 0, CURCURS, 0);
                masterPrint(ma, "-----", 15, 0, CURCURS, 0);
                break;

            case 2:                /* IDLE screen 2 */
                masterPrint(ma, "-----CURRENT STATE: IDLE-----", 1, 0, CURSOFF, 0);
                masterPrint(ma, "<-Teleop", 2, 0, CURCURS, 0);

```

APPENDIX 3. SELECTED CODE FOR THE MINIMASTER COMPONENT

```

masterPrint(ma, "          ", 3, 0, CURCURS, 0);
masterPrint(ma, "          ", 4, 0, CURCURS, 0);
masterPrint(ma, "          ", 5, 0, CURCURS, 0);
masterPrint(ma, "<-Autoexec          ", 6, 0, CURCURS, 0);
masterPrint(ma, "          ", 7, 0, CURCURS, 0);
masterPrint(ma, "          ", 8, 0, CURCURS, 0);
masterPrint(ma, "          ", 9, 0, CURCURS, 0);
masterPrint(ma, "<-Idle          ", 10, 0, CURCURS, 0);
masterPrint(ma, "          ", 11, 0, CURCURS, 0);
masterPrint(ma, "          ", 12, 0, CURCURS, 0);
masterPrint(ma, "          ", 13, 0, CURCURS, 0);
masterPrint(ma, "          EStop->", 14, 0, CURCURS, 0);
masterPrint(ma, "-----", 15, 0, CURCURS, 0);
break;

case 3: /* Autoexec screen 3 */
masterPrint(ma, "-----CURRENT STATE: PAUSE-----", 1, 0, CURSOFF, 0);
masterPrint(ma, "<-Execute          Quit->", 2, 0, CURCURS, 0);
masterPrint(ma, "          ", 3, 0, CURCURS, 0);
masterPrint(ma, "          ", 4, 0, CURCURS, 0);
masterPrint(ma, "          ", 5, 0, CURCURS, 0);
masterPrint(ma, "<-Pause          ", 6, 0, CURCURS, 0);
masterPrint(ma, "          ", 7, 0, CURCURS, 0);
masterPrint(ma, "          ", 8, 0, CURCURS, 0);
masterPrint(ma, "          ", 9, 0, CURCURS, 0);
masterPrint(ma, "<-Teleop          ", 10, 0, CURCURS, 0);
masterPrint(ma, "          ", 11, 0, CURCURS, 0);
masterPrint(ma, "          ", 12, 0, CURCURS, 0);
masterPrint(ma, "          ", 13, 0, CURCURS, 0);
masterPrint(ma, "<-Continue          EStop->", 14, 0, CURCURS, 0);
masterPrint(ma, "-----", 15, 0, CURCURS, 0);
break;
default:
fprintf(stderr, "Unknown screen %d\n", _screen->El(0));
return -1;
}
last_screen = _screen->El(0);
}
return 0;
}
// End of miniMaster::OnExecute

```

Appendix 4

Selected code for the Communication component

```
/* Communication \- Data-flow component short description
modification history
```

```
-----
rti,01jan1998,xxx Skeleton generated
```

```
DESCRIPTION:
```

```
  This file implements the class for the Communication data-flow component.
  Generated automatically by ControlShell.
  User modifications should be done in Communication.cxx, not in
  Communication_stubs.cxx.
```

```
===== */
#include <stdio.h>
```

```
#include "rtilib/makeheader.h"
```

```
#include "cs.h"
```

```
// Included methods
```

```
#include "cs_core/NoArg.h"
```

```
.....
Public
class Communication : public CSDfcClass {
```

```
protected:
```

```
  intMat      *_pb1; /* input */
  intMat      *_pb2; /* input */
  intMat      *_pb3; /* input */
  intMat      *_pb4; /* input */
  intMat      *_pb5; /* input */
  intMat      *_pb6; /* input */
  intMat      *_pb7; /* input */
  intMat      *_pb8; /* input */
  intMat      *_pbUp; /* input */
  intMat      *_pbDown; /* input */
  intMat      *_pbLeft; /* input */
```

APPENDIX 4. SELECTED CODE FOR THE COMMUNICATION COMPONENT

```
intMat      *_pbRight; /* input */
intMat      *_screen; /* output */
intMat      *_changescreen; /* input */

public:
    virtual ~Communication();
    // Standard Component Methods
    ///
    virtual int OnExecute( int rateIdx );

virtual int OnStartup( int rateIdx );

    // Custom Component Methods
virtual int Continue( );
virtual int Quit( );
//
virtual int Teleop( );
//
virtual int Execute( );
//
virtual int Auto( );
//
virtual int Idle( );
//
virtual int EStop( );
//
virtual int MTeleop( );
//
virtual int Pause( );
//
virtual int ImpControl( );
//
virtual int InvJac( );
//
virtual int InvKin( );

.....

// Additional user-defined protected members and methods

private:
    // Additional user-defined private members and methods
    int next_screen; /* veronika */
    int control_number;
    bool Control( int control_number );

.....

// Additional initialization code goes here.
    next_screen = 1; /* veronika */
    control_number = 1;
}
// End of Communication::Communication
```

APPENDIX 4. SELECTED CODE FOR THE COMMUNICATION COMPONENT

```

.....
// Control selection                                /* veronika */
bool Communication::Control(int control_number)
{
    switch (control_number)
    {
        case 1:                                     /* Dir Inv Kin Control */
            DirControl( );
            InvKin( );
            break;

        case 2:                                     /* Dir Inv Jac Control */
            DirControl( );
            InvJac( );
            break;

        case 3:                                     /* Imp Inv Kin Control */
            ImpControl( );
            InvKin( );
            break;

        case 4:                                     /* Imp Inv Jac Control */
            ImpControl( );
            InvJac( );
            break;
        default: return -1;
    }
    return true;
}
/**

```

```

.....
int Communication::OnExecute( int /* rateIdx */)
{
    // If Error, return negative value
    fprintf(stderr, "*** Communication on execute\n");
    if (_changescreen->El(0) != 1);
    {
        switch ( next_screen )
        {
            case 1:                                 /* WELCOME screen */
                if ( _pb1->El(0) == 1 ) { control_number = 1;
                fprintf(stderr, "*** Checking pb1\n"); }
                else if ( _pb2->El(0) == 1 ) { control_number = 2; }
                else if ( _pb3->El(0) == 1 ) { control_number = 3; }
                else if ( _pb4->El(0) == 1 ) { control_number = 4; }
                Control(control_number);
                next_screen = 2;
                break;

            case 2:                                 /* IDLE screen */

```

APPENDIX 4. SELECTED CODE FOR THE COMMUNICATION COMPONENT

```

Control(control_number);
if (_pb1->El(0) == 1) {
    MTeleop();
} else if (_pb2->El(0) == 1) {
    Auto();
    next_screen = 3;
} else if (_pb3->El(0) == 1) {
    Idle();
} else if (_pb8->El(0) == 1) {
    next_screen = 1; }
    break;

    case 3:                                /* AUTOEXEC screen */
Control(control_number);
if (_pb1->El(0) == 1) {
    Execute();
} else if (_pb2->El(0) == 1) {
    Pause();
} else if (_pb3->El(0) == 1) {
    Teleop();
} else if (_pb4->El(0) == 1) {
    Continue();
} else if (_pb5->El(0) == 1) {
    Quit();
} else if (_pb8->El(0) == 1) {
    next_screen = 1; }
    break;
default: return -1;
}
    _screen->El(0) = next_screen;
}
    _screen->El(0) = 2;
    return 0;
}
// End of Communication::OnExecute

```

Vita

Veronika Leonidovna Gospodareva was born in Kaliningrad, Russia, daughter of Elena and Leonid. She has one younger sister Julia.

At thirteen she was accepted to the Special School of Physics and Mathematics for gifted teenagers in Moscow, Russia. After high school, Veronika attended the Moscow State University, department of the Applied Mechanics and Mathematics. She graduated in 1989 with her Masters of Science degree in Applied Physics and Mechanics. Before living Russia, Veronika worked as a research fellow in the Moscow Institute of Mechanics, specializing in development of walking robotic systems. In 1991 when Russians, finally, were allowed to go out of the country, she left to journey around the world.

In 1997 Veronika attended the University of Tennessee, Knoxville. During her time in the University she was member of the RTSA team at the Robotics and Electromechanical Systems Laboratory. She hopes to graduate this year with a Masters of Science degree in Mechanical Engineering, and find a job in area of robotics. Veronika has three hobbies: sports, books, and the company of interesting people.