Action Selection in Teleautonomous Systems *

Sean Graves   Richard Volz

Department of Computer Science, Texas A&M University

Abstract

Telerobotic techniques enable the performance of tasks which may be hazardous or impossible for humans to perform directly and provide the operator considerable flexibility. Autonomous systems are intended to operate without human intervention; however, they are brittle and have limited functionality at present. Teleautonomous systems *attempt to combine the advantages of telerobotics with those of autonomous systems.*

Numerous generalized architectures for telerobotic systems have been proposed; however, most are hard-wired instantiations of a specific approach, and the interactions between human and autonomous elements of teleautonomous systems have rarely been addressed by these systems. The work described here provides a much more general, easily extensible and modifiable architecture. Specific issues addressed include methods that fuse decisions from multiple sources (such as human, reactive, or deliberative), that integrate multiple command sources, and that accommodate multi-purpose event recognizers to change operating modes. These methods may be dynamically altered to generate a variety of control modes and generate reliable dextrous telerobot motion.

A result of this research is a modifiable, extensible framework for the unification of the many modes of control, such as shared, traded, and supervisory, which are described in the telerobotics literature.

1 Introduction

Teleautonomous systems (TaS's) [1] attempt to combine the advantages of telerobotics with those of autonomous systems by including the projection of cognitive processing. By placing intelligent functions at the work site, operator performance can be increased [2]. Also, communication delays between the robot and its control site can be tolerated more readily; indeed, a total loss of communication to the robot does not necessarily mean it must stop functioning.

In most previous work, the design of the telerobot software system is dictated by the specific application requirements and the specific hardware that is to be controlled. Although there is an intuitive feel that some standardization is needed in the design of software systems for controlling intelligent robotic systems, development of new systems typically proceeds as though particular design requirements necessitate the creation of a unique control system.

---

*This work has been supported by the State of Texas under TATP grant 999903-267 and by the NSF under grant CDA-9115123.
The authors may be contacted via Internet at sean@cs.tamu.edu.

Generalized architectures for telerobotic systems have been proposed, e.g., most notably the NASREM (NASA/NBS Standard Reference Model for Telerobot Control System Architecture) specification [3]. However, the interaction between human and autonomous elements of TaS's has not been addressed, e.g., the method by which the operator may switch between autonomous functions, or the techniques that might be used to specify when and how mode transitions should occur.

The problems attacked by this work are: How may control decisions from multiple sources (such as human, reactive, or deliberative) be integrated to generate reliable telerobot motion? How may the integration method be dynamically altered to generate a variety of control modes in response to recognition of different run-time events?

A result of this research is a framework for the unification of the many control methods described in the telerobotics literature. Rather than have a fixed set of control modes, this framework allows any number of possible configurations. The design of the architecture enables robust control of a remote robotic system under conditions of signal propagation delay, sensor noise, control errors, unstructured environments, and limited processing capability at the remote work site.

Specific limitations of previous teleautonomous architectures addressed by this work include:

- Improved flexibility in command sequencing. Previous teleautonomous systems, such as the MOTES system [4] of the Jet Propulsion Lab (JPL) execute commands from the control site in a strictly sequential fashion. Our design adds the capability of conditional and iterative execution of task commands. Just as programming language control constructs increase the expressiveness and power of the language, non-sequential execution of task commands based on the current system state allows greater flexibility.

- Intelligent use of system state. Reflexive actions should supplement regular task commands, not necessarily negate them. JPL's MOTES system includes reflex actions, but their execution subsumes commands from the operator. We seek greater flexibility in reflex action execution, by allowing reflex actions to adjust the system state without terminating the current task.

- User-configurable action selection. *Action selection* [5] refers to the operation of combining task commands from multiple sources into actions which a robot may perform. The method used to perform action selection should be configurable by the user. The shared control technique described by Hayati [6] used a mixing matrix which combined commands from an autonomous agent (a force controller) with commands from an operator. Our work expands this concept to include an arbitrary number of command sources, a-priori or on-the-fly specification of the shared control mode, and blending functions for transitioning between control modes.

- True reactive control capability. While most of the systems surveyed have some element of reactive control, most do not have provisions for directly coupling sensing to acting – the hallmark of reactive control [7]. Without this capability, it is difficult to ensure that the real-time constraints of the reactive controller will be met. This is due to the fact that reactive control systems forgo modeling in favor of acting and sensing and use the world as its own model. Without a very efficient link between actuators and sensors, reactive control becomes impossible.

- Distribution of command sources. Teleautonomous architectures have typically consisted of a tightly coupled set of processes which generate task commands. Improved flexibility results by allowing command sources to be distributed across multiple processors on a network.

The basic problems that must be addressed by telerobotic systems are well described by Sheridan [8]. Of particular interest are the stability problems created by time delay in the telerobotic system. The inclusion of autonomous agents provides one of the most obvious solutions to the time delay problem, and motivates our system design.

The problem of providing an intuitive and powerful interface between the operator and autonomous functions is exacerbated by the use of many control modes and functionalities. Sato and Hirai [9] discuss the need for the operator to interact with the telerobot system at various levels, such as the servo level (master/slave control of robot motions) and task level (task execution by programmed control). Although their system supports five different control modes, a method of automatically selecting and sequencing mode changes is not supported. Instead, mode switches are initiated explicitly by the operator.

The need for multi-level operator interaction is also discussed by Albus [3], but the manner in which this interaction should be performed or sequenced is not described.

The telerobotics work conducted by JPL has been described in numerous publications [10, 11, 4]. Their work includes MOTES (Modular Telerobot Task Execution System), which is a software architecture for sequencing commands for a telerobotic system. MOTES allows supervisory, teleoperated, and shared control. In supervisory control, the operator specifies task primitives or sequences of primitives for autonomous execution. In teleoperative mode, initial specification of parameters is performed by the operator, followed by direct control using force-reflecting hand controllers. Shared control merges inputs provided by the operator via hand controllers with task actions.

One limitation of MOTES involves the structure of the reflex action queues. When a reflex action is executed, the Task Command Queue is cleared. Thus, reflexive actions cannot be integrated with task commands. Also, since Reflex Command Queues are tied to specific monitor events, all possible exceptional conditions requiring reflex action must be pre-determined.

The German ROTEX telerobotic experiment [12] is another example of shared human-autonomous control. This space robot technology experiment flew in Spacelab mission D2 in 1993. A sophisticated 6-axis robot was placed inside a Spacelab rack where it performed assembly, connection/disconnection and grasping operations. The control system allowed for shared control, task planning, predictive simulation, and sensor prediction. ROTEX used a hierarchical architecture with three levels: a task planning level which generated a list of subtasks arranged in a graph structure, a task coordination level which sequenced the tasks, and a task execution level which carried out the control actions. Task planning is performed manually a-priori by the system designer. While an excellent demonstration of telerobotics, control modes in ROTEX were predetermined and inflexible. An open question is the scalability of the technology demonstrated.

The interaction of a human with a reactive wheelchair system is described in [13]. Because of ergonomic constraints, the operator did not have direct control of many system parameters; however, this system gave a very 'hands-on' demonstration of the power of coupling human intelligence with autonomous capability.

The use of reactive control in a telerobotic scenario is briefly described by Bon and Beahan in [11]. This system allows the user to specify the parameters of a motion or sensing command to be performed at the remote site, together with a set of reflexes and monitoring functions to be executed in real time during the motion. These reflex actions, however, form a small fixed set, and do not have much flexibility for reconfiguration or user interaction.

Action selection can be considered a subproblem of reactive control, however, it also has important links to teleautonomy. For instance, a reactive controller typically has multiple low-level behaviors which generate (possibly conflicting) commands. Similarly, a TaS must integrate commands from the operator(s) with those from autonomous agents. How this arbitration is performed is a crucial problem. Tyrrell [5] compared a number of action selection mechanisms (ASM's), primarily from the field of reactive control, including the techniques used by Maes [14], Rosenblatt and Payton [15], and Brooks [7]. The comparison was done using an elaborate simulated environment, which uncovered flaws in each of the ASM's. By making minor additions and modifications to the various ASM's, Tyrrell was able to achieve improved results, particularly in the case of Rosenblatt and Payton's system. We feel that the application of the concept of action selection to teleautonomous system is particularly useful because of the large amount of work which has been done in the field of reactive control.

2 System Object Model

Our approach to building the general system was to analyze first the problem of teleautonomy and create a basic system model. This model is built according to the Object Modeling Technique (OMT) of Rumbaugh [16]. OMT is an object-oriented modeling technique which was developed to improve the front-end tasks of systems development (requirements analysis, system design, and implementation). An object-oriented model (OOM) is valid for a wide range of functions that may be required in a particular domain – if the system's functional requirements change, the object-oriented model will remain valid. It is the use of object-oriented techniques that allows us to realize the capabilities claimed in Section 1. For further information on OMT, the reader is referred to Rumbaugh's text [16].

The teleautonomous system is comprised of at least the following elements: control sources, communication links, robots, sensors, and humans. A set of control sources must be tied to a particular robot through some mechanism, which we refer to as the Action Selection Mechanism (ASM). Other components which may be present include planners, modeling mechanisms, and simulations. These components may be aggregated in various ways to form control sites, work sites, or monitoring sites. A set of sites forms a configuration. This view of the teleautonomous system is shown graphically in OMT notation in Figure 1.

3 ASM Object Model

In a similar manner to that used to develop the teleautonomous system model, we construct an object model of the Action Selection Mechanism. We consider each of the requirements given in the Introduction, and construct the ASM Object Model accordingly. Refer to Figure 2 for details of the relationships between the ASM components. In the later sections, we will describe these components in more detail.

3.1 Flexible command sequencing

The fact that the teleautonomous system may take commands from multiple command sources and that these command sources produce commands at rates that differ from the rate of the telerobot dictates that commands

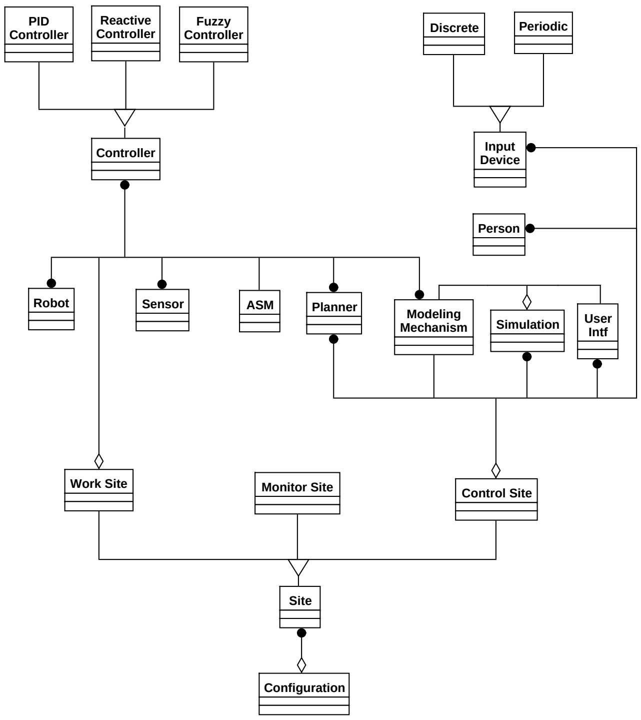

Figure 1: Basic teleautonomous system model, showing object class generalizations and aggregations

must be buffered. The command buffers objects contain the task commands pending evaluation, as well as methods to check preconditions on commands, to add or delete commands, and process requests for commands.

Each task command is also an instantiation of a object. The Command object class contains methods for performing a variety of sequencing operations, such as successor, predecessor, conditional execution, branching, and iteration. Control scope is provided by special Command objects which delimit the scope of conditional and iteration operations. To add other sequencing capabilities simply requires the addition of new methods to the Command object class.

3.2 Intelligent use of system state

Reflex action support is provided by *Event Recognizers,* which may asynchronously produce task commands or command sequences. These commands may produce motion, change shared control mode, flush command buffers, or modify other system parameters. The Event Recognizers have access to all system parameters, and may contain memory for recognition of trends or previous states.

Depending on the shared control mode, reflex commands may subsume control-site task commands, as occurs in MOTES. However, other shared control modes may be dynamically defined which allow reflex control of certain degrees of freedom, while preserving teleoperator control of others. Therefore, error recovery and retry are facilitated.

Module: asm Sheet: 1: 1 Panel: (0,0) Time: Tue Dec 13 23:13:50 1994

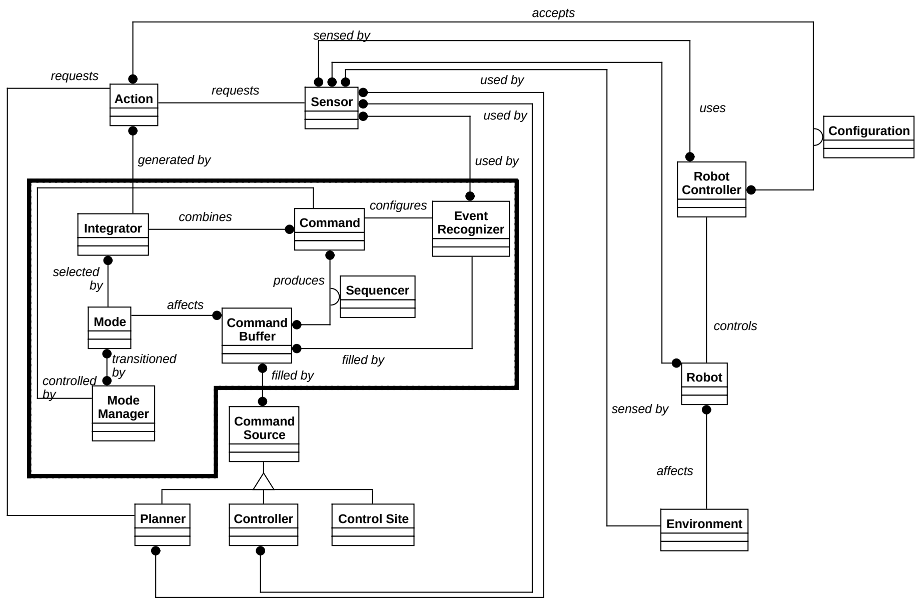

Figure 2: Action selection mechanism detail. The ASM is enclosed in the heavy outline.

3.3 User-configurable action selection

We have extended Hayati's mixing matrix capability to accommodate an arbitrary number of command sources. An Integrator object is instantiated for each device (robot, camera, light) which may be controlled remotely. Each Integrator contains a table of valid task commands. This table is updated by the Command Buffer objects, which add Command objects as their preconditions become true and remove them when they have expired. The shared control mode of each Integrator is represented by a Mode object. The current Mode object is passed to the Integrator by the Mode Manager, which performs transitions between shared control modes.

3.4 Support for reactive control

In order support reactive control, we have provided direct links between sensors and controller objects. Controller outputs are directed into command buffers in a similar fashion as other task commands. As commands enter the command buffer, they are first checked for eligibility to execute. The commands received from reactive controllers receive highest priority in this regard, and are passed immediately to the Integrator.

Due to space limitations, we cannot describe here the details of the dynamic and functional models which were constructed for teleautonomous systems. Suffice it to say that these models provided a clearer picture of the architectural requirements for teleautonomy, and eased the design and implementation phases. Further details of the object, dynamic, and functional models will be documented in [17]

4 System Design – Overview of Action Selection

The design of the action selection mechanism is dictated by the analysis given in the previous section. This section will describe details of the system design. Figure 3 shows the relationships between the components of

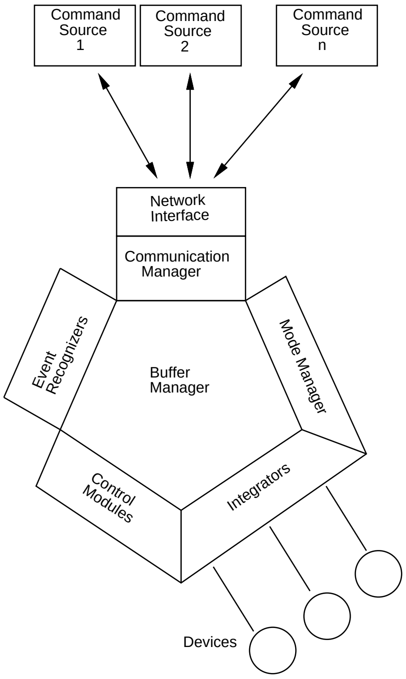

Figure 3: Action selection mechanism block diagram. Components which directly communicate are shown as connected regions.

the ASM design.

In previous sections, motivation has been given for a structured approach to action selection for teleautonomous systems. An object-oriented analysis has been performed, yielding a small set of objects which together may be used to perform action selection. In the following sections, a detailed design of the action selection mechanism will be developed. This design, and the resulting implementation, is named ASIAGO – A System for Intelligent Action Generation for teleOperation. The implementation is written using the LynxOS operating system, which provides multi-threaded realtime processing.

Action selection in ASIAGO is a process which transforms a series of inputs from various sources into streams of action requests. These streams of action requests are directed towards devices which perform the actions. The object classes needed for this process include the input command and action request, the integrator, and the mode. Objects which may optionally influence the action selection process include the event recognizers, homogeneous transforms, and transformation functions. Each of these objects will be reviewed before describing the action selection process in detail.

4.1 Buffer Manager

The Buffer Manager is an object responsible for organizing and maintaining Command objects. Buffers are created by explicit request. Requests for connection to the ASM are issued by either the program which will be generating task commands, or by a configuration utility. Buffer configuration may be done upon buffer creation, or later using a configuration command. Parameters of the buffer which may be changed include buffer size, immediate vs. sequenced, and command lifespan.

4.2 Communication Manager

The Communication Manager provides an interface between the communications medium and the ASM. Specifically, it receives task and configuration commands from the control site and passes them to the Buffer Manager. The Communication Manager also issues status data from the ASM to the control site.

4.3 Mode

In ASIAGO, the Mode is a representation of a method used for command processing and action generation to control a device. The Mode determines what actions are generated by the current set of valid commands. There is the potential for confusion here because the word 'mode' is a common term in command and technical usage. In order to make the distinction clear, Mode will be capitalized when it refers to the action generation procedure.

Within ASIAGO, the Mode Manager is in charge of storing and processing Modes. There may be many Modes defined — however, for each device there is only one Mode active at any given moment, which is called the *Current Mode*. Other defined Modes are inactive, and are available when they are needed.

4.4 Inputs

Inputs to the action selection process are supplied by the Buffer Manager, which buffers data which may come from sources external or internal to ASIAGO. Each buffer β provides a particular type of command from a particular command source. These commands may actually affect multiple degrees of freedom, such as a cartesian motion command which provides commands for the x, y, and z axes.

External command sources, such as hand controllers or planners, may be located at the work site or at a control site. Externally generated commands are issued to ASIAGO via the network interface. They are received by the ASIAGO Communication Manager which routes them to the appropriate module within ASIAGO.

Inputs may also be generated internally in response to events occurring within ASIAGO, as a result of event recognition, or as a function of external inputs. These commands are issued directly to the Communication Manager, bypassing the network interface. Otherwise, they are handled in exactly the same fashion as externally generated commands. Any command which may be issued to ASIAGO externally may be generated internally, and vice-versa. This feature enables event recognizers to perform dynamic configuration, mode changes, or issue motion requests.

4.4.1 Force/Motion Commands

Force/Motion Commands cause relative or absolute positioning of an actuator or cause a change in the amount of force the robot applies to the environment. As described above, they may be generated externally or internally. For simplicity, we refer to these commands simply as 'motion commands.' External sources of motion commands could be hand controllers, motion planners, or off-line programming systems. These external commands are buffered by ASIAGO for use in the action selection process.

Internally, motion commands may originate from event recognizers or from transformation functions. Transformation functions are invoked on-the-fly by integrators to produce derived commands as a function of external inputs. This is a useful capability because it allows for efficient implementation of reactive control or for resolution of redundant degrees of freedom.

4.4.2 Status Commands

Status commands are treated by ASIAGO in the same fashion as motion commands. That is, they are buffered and may be used as input to the integrator. Status commands may represent sensor readings (either externally produced, or posted by a device interface) or software events (such as 'program started' or 'sensor failure'). Event recognizers may make use of status commands to generate mode change commands.

The topic of command generation internal to ASIAGO has been discussed with regard to event recognizers. However, there are other sources for internal 'signals' which are useful — primarily for feedback to the operator and as input to event recognizers. For instance, creation or deletion of data buffers by the Buffer Manager triggers generation of internal status commands. These may be used to trigger event recognizers.

4.4.3 Configuration Commands

Configuration commands are commands which provide dynamic reconfiguration of ASIAGO and of the devices under its control. These commands are needed to define Modes, configure devices, and to change ASIAGO operating parameters. Configuration commands are asynchronous commands which are issued by the operator.

Commands which perform configuration of the devices which are under control of ASAGIO are called Device Configuration Commands (DCC). These commands are buffered by ASIAGO just like motion or status commands, and are issued to the device interface by the integrator. Because the data within a DCC is not interpreted by ASIAGO, it is the responsibility of the system designer to ensure that the device interface is capable of processing the DCC.

Control of ASIAGO's internal functions is also performed via configuration commands. One class of internal configuration is that of Buffer Manager control. These commands may be directed toward individual buffers or toward all buffers containing a particular command type or from a particular source. Typical buffer operations include 'flush buffer' or 'delete buffer'.

Another form of internal configuration takes place within the Mode Manager. Modes and mode transitions may be defined and stored, and mode change commands may be processed via configuration commands. These topics will be discussed at length in the remainder of this chapter.

4.5 Integrator

The Integrator is responsible for taking a set of commands from different buffers and integrating them to produce an action request. Each device to be controlled is assigned a single integrator, and this association remains in effect at all times. The integrator is a generic thread within ASIAGO — the specific functionality of the integrator is governed by the Current Mode.

The fact that the integrator is a generic module which is instantiated for each device has three benefits. First, reliability is improved, since new modes or device types do not require programming modifications. Validation of the integrator code is a one-time event and is not affected by any Mode the operator may specify. Second, system performance analysis is decoupled from the device or mode that is used. Performance measures such as processor and memory usage, response time, and loading characteristics (how the system behaves as more inputs are added) may be measured independent of the control Mode. Third, flexibility is improved, since the generic integrator may be used with a suitable Mode to perform tasks which have not yet been foreseen.

4.6 Transformation functions

Transformation functions allow inputs to the integrator to be derived from commands from one or more command buffers. For example, a reactive controller could be constructed by transforming sensor readings into motion commands via a transformation function. This design would be very efficient since all processing could be done within ASIAGO. Transformation functions are a very useful construct because they provide a general interface for routines for kinematics, dynamics, safety, and sensor fusion. Transformation functions are useful for standardizing data from different frames of reference or configuration spaces. Such standardization is necessary for command integration, and so a generalized technique for its performance is a necessity.

Transformation functions are subprogram modules which are linked into the ASIAGO executable. These modules are supplied by the user; however, several useful transformation functions are supplied with ASIAGO. Transformation functions are discussed in detail in Section 8.

Transformation functions are denoted by $\mathcal{F}$.

4.7 Homogeneous Transforms

In most cases, transformation functions are not needed, because inputs to the integrator can be taken directly from buffered commands. However, there may be situations where these commands are expressed in a frame of reference different from that which is expected by the integrator. Rather than pass these commands through a transformation function, the integrator may simply apply a homogeneous transform to the commands. This is a more efficient technique, and as will be shown later, has the added advantage of increased flexibility because the homogeneous transform is part of the Mode which can be changed at will. To accomplish a similar functionality using transformation functions, a different transformation function for each possible homogeneous transform would have to be linked into ASIAGO.

Homogeneous transformations are denoted by T.

4.8 Event Recognizers

Event Recognizers are used to trigger Mode changes or to issue other commands automatically into ASIAGO. To do this, event recognizers monitor the input buffers for specific data patterns or conditions which indicate situations for which new Modes or command sequences should be issued. Event recognizers are user-supplied subprogram modules which are linked into ASIAGO.

4.9 Action Selection Data Flow

Figure 4 provides an overview of the dataflows associated with action selection. Beginning with the Buffer Manager, we may trace the flow: Input commands arrive at the Buffer Manager via the Communication Manager. These commands are buffered, and are provided to Event Recognizers and to the integrator. Before arriving at the integrator, homogeneous transforms may be applied to certain types of input data in order to place them into a common reference frame. Additionally, the input data may be passed through transformation functions in order to generate derived commands.

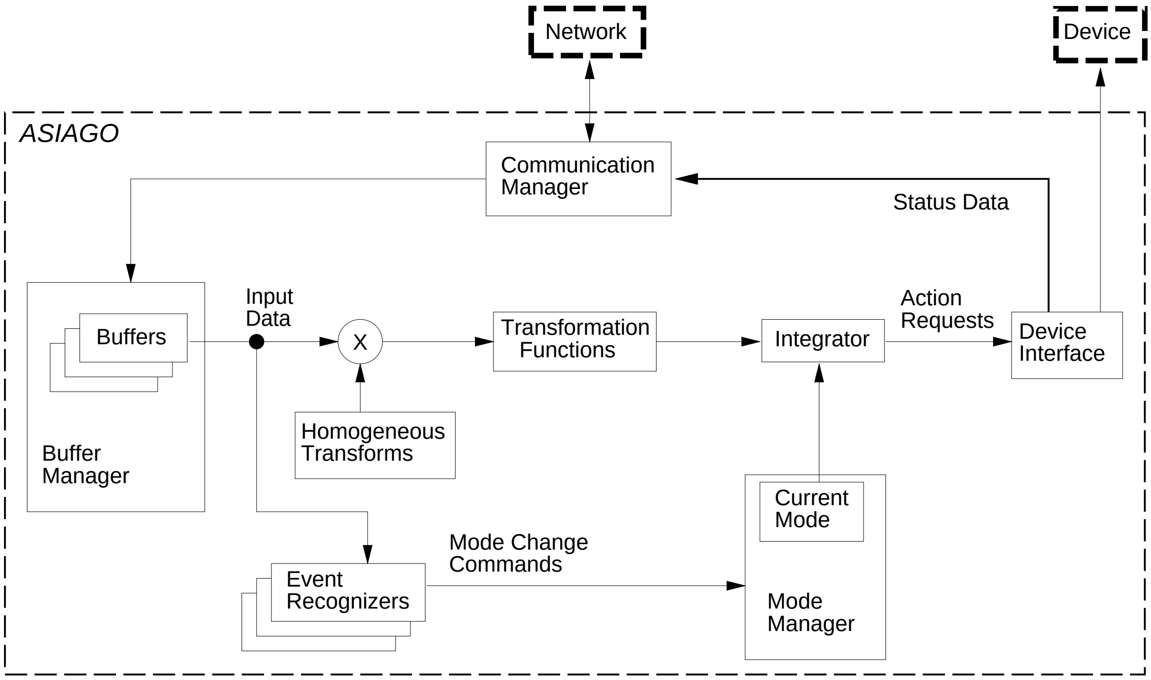

Figure 4: Simplified overview of dataflow associated with the action selection process.

Once the inputs have arrived at the integrator, action requests are created according to the Current Mode. These action requests are then directed to the device interface, and are then issued to the device. Status data from the device is gathered by the Communication Manager, and may be posted to the Buffer Manager and/or distributed externally to ASIAGO via the network. Input data is received by the Communications Manager from the device interface and the network.

The Current Mode is supplied by the Mode Manager, which is configured by commands issued by Event Recognizers.

Each of these objects will be discussed in detail in the following sections.

5 Devices and Device Interfaces

5.1 Devices

The concept of the *device* is central to ASIAGO, because without devices there would be no need to solve the action selection problem. Devices are pieces of hardware which respond to commands issued via software. The devices themselves may be interfaced to ASIAGO in various ways — through shared memory, parallel or serial connections, or network interfaces. In all cases, the device must be capable of performing some action upon receipt of a command. In addition, a device may provide data to ASIAGO. In the case of a sensor, this is the primary duty of the device.

Device types include manipulators, lights, cameras, grippers, vision systems, force/moment sensors, and range sensors. When one pictures a robotic system, the first device that comes to mind are active devices such as manipulators. However, not all devices are active or affect the environment. For example, cameras do not typically affect the environment. Yet a camera system will typically have controls for focus, zoom, and aperture.

5.2 Device Interfaces

In order to interface these devices to ASIAGO, a standardized *device interface* is required. The each device is coupled with a device interface to provide a method of passing commands from ASIAGO to the device. The device interface also provides a way for ASIAGO to obtain feedback from the device.

Device interfaces are user-supplied subprogram modules which are linked into the ASIAGO executable. The device interface may simply provide a shared memory interface to another device control program or it may interface to the device directly through a device driver. In any case, the device interface is linked into ASAIGO at compile time, and is referenced at execution time by name through a lookup table.

The device interface concept is an important feature of ASIAGO's design. Without a standardized interface, it would be difficult to design a generic integrator module. The device interface provides flexibility since new actions or devices may be installed without making modifications to ASIAGO. Also, the device interface operates at the application level, not the operating system. Thus, ASIAGO is insulated from variations in device drivers or operating system versions.

All device interfaces used in ASIAGO are described in the devices configuration file. This file contains the name of the device interface (used for lookup of the code at runtime), the possible actions performed by the device, the kinds of status data which the interface provides, the type of status updates (synchronous or asynchronous), and the rate at which status data is provided (if applicable). The list of possible actions forms an ordered list called the Device Action List (DAL).

Status data from a device interface is processed much like commands received from the network. When a device interface is initialized, it informs the Communication Manager of the status data it will be providing. For asynchronous status data updates, the device interface issues the status data to the Communication Manager, which in turn passes the data to the Buffer Manager. If the device interface provides status data synchronously, the Communication Manager is informed of the rate at which the device interface should be queried. The Communication Manager will then query the status data from the interface at the specified interval and post it in a buffer via the Buffer Manager.

5.3 Actions

The various commands issued to devices by ASIAGO are collectively called *actions*. The action vector $\mathbf{a}_i$ is defined as a ordered vector of commanded values for each controllable parameter supported by device interface i. The term *degrees of freedom* will be used to refer to the controllable parameters of the device interface. The number of degrees of freedom is the dimension of the space of controllable parameters. If the number of degrees of freedom for device interface i is d, then $\mathrm{card}(\mathbf{a}_i) = d$, where the operator $\mathrm{card}(\mathbf{a})$ returns the cardinality of the vector $\mathbf{a}$. If there is not a command for a particular degree of freedom, that element a will contain a null command which signals the device interface to leave that degree of freedom unaffected. The order of the entries in $\mathbf{a}$ corresponds to that of the DAL.

field	*description*
name	Symbolic name of integrator
status	Flag indicating integrator running, paused, or exited
threadID	Thread ID for integrator
threadAttr	Thread attributes for integrator
mode	Pointer to current Mode
integAGC	Action Generation Criteria selector
cb	Pointer to control block
integPeriod	Period of action generation in AGC_PERIODIC

Table 1: Contents of integrator descriptor record

6 Integrator

6.1 Integrator initialization

Each device to be controlled by ASIAGO is associated with an instance of generic integrator, which runs as a separate thread. As previously discussed, this design provides flexibility and reliability since the same software module is used for all control modes. Each integrator is, in turn, associated with a descriptor record. A configuration file read by ASIAGO is used to store the attributes of each integrator into its descriptor record. This file contains a symbolic name for the integrator (such as *puma*) a reference to the device interface code, a list of the ASIAGO datatypes which the integrator recognizes, and the action generation criteria (AGC). This information is placed into a descriptor record associated with each integrator.

Each integrator is assigned a control block which is used for communication and synchronization between the integrator and other ASIAGO threads such as the Buffer Manager, device interfaces, and the Communication Manager. This control block is stored in the integrator's descriptor record.

Integrators run as separate threads which are initiated by the Communication manager. Once the descriptor records for all integrators have been built, a separate thread is started for each integrator. Each integrator is passed a reference to its descriptor record upon startup. The control block may then be accessed by the integrator for use in inter-thread communication. The structure of the descriptor records and thread startup is shown in Figure 5. A summary of the fields present in the integrator descriptor record is shown in Table 1.

6.2 Integration Control

As shown in Table 1, each integrator descriptor record contains a reference to a control block. The control block is a data structure which is protected by mutex operators and is shared by the integrator and the Communication Manager. Dynamic configuration of the integrators is performed via the control block. Integration may also be paused, enabled or terminated.

A summary of the fields present in the control block is shown in Table 2.

6.3 Triggering Action Generation

The integrator has three techniques for determining when a new action should be given to the controlled device. These techniques are called Action Generation Criteria (AGC). In the first AGC, called AGC_REQUEST, the

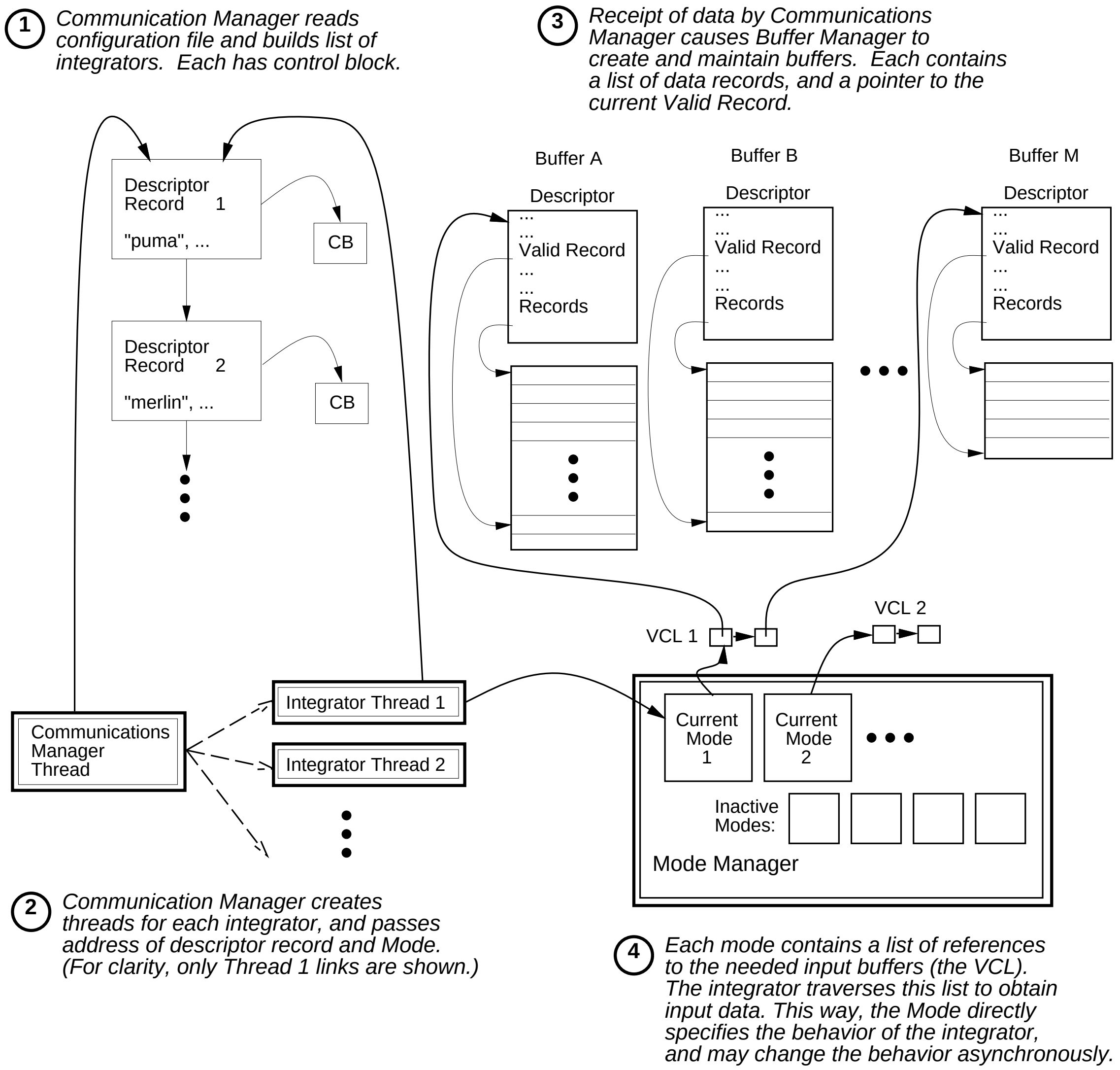

Figure 5: Structure of descriptor records, threads, and Modes.

field	*description*
controlBlkIndex	Unique index labeling this control block
controlBlkCommand	Opcode of command issued to integrator
controlBlkMut	Mutex operator for control block
controlBlkCond	Condition variable for control block
controlBlkDataPtr	Pointer to data associated with command

Table 2: Contents of control block

name	*description*	*initiator*
AGC_REQUEST	actions requested asynchronously	device interface
AGC_PERIODIC	actions generated synchronously	comm. manager
AGC_INFORM_UPDATE	integrate when buffer is updated	buffer manager

Table 3: Action generation criteria

device may make a request for a new action. This technique assumes that the device code is running in a separate thread of control, as occurs in RCCL. The integrator's control block is used by the device to signal that an action should be generated. When the device is ready for a new action, it places a **`CB_DEV_REQ`** command into the *controlBlkCommand* field. This causes the integrator to unblock and process the request.

Another common criteria for action generation is based on periodic action updates. This AGC, called AGC_PERIODIC, causes the integrator to generate actions at a fixed interval. This interval is initialized via an integrator configuration initialization file, and may be updated by a configuration command. The period of an integrator running under the AGC_PERIODIC scheme is stored in the *integPeriod* field of the integrator's control block.

Under the AGC_PERIODIC scheme, the communications manager is responsible for performing timing functions. To initiate action selection, the Communication Manager places a **`CB_PERIODIC_REQ`** command into the *controlBlkCommand* field. This causes the integrator to unblock and process the request. To enable this type of action selection, the integrator must schedule the timer interrupt with the communications manager by calling the *commMgrScheduleTimer* function. This function accepts the control block reference and a period.

The third AGC is called AGC_INFORM_UPDATE. In this protocol, the buffer manager informs the integrator when updated data becomes available in one or more designated buffers. This mode is useful for implementing tasks such as station-keeping or relative motions. Consider a situation in which the robot is to maintain a fixed position relative to a moving object. This object's pose is sensed and stored in a buffer, as is the relative position that the robot is to maintain. This relative position is unchanging, and yet the robot's position may need to be constantly updated in order to maintain the proper relationship to the object. In this situation, the AGC_INFORM_UPDATE protocol could be used to generate new position updates for the robot only when updates are made to the position of the object.

The AGC_INFORM_UPDATE protocol has the potential for inefficiency, since each sensor update could generate inter-thread communication within ASIAGO. Therefore, some care has been taken to make the implementation as efficient as possible. Each buffer has an associated list of 'interested parties,' such as integrators or event recognizers. These lists are accessed via Buffer Manager entry points bufferMgrRegisterInterest and bufferMgrRemoveInterest. These routines take as input a description of the buffer and a pointer to the control block of the calling party. This control block reference is then placed into the corresponding buffer's descriptor record. When an update is made to a buffer, the list of control block references is traversed, and a CB_DATA_READY command is placed into the *controlBlkCommand* field. This causes each interested party to wake up and process the new data. Table 3 summarizes the action generation criteria.

6.4 Action Selection Algorithm

The result of the action selection process is an action vector $\mathbf{a}$ which modifies one or more controllable parameters of the device interface. There are two steps which must be performed for action selection. First, the inputs to the action selection process must be gathered. This involves the retrieval of commands from Buffer Manager buffers, transformation of commands into a common reference frame, and calculation of derived commands. Once the inputs have been collected, the second step may be performed — actions are derived from the inputs. We now consider the latter step.

Consider the problem of generating an action vector for a particular device interface from a set of input commands. Let s be the number of input sources, and $\{\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_s\}$ be the set of input commands. Each $\mathbf{c}_i$ affects various elements of the action vector $\mathbf{a}$. Recall that $\mathrm{card}(\mathbf{a}) = d$, the number of degrees of freedom supported by the device interface. We arrange the c_i into a matrix:

$$\mathbf{C} \;=\; \begin{bmatrix} \mathbf{c}_1 \\ \mathbf{c}_2 \\ \vdots \\ \mathbf{c}_s \end{bmatrix} \;=\; \begin{array}{c} \begin{bmatrix} c_{11} & c_{12} & \ldots & c_{1d} \\ c_{21} & c_{12} & \ldots & c_{1d} \\ \vdots & \vdots & \ddots & \vdots \\ c_{s1} & c_{s2} & \ldots & c_{sd} \end{bmatrix} \\ \| \\ \begin{bmatrix} \tilde{\mathbf{c}}_1 & \tilde{\mathbf{c}}_2 & \ldots & \tilde{\mathbf{c}}_d \end{bmatrix} \end{array} \tag{1}$$

If $\mathbf{c}_i$ does not contain a command for degree of freedom j, a null value is used for c_{ij}.

Column j of $\mathbf{C}$, labeled $\tilde{\mathbf{c}}_j$, contains all commands for the jth degree of freedom supported by the device interface. Thus, $\mathbf{C}$ is a $s \times d$ matrix of input commands, which will be called the *input matrix*. The method used to build $\mathbf{C}$ will be described in Section 7.

Given an input matrix $\mathbf{C}$, we may define a matrix $\mathbf{M}$ called the *blending matrix*. The blending matrix is similar in purpose to the mixing matrix described in [6]. A blending matrix $\mathbf{M}$ is a $s \times d$ matrix of coefficients m_{ij} which regulate the contribution of inputs towards generating a particular degree of freedom of the action vector.

Each of the s rows of $\mathbf{M}$, labeled $\mathbf{m}_i$, corresponds to an input to the integrator, and the d columns represent degrees of freedom supported by the device interface. The total contribution of all inputs for a degree of freedom must total 100%, that is, for a given column j,

$$\sum_{i=1}^{s} m_{ij} = 1. \tag{2}$$

All coefficients must be in the range $0 \leq m_{ij} \leq 1$.

$\mathbf{M}$ is used in conjunction with $\mathbf{C}$ to calculate the action vector $\mathbf{a}$ (the symbol $\cdot$ indicates the inner product).

$$\mathbf{a} = [\mathbf{m}_1 \cdot \tilde{\mathbf{c}}_1, \mathbf{m}_2 \cdot \tilde{\mathbf{c}}_2, \ldots, \mathbf{m}_d \cdot \tilde{\mathbf{c}}_d] \tag{3}$$

or, equivalently:

$$\mathbf{a} = \mathrm{diag}(\mathbf{M}^T \mathbf{C}) \tag{4}$$

7 The Mode

At any given moment, a Current Mode is associated with each integrator. Other Modes may be inactive – these are stored by the Mode Manager for later use. The Current Mode is referenced by an integrator through a pointer in its descriptor record. This reference allows the Mode Manager to update the Current Mode asynchronously for any integrator. See Figure 5.

Modes are created and stored by the Mode Manager upon receipt of a mode definition command. Modes may be specified by the operator during system operation or a-priori. Each Mode definition contains a user-specified signature which labels the Mode. This signature may be later used to reference the Mode for selection or deletion. The signature is also used in the user interface to represent the Mode symbolically.

7.1 Mode Representation

As stated above, Mode Specification Commands (MSC's) are processed by the Mode Manager. Upon receipt of an MSC, the Mode Manager must construct an internal representation of the specified Mode. This internal representation may differ from the representation used in the MSC. The internal representation of Modes used by the Mode Manager must allow efficient action generation, mode interpolation capability, consistency checking, and have modest storage requirements.

We have already described one component of the Mode specification: the blending matrix. The second component of the Mode representation, the *input map* $\mathbf{\Delta}$, describes how input buffers are used to construct the matrix $\mathbf{C}$ which contains the data to be integrated. Construction of $\mathbf{C}$ is performed by a built-in input mapping function $\mathcal{G}$ which accepts an input map $\mathbf{\Delta}$ and returns $\mathbf{C}$. $\mathbf{\Delta}$ is a vector of cardinality s,

$$\Delta = [\delta_1, \delta_2, \ldots, \delta_s] \tag{5}$$

where

$$\delta_i = \{\mathcal{F}, \{\beta_1, T_1\}, \{\beta_2, T_2\}, \ldots, \{\beta_k, T_k\}\} \tag{6}$$

Thus, $\mathcal{G}(\delta_i) \rightarrow \mathbf{c}_i$ and $\mathcal{G}(\mathbf{\Delta}) \rightarrow \mathbf{C}$.

Recall from Section 4.4 that each buffer β provides a particular type of command from a particular command source. These commands may actually affect multiple degrees of freedom, such as a cartesian motion command which provides commands for the x, y, and z axes.

In $\mathbf{\Delta}$, each reference to a command buffer β_i may be accompanied by a homogeneous transform T_i. If a homogeneous transform is assigned to a particular input buffer, all inputs from that buffer will be multiplied by the given transform before it is passed to the transformation function $\mathcal{F}$. The homogeneous transforms allow data which is with respect to an arbitrary user frame to be transformed into the frame expected by the integrator. This may only be done for buffers which contain geometric data. If no transformation is required, the identity matrix may be specified.

$\mathcal{F}$ is a transformation function which takes a set of input parameters and returns a command:

$$\mathcal{F}(\beta_1 T_1, \beta_2 T_2, \ldots, \beta_k T_k) \rightarrow \mathbf{c} \tag{7}$$

For a particular δ_i, application of $\mathcal{F}$ results in a vector $\mathbf{c}_i$ which is used as the ith row of $\mathbf{C}$. If no transformation function is required, a null value is specified in place of $\mathcal{F}$. In such cases, $k \equiv 1$; input i to the integrator is taken

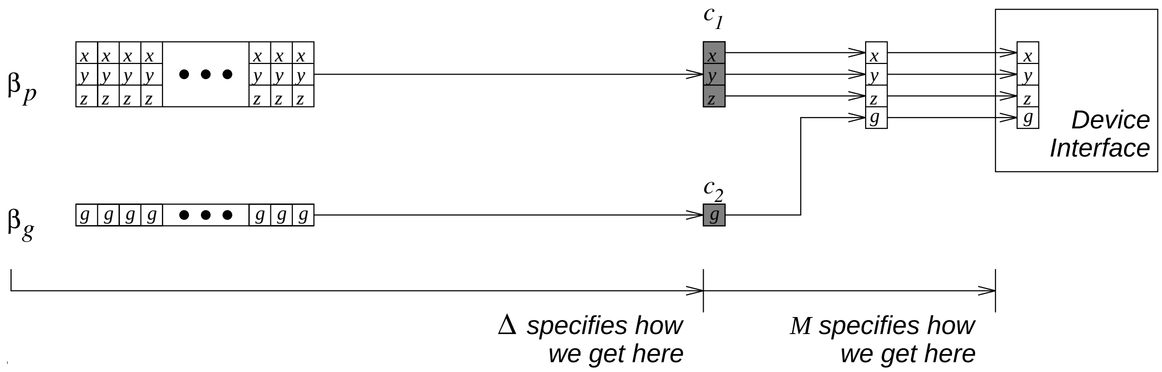

Figure 6: Mode $\mathcal{M}_1$.

directly from the buffer specified by β_1:

$$\mathbf{c}_i = \beta_1 T_1 \tag{8}$$

A Mode $\mathcal{M}$ is represented internally by the Mode Manager with a blending matrix $\mathbf{M}$ and an input map $\boldsymbol{\Delta}$. The labels for the blending matrix and input map may be given a subscript of the name of the Mode to which they belong, e.g.

$$\mathcal{M}_S = \{\mathbf{M}_S, \boldsymbol{\Delta}_S\}. \tag{9}$$

The following two examples show how $\mathbf{M}$ and $\boldsymbol{\Delta}$ are constructed from a Mode specification.

Example 1 First suppose that the integrator is running under the following conditions:

- The device interface supports absolute position control of the $x, y,$ and z axes (with respect to the base frame of the robot), as well as gripper open/close commands.
- Absolute position commands for x, y, z (with respect to the base frame of the robot) are available in buffer β_p. These commands are supplied by a manual controller.
- Gripper commands are available in buffer β_g.

Now suppose a Mode $\mathcal{M}_1$ is to be defined with the following properties:

- Absolute positioning is to be controlled by commands from β_p.
- Gripper control is to be performed by commands from β_g.

Figure 6 illustrates this example.

First, the input mapping $\boldsymbol{\Delta}_1$ is constructed.

$$\delta_1 = \{\bullet, \{\beta_p, \bullet\}\} \tag{10}$$

$$\delta_2 = \{\bullet, \{\beta_g, \bullet\}\} \tag{11}$$

$$\boldsymbol{\Delta}_1 = \{\delta_1, \delta_2\} \tag{12}$$

The symbol $\bullet$ is used as a placeholder to indicate that the given field is unused. δ_1 indicates that no transformation function is needed, so input c_1 is to come directly from buffer β_p. Also, no homogeneous transform is needed since the input data in β_p is already in the correct frame. δ_2 is given similarly.

The dimensions of $\mathbf{M}_1$ will be 2×4, since there are two inputs ($\mathrm{card}(\mathbf{\Delta}_1) = 2$) and the device interface has 4 degrees of freedom. $\mathbf{M}_1$ is given below, each column labeled with the device degree of freedom.

$$\begin{array}{cc} & \begin{array}{cccc} X & Y & Z & \text{Grip} \end{array} \\ \mathbf{M}_1 = & \left(\begin{array}{cccc} 1 & 1 & 1 & \bullet \\ \bullet & \bullet & \bullet & 1 \end{array} \right) \end{array} \tag{13}$$

Thus, $\mathcal{M}_1$ may be defined as

$$\mathcal{M}_1 = \{\mathbf{M}_1, \mathbf{\Delta}_1\}. \tag{14}$$

Example 2 We may now build a more complicated mode, $\mathcal{M}_2$. Assume the integrator using $\mathcal{M}_2$ is running under the same conditions as in the previous example, with the following additional conditions:

- Buffer β_q contains absolute position commands for x, y, z (with respect to a frame located at $(10, 10, 0)$ in the base frame of the of the robot). These commands are supplied by an off-line planner.
- Buffer β_m contains range sensor data from a laser range finder in the wrist of the robot. The sensor returns the distance d_m from the end frame of the robot to the work environment, along the z axis of the end frame.
- Buffer β_s contains the current x, y, z position of the robot, derived from encoder readings.
- A transformation function $\mathcal{F}_r$ is available. $\mathcal{F}_r$ takes as input a desired distance d_d, a measured distance d_m, and a cartesian coordinate X_s. When passed the current position of the robot, $\mathcal{F}_r$ will return an absolute position which is the desired distance away from the environment. Specifically:

$$\mathcal{F}_r(d_d, d_m, (x_s, y_s, z_s)) \rightarrow (x_s, y_s, z_s + (d_d - d_m)) \tag{15}$$

Under the given conditions, many control modes are possible. For the sake of this example, suppose that the operator decides to manually control the x position of the robot using the handcontroller (which supplies buffer β_p), to control the y position using the off-line planner (buffer β_q), and to allow the z position to be controlled reactively. This Mode, $\mathcal{M}_2$, would have the following properties:

- The robot's x position is to be given by the x value in input buffer β_p.
- The robot's y position is to be given by the y value in input buffer β_q. This position must be transformed into the robot's base frame.
- The robot's z position is to be maintained 20 units away from the environment, based on range sensing.
- The gripper is to be controlled by commands from β_g.

Figure 7 illustrates this example. Note that there is data available which will be discarded according to the Mode specification. For example, although $\mathcal{F}_r$ returns x, y, and z, only the z value will be used.

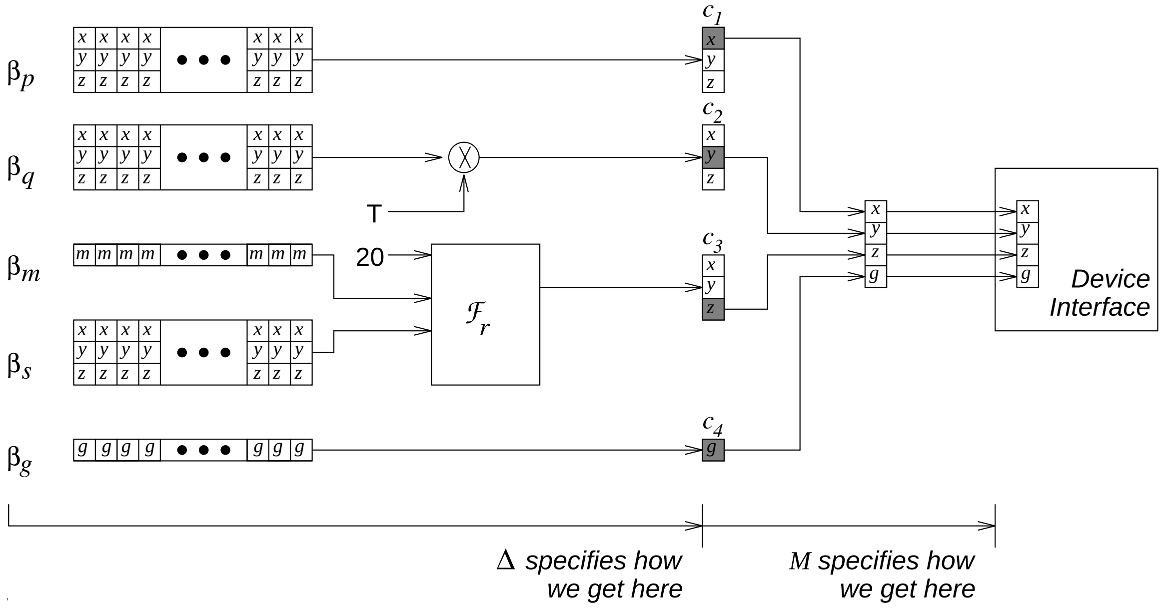

Figure 7: Mode $\mathcal{M}_2$.

As in the previous example, the input mapping $\mathbf{\Delta}_2$ is constructed first. In this case card$(\mathbf{\Delta}_2) = 4$.

$$\begin{aligned}
\delta_1 &= \{\bullet, \{\beta_p, \bullet\}\} && (16)\\
\delta_2 &= \{\bullet, \{\beta_q, \begin{bmatrix} 1 & 0 & 0 & -10 \\ 0 & 1 & 0 & -10 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}\}\} && (17)\\
\delta_3 &= \{\mathcal{F}_r, 20, \{\beta_m, \bullet\}, \{\beta_s, \bullet\}\} && (18)\\
\delta_4 &= \{\bullet, \{\beta_g, \bullet\}\} && (19)\\
\mathbf{\Delta}_2 &= \{\delta_1, \delta_2\, \delta_3, \delta_4\} && (20)
\end{aligned}$$

Note that δ_3 lists each parameter to $\mathcal{F}_r$, including a literal value of 20 for the first parameter.

The dimensions of $\mathbf{M}_2$ will be 4×4, since there are now four inputs and the device interface has four degrees of freedom. $\mathbf{M}_2$ is given below with each column labeled as before.

$$\mathbf{M}_2 = \begin{array}{c} \begin{matrix} X & Y & Z & \text{Grip} \end{matrix} \\ \begin{pmatrix} 1 & 0 & 0 & \bullet \\ 0 & 1 & 0 & \bullet \\ 0 & 0 & 1 & \bullet \\ \bullet & \bullet & \bullet & 1 \end{pmatrix} \end{array} \qquad (21)$$

Thus, Mode $\mathcal{M}_2$ may be defined as

$$\mathcal{M}_2 = \{\mathbf{M}_2, \mathbf{\Delta}_2\}. \qquad (22)$$

Example 3 Now that we have defined some Modes, we may proceed to generate an action vector. We will use Mode $\mathcal{M}_2$ as the current mode. Assume that at time t, the input buffers described in the previous example contain the following data:

$$\beta_p = (x = 10, y = 15, z = 20) \tag{23}$$

$$\beta_q = (x = -12, y = -10, z = 100) \tag{24}$$

$$\beta_m = (d_m = 19) \tag{25}$$

$$\beta_s = (x = 100, y = 100, z = 25) \tag{26}$$

$$\beta_g = (g = 1) \tag{27}$$

The first step is to build the input matrix $\mathbf{C}$ using $\mathbf{\Delta}_2$. Each $\mathbf{c}_i$ must be contain four elements $(x, y, z, gripper)$ corresponding to the degrees of freedom of the device interface, as described in the previous example. In cases where a buffer does not supply commands for all degrees of freedom, a null command is placed in the appropriate position in the command vector.

$\mathbf{c}_1$ is constructed according to δ_1 as given in Equation 16. That is:

$$\mathbf{c}_1 = [10, 15, 20, \bullet] \tag{28}$$

where $\bullet$ represents the null command for the gripper. $\mathbf{c}_2$ is similar to $\mathbf{c}_1$, except that the data from β_q must be transformed according to the homogeneous transform of Equation 17 giving:

$$\mathbf{c}_2 = [-22, -20, 100, \bullet] \tag{29}$$

$\mathbf{c}_3$ is derived from buffers β_m and β_s, as shown in Equation 18. The contents of these buffers, as well as the constant value of 20 are passed to transformation function $\mathcal{F}_r$ which returns $\mathbf{c}_3$.

$$\mathcal{F}_r(20, 19, (100, 100, 25)) \rightarrow (100, 100, 25 + (20 - 19)) \tag{30}$$

$$\mathbf{c}_3 = [100, 100, 26, \bullet] \tag{31}$$

As shown by Equation 19, the gripper command is supplied by β_g:

$$\mathbf{c}_4 = [\bullet, \bullet, \bullet, 1] \tag{32}$$

The input matrix $\mathbf{C}$ may be given as

$$\mathbf{C} = \begin{bmatrix} \mathbf{c}_1 \\ \mathbf{c}_2 \\ \vdots \\ \mathbf{c}_s \end{bmatrix} = \begin{array}{c} \begin{bmatrix} 10 & 15 & 20 & \bullet \\ -22 & -20 & 100 & \bullet \\ 100 & 100 & 26 & \bullet \\ \bullet & \bullet & \bullet & 1 \end{bmatrix} \\ \| \\ \begin{bmatrix} \tilde{\mathbf{c}}_1 & \tilde{\mathbf{c}}_2 & \ldots & \tilde{\mathbf{c}}_d \end{bmatrix} \end{array} \tag{33}$$

Column j of $\mathbf{C}$, labeled $\tilde{\mathbf{c}}_j$, contains all commands for the jth degree of freedom supported by the device interface. As shown in Equation 3, the action vector $\mathbf{a}$ is computed by

$$\mathbf{a} = [\mathbf{m}_1 \cdot \tilde{\mathbf{c}}_1, \mathbf{m}_2 \cdot \tilde{\mathbf{c}}_2, \ldots, \mathbf{m}_d \cdot \tilde{\mathbf{c}}_d] \tag{34}$$

In this case, $\mathbf{m}_i$ is row i of $\mathbf{M}_2$ from Equation 21.

Therefore,

$$a_1 = (10 \times 1) + (-22 \times 0) + (20 \times 0) = 10 \tag{35}$$

$$a_2 = (15 \times 0) + (-20 \times 1) + (50 \times 0) = -20 \tag{36}$$

$$a_3 = (100 \times 0) + (100 \times 0) + (26 \times 1) = 26 \tag{37}$$

$$a_4 = (1 \times 1) = 1 \tag{38}$$

and

$$\mathbf{a} = [10, -20, 26, 1]. \tag{39}$$

Thus, the robot would be commanded to the position (10, -20, 26), and the gripper would be set to a value of 1.

7.2 Specification of Modes

Mode Specifications Commands (MSC's) are a type of configuration command which provide a mechanism for defining Modes and issuing them from the control site to ASIAGO. MSC's allow M and $\mathbf{\Delta}$ to be expressed symbolically. Once received by the Mode Manager, MSC's are interpreted to build an internal repesentation of the Mode. This internal representation contains memory references to various buffers and transformation functions. The MSC must reference buffers and transformation functions by names which can be resolved by the Mode Manger using lookup tables. While the coefficients of M may be issued directly, the references to $\mathcal{F}$ and β entries in $\mathbf{\Delta}$ are given as symbolic names. Transformation functions are given such names by the programmer when they are linked into the ASIAGO executable. Command buffers are named implicitly by the data type they contain and the source of the data.

The steps taken by the Mode Manager upon receipt of an MSC are:

1. Perform error checking on MSC (See below).
2. Allocate storage for internal Mode structure.
3. Copy blending matrix coefficients into $\mathbf{M}$.
4. Create linked list of δ entries for $\mathbf{\Delta}$. For each:
   (a) Lookup transformation function by name; place reference into δ.
   (b) Copy buffer descriptions into δ, actual reference to be looked up upon Mode transition.
   (c) Copy homogeneous transforms into δ

Some limited error checking is performed by the Mode Manager upon receipt of the MSC. Error checking cannot be total because buffers referenced by a MSC may not have been created when the MSC is issued, and because the Mode is not assigned to an integrator until the Mode Change Command is issued. Error checking that is done at time of specification:

- Mode names must be unique.

- Number of rows of M must equal cardinality of $\mathbf{\Delta}$.
- Columns of M must sum to 1.0.
- Coefficients of M must be in range $0 \leq m_{ij} \leq 1$.
- Transformation functions specified by the MSC must be defined.
- Homogeneous transforms may only be applied to geometric data.

Error checking that is done at time of transition:

- Buffers listed in input map must exist.
- Number/type of columns of M must match $\mathbf{a}$

Errors are reported to the operator via the Communication Manager. A Mode specification error which is detected at specification time results in a warning message to the operator. Errors which are detected at transition time are more serious, since they possibly occur while the robot is performing a task. In such cases, an error message is issued to the operator and an automatic Mode transition is performed into a failsafe Mode (See Section 7.5).

Modes are specified in three ways:

- batch specification at ASIAGO startup (Modes stored at work site)
- batch specification at operator signon (Modes stored at control site)
- on-the-fly Mode construction.

Batch specification is performed by ASIAGO automatically upon initialization. A Mode configuration file contains zero or more MSC's stored in a human-readable form. These Modes are read and issued to the Mode Manager by the Communication Manager at system startup.

At the control site, a similar file is read by the ASIAGO user interface upon startup. The Mode specifications in this file are issued to ASIAGO, which treats them as a series of MSC's. This file has an identical format to that used at the work site.

On-the-fly Mode construction is performed via the ASIAGO user interface. Modes constructed in this manner may be issued at any time by the operator. This provides great flexibility, because the operator may use his/her judgement during a performance of a task to design a control Mode which is suited to the situation at hand.

The ASIAGO user interface may also be used to design Modes which are stored in the Mode configuration file or into separate files for later issuance.

7.3 Default Modes

As previously described, the Communication Manager reads the integrator configuration file to configure the integrator threads. This file may assign a default Mode to each integrator. The default Mode must be defined in the Mode configuration file so that it will be available regardless of the status of the connection to the control site. The Mode assignment is performed by issuing a Mode change command to the integrator upon startup.

There is a potential for a timing problem at initialization, since the default Mode may refer to buffers which have not yet been created. Therefore, the assignment of a default Mode which makes such a reference does not generate an error. The integrator will remain inactive until all needed buffers have been created, or until a different Mode is selected.

7.4 Mode Transitions

Mode transitions are performed by interpolating from one mode to another. One of ASIAGO's most powerful features is its support for interpolated Mode transitions. Interpolated transitions may be used for task handoffs between operators and for 'fading in' control sources.

The interpolation task is performed by the Mode Manager, which updates the Current Mode according to the original Mode, the new Mode, and the interpolation function. The transition between modes is transparent to the integrator. It simply references the Current Mode to generate an action — the Mode Manager asynchronously changes the Current Mode.

The transition between Modes is initiated by a Mode Change Command, which may be generated externally or internally. Triggering of transitions may be performed manually by the human operator or automatically by an Event Recognizer. Manual triggering is accomplished via a user interface by which the operator may request issuance of a Mode Change Command.

7.4.1 Immediate mode transitions

Mode transitions may be immediate or take place over a period of time. Immediate transitions are performed by the Mode Manager, which simply replaces the existing $\mathbf{\Delta}$ and $\mathbf{M}$ with those of the new Mode.

7.4.2 Non-immediate mode transitions

Non-immediate transitions are performed by the generation of intermediate Modes by the Mode Manager. Intermediate modes are constructed according to a blending value λ, $0 \leq \lambda \leq 1$, which is generated by a blending function $\mathcal{B}(t) \to \lambda$. At time t during a transition from a 'starting' Mode $\mathcal{M}_S$ to a 'finishing' Mode $\mathcal{M}_F$, the intermediate mode may be conceptualized as an interpolation between $\mathcal{M}_S$ and $\mathcal{M}_F$:

$$\mathcal{M}' = (1-\lambda)\mathcal{M}_S + \lambda\mathcal{M}_F. \tag{40}$$

where

$$\lambda = \mathcal{B}(t) \tag{41}$$

However, it is not possible to interpolate Modes directly — while the blending matrices could be interpolated (under special conditions described below), input maps can not. This is because input maps contain information which cannot be parameterized, such as references to transformation functions and input buffers. The method used to perform the blending of Modes will be discussed in detail below.

If a transition is to be non-immediate (occurring over a time t), two transitional Modes are constructed: $\mathcal{M}'_S$ and $\mathcal{M}'_F$. These transitional modes are constructed such that they have identical input maps, allowing them to be interpolated simply by interpolating their blending matrices. The sequence of Modes during the mode transition then becomes:

$$\mathcal{M}_S \rightarrow \mathcal{M}'_S \xrightarrow{t} \mathcal{M}'_F \rightarrow \mathcal{M}_F, \tag{42}$$

where $\rightarrow$ implies an immediate transition and $\xrightarrow{t}$ implies a transition over an interval of time t.

We now consider how to construct $\mathcal{M}'_S$ and $\mathcal{M}'_F$, beginning with the transitional matrices. In order to define $\mathbf{M}'_S$ and $\mathbf{M}'_F$, we first define $\mathbf{0}_S$ and $\mathbf{0}_F$ as zero matrices of the same dimension as $\mathbf{M}_S$ and $\mathbf{M}_F$, respectively. The transitional blending matrices may then be defined as

$$\mathbf{M}'_S = \begin{bmatrix} \mathbf{M}_S \\ \mathbf{0}_F \end{bmatrix} \tag{43}$$

$$\mathbf{M}'_F = \begin{bmatrix} \mathbf{0}_S \\ \mathbf{M}_F \end{bmatrix} \tag{44}$$

The transitional input map is defined as

$$\mathbf{\Delta}' = [\mathbf{\Delta}_S \mathbf{\Delta}_F] \tag{45}$$

The transitional Modes may then be defined as

$$\mathcal{M}'_S = \{\mathbf{M}'_S, \mathbf{\Delta}'\} \tag{46}$$

$$\mathcal{M}'_F = \{\mathbf{M}'_F, \mathbf{\Delta}'\} \tag{47}$$

Note that the final immediate transition from $\mathcal{M}'_F$ to $\mathcal{M}_F$ is not strictly necessary. However, $\mathbf{\Delta}$ would grow increasingly larger with each non-immediate transition without the final $\mathcal{M}'_F \rightarrow \mathcal{M}_F$ transition.

During a Mode transition, the integrator maintains two blending matrices corresponding to $\mathbf{M}'_S$ and $\mathbf{M}'_F$. At time t of the transition, the integrator calculates the current blending matrix as:

$$\mathbf{M}' = (1 - \lambda)\mathbf{M}'_S + \lambda\mathbf{M}'_F \tag{48}$$

where λ is calculated according to Equation 41.

The method used for generating the blending value produces a number of useful transitions. Possible transition specifications include:

- immediate transition
- linear interpolation between modes
- user-designed transition, specified as a bezier spline.

7.5 Failsafe Modes

A Failsafe Mode is a Mode to which the integrator will be automatically transitioned by the Mode Manager under error situations. In order to qualify as a failsafe Mode, the Mode must not reference input buffers which are filled from external sources. These Modes must only reference device interface status buffers, or internally generated buffers. Failsafe Modes are needed to handle error situations where Mode transitions have been specified which cannot be performed, due to missing inputs, or communication has been interrupted between the control and work sites. Failure of a failsafe mode is fatal, and results in a call to the device interface routine for device shutdown.

Each integrator may have only one failsafe Mode specified at a time. The default failsafe Mode for a particular integrator is listed in the integrator configuration file. The failsafe Mode may be changed by the operator using the ASIAGO user interface.

7.6 Specification of Transitions

Mode transitions are defined in advance and stored by the Mode Manager, just like Modes. Libraries of transitions suited to particular applications may be defined and loaded into ASIAGO before they are needed. Each transition is assigned a name by the operator, and this name is used to reference the transition later. When a Mode transition occurs, the Mode Change Command specifies which transition to use — the only aspect of the transition that must be specified at transition time is the duration.

7.7 Mode Performance Issues

Under typical telerobotic situations, Mode transitions may occur infrequently, or not at all. However, rapid Mode transitions may be used to perform actions such as hexapod locomotion or snake motion. In such cases, the efficiency of the Mode transition mechanism is important. Fortunately, the design provides a very efficient mechanism, for the following reasons:

- Modes are specified in advance. This saves time and bandwidth, especially in highly repetitive tasks.
- Mode transitions are specified in advance. In scenarios where Mode transitions are not instantaneous, the overhead of describing each transition is avoided.
- Integrators reference their modes via memory references. Mode transitions and specifications do not require communication to occur between the integrator and Mode Manager. The integrator does not have a copy of the Mode specification, allowing the Mode Manager to transition asynchronously between Modes. This saves overhead of message and data passing between the integrator and the Mode Manager.

An additional increase in efficiency is provided by the Buffer Manager. Each buffer maintained by the Buffer Manager contains a flag indicating when it was last updated. When the integrator is building the the command matrix **C**, only those entries originating in changed buffers need to be recalculated. Therefore, the task of gathering inputs to the integrator is made as efficient as possible.

8 Transformation functions

8.1 Structure

Transformation functions allow inputs to the integrator to be derived from other commands. For example, a reactive controller could be constructed by transforming sensor readings into motion commands via a transformation function. This design would be very efficient since all processing could be done within ASIAGO.

Transformation functions are subprogram modules which are linked into the ASIAGO executable. These modules are supplied by the user; however, several useful transformation functions are supplied with ASIAGO.

Transformation functions are referenced by a particular Mode's input map $\mathbf{\Delta}$. The transformation function accepts one or more input parameters and return a integrator input vector $\mathbf{c}_i$. See Section7.1 for further details on how the integrator uses transformation functions.

8.2 Installation

To install a transformation function, the user links the transformation function's object code into the ASIAGO executable. The transformation function then becomes available for use in a Mode specification. Installation is performed a-priori, as it involves the compilation and linking of the transformation function subprogram into the ASIAGO executable. To allow selection of transformation functions at run-time, each function is given an human-readable name. These names, the reference to the corresponding transformation function entry point, and the input and output signature of the transformation function are contained in a special lookup table. Upon linking of a transformation function, the user must insert the appropriate entry into the lookup table.

Restrictions placed on transformations are that they must be reentrant (since multiple integrators may be making use of the same function at one time), and that they do not take much processor time. The latter specification is vague, and it is left up to the programmer to ensure that the running time of the transformation function is not a large percentage of the cycle time of the device under control.

8.3 Invocation

Invocation of transformation functions is controlled by the integrators. When an action is requested from an integrator, it proceeds to build the $\mathbf{C}$ matrix as described in Section 6.4.

For a particular $\delta_i = \{\mathcal{F}, \{\beta_1, T_1\}, \{\beta_2, T_2\}, \ldots, \{\beta_k, T_k\}\}$ the integrator will invoke transformation function $\mathcal{F}$ if the Buffer Manager indicates that any of $\beta_1, \beta_2, \ldots, \beta_k$ contain new data. $\mathcal{F}$ will then process the data passed to it and return $\mathbf{c}_i$ of C. If no buffers have changed, the integrator will use the previous value of $\mathbf{c}_i$, if one exists.

8.4 Built-in Transformations

Transformation functions which are of general utility, such as forward/inverse kinematics and encoder/joint conversions may be provided with ASIAGO.

9 Results and Conclusions

We have begun implementing a prototype of ASIAGO. Currently, ASIAGO is only partially implemented, but we have been able to get a feel for the efficiency of data communication through the Communication Manager and the Buffer Manager.

ASIAGO is intended to be integrated with other elements of the Telerobotic Construction Set (TCS), a modular telerobotic system, which has been developed by the Universities Space Automation/Robotics Consortium (USARC) [18]. TelRIP [19], an object-oriented data exchange mechanism, is used for communication between ASIAGO and the other TCS components. TelRIP runs above TCP/IP, and provides an event-driven application programmer's interface. We utilize the Internet as the physical connection between control sites and work sites, which provides a good test of the effects of time delay and limited bandwidth.

ASIAGO will be initially used to control a Puma-560 and an American Robot Merlin manipulator. These robots are interfaced to ISA bus computers running LynxOS and RCCL [20]. ASIAGO's Integrators will contain a standardized RCCL interface which will be used to issue motion commands and receive sensor data.

Ongoing work includes the analysis of algorithm performance, deadlocks, and synchronization issues. We also plan to work toward completion of the implementation of ASIAGO and baseline testing of ASM performance using a quantitative approach similar to Tyrrell [5]. Possible extensions include the addition of a generalized Event Recognizer object which the operator may reconfigure during task execution and implementation of fuzzy or neural controllers to study their interactions with operator task commands.

In the process of developing a general architecture for intelligent telerobot action selection, we have accomplished a number of useful sub-goals. The requirements analysis and corresponding system object-oriented model will be useful during the design of future teleautonomous systems. The analysis of the ASM requirements has led us to bring in concepts from reactive control and programming languages, and has enabled a very flexible design that directly maps to object-oriented programming constructs.

References

[1] L. Conway, R. A. Volz, and M. W. Walker, "Teleautonomous systems: Projecting and coordinating intelligent action at a distance," *IEEE Transactions on Robotics and Automation*, vol. 6, no. 2, pp. 146–158, April 1990.

[2] L. Conway, R. A. Volz, and M. W. Walker, "Teleautonomous systems: Methods and architectures for intermingling autonomous and telerobotics technology," In *Proc. IEEE Intl. Conf. on Robotics and Automation*, Raleigh, NC, April 1987, IEEE.

[3] J. S. Albus, H. G. McCain, and R. Lumia, "NASA/NBS standard reference model for telerobot control system architecture (NASREM)," Technical Report NIST Technical Note 1235, 1989 Edition, National Institute of Standards and Technology, Robot Systems Division, April 1989.

[4] P. Backes, M. Long, and R. Steele, "The modular telerobot task execution system for space telerobotics," In *Proc. IEEE Intl. Conf. on Robotics and Automation*, pp. 524 – 530, Atlanta, GA, May 1993, IEEE. Vol. 3.

[5] T. Tyrrell, *Computational Mechanisms for Action Selection*, PhD thesis, University of Edinburgh, 1993.

[6] S. Hayati and S. T. Venkataraman, "Design and implementation of a robot control system with traded and shared control capability," In *Proc. IEEE Intl. Conf. on Robotics and Automation*, pp. 1310–1315, Scottsdale, AZ, May 1989, IEEE.

[7] R. A. Brooks, "Intelligence without reason," Technical Report AIM-1293, MIT Artificial Intelligence Lab, April 1991. Also appears in Proc. IJCAI91.

[8] T. B. Sheridan, "Human supervisory control of robot systems," In *Proc. IEEE Intl. Conf. on Robotics and Automation*, pp. 808–812, San Francisco, CA, April 1986, IEEE.

[9] T. Sato and S. Hirai, "MEISTER: A model enhanced intelligent and skillful teleoperation robot system," In *Robotics Research – The Fourth International Symposium*, pp. 155 – 162, MIT Press, 1988.

[10] S. Hayati, T. Lee, et al., "A unified teleoperated autonomous dual-arm robotic system," *IEEE Transactions on Control Systems*, vol. 11, no. 2, pp. 3 – 8, February 1991.

[11] B. Bon and J. Beahan, "A graphics-based operator control station for local/remote telerobotics," In *Proceedings of the SPIE Applications of Artificial Intelligence X: Machine Vision and Robotics Conference*, Orlando, FL, April 1992.

[12] G. Hirzinger, B. Brunner, J. Dietrich, and J. Heindl, "Sensor-based space robotics — ROTEX and its telerobotic features," *IEEE Transactions on Robotics and Automation*, vol. 9, no. 5, pp. 649 – 663, October 1993.

[13] J. Connell and P. Viola, "Cooperative control of a semi-autonomous mobile robot," In *Proc. IEEE Intl. Conf. on Robotics and Automation*, pp. 1118 – 1121, Cincinnati, OH, May 1990, IEEE.

[14] P. Maes, "Situated agents can have goals," In *Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back*, P. Maes, editor, pp. 49–70, MIT Press, Cambridge, MA, 1990.

[15] J. K. Rosenblatt and D. W. Payton, "A fine-grained alternative to the subsumption architecture," In *Proc. AAAI Stanford Spring Symposium Series*, pp. 70 – 74, 1989.

[16] J. Rumbaugh et al., *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, New Jersey, 1987.

[17] B. S. Graves, *Situation-Based Action Selection in Teleautonomous Systems*, PhD thesis, Texas A&M University, 1995. In preparation.

[18] G. V. Kondraske, R. A. Volz, et al., "Network-based infrastructure for distributed remote operations and robotics research," *IEEE Transactions on Robotics and Automation*, vol. 9, no. 5, pp. 702 – 704, October 1993.

[19] J. D. Wise and L. Ciscon, "TelRIP distributed applications environment operating manual," Technical Report 9103, Rice University, 1992. Universites Space Automation/Robotics Consortium.

[20] V. Hayward and R. Paul, "Robot manipulator control under UNIX: RCCL, a robot control C library," *International Journal of Robotics Research*, vol. 5, no. 4, pp. 94–111, 1986.