ASIAGO User's Guide

A System for Intelligent Action Selection for Teleoperation

Sean Graves
Dept. of Mechanical Engineering
Division PMA
Katholieke Universiteit Leuven
3001 Heverlee, Belgium

Marge Skubic
Dept. of Computer Science
Texas A&M University
College Station, Texas 77843-3112

October 9, 1995

Contents

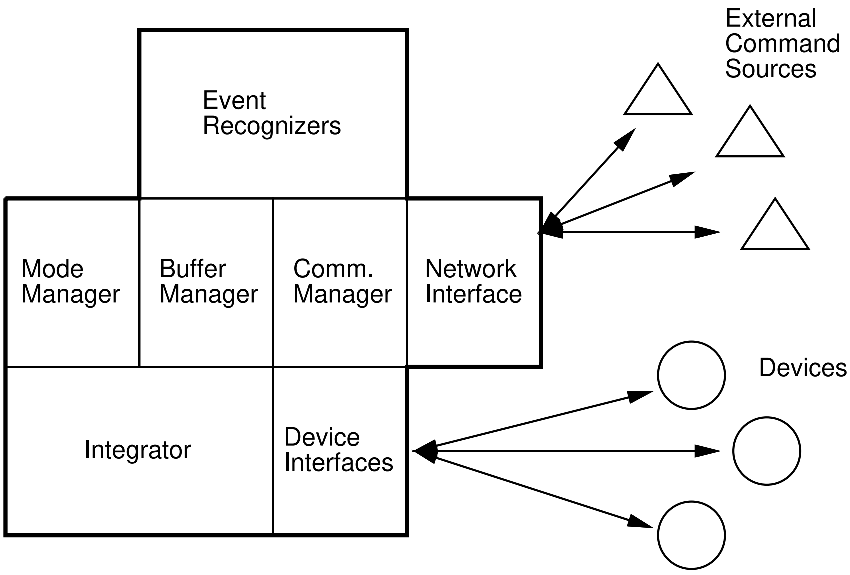

Figure 1: ASIAGO block diagram. Components that directly communicate are shown as connected regions

1 Introduction to ASIAGO

This documents describes ASIAGO – A System for Intelligent Action Generation for teleOperation. ASIAGO is designed to provide a flexible, extensible, and efficient action selection mechanism for telerobot systems. 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.

ASIAGO's primary concept is the *mode*. The mode determines how commands from various command sources are integrated to produce actions. Typical control modes described in the literature include shared control, supervisory control, and traded control. ASIAGO unifies these modes of control by providing a generic mode representation which can be used to produce a variety of useful control modes.

2 Design of ASIAGO

This section describes the structural design of ASIAGO, including the software components and data flows. Some aspects of the design remain to be implemented, as described in section 9.

2.1 Overview of ASIAGO

Figure 1 shows the relationships between the components of ASIAGO, which is constructed as a collection of lightweight processes (threads). Components that directly communicate are shown as connected regions. The heavy line enclosing all of the regions indicates the heavyweight process in which all components run. Communication between other components is performed via the buffer manager. Each object in Figure 2 will be discussed in detail in the following sections.

Figure 2 provides an overview of the dataflows associated with action selection. Beginning with

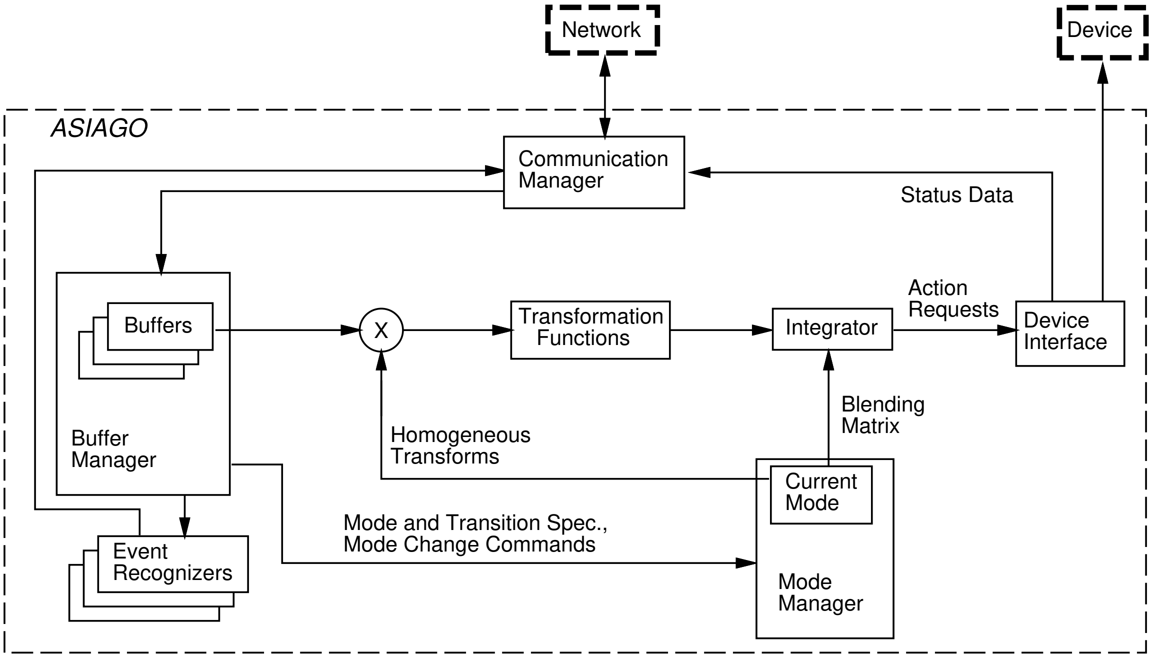

Figure 2: Simplified overview of dataflow associated with the action selection process

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 (ERs) and to the integrator. Before arriving at the integrator, homogeneous transforms may be applied to the 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.

Once the inputs have arrived at the integrator, actions are created according to the current mode. These actions 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 communication manager from the device interface and the network. The current mode is supplied by the mode manager, which is configured by commands issued by ERs.

Each of these components will now be described.

2.2 Communication Manager

The communication manager is where data enters and exits the system, and where creation and initialization of the lightweight processes takes place. The communication manager provides a level of abstraction which shields the rest of ASIAGO from the network interface.

The responsibilities of the communication manager include:

- startup of integrators, mode manager, and event recognizers,
- catching operating system signals for SIGKILL and SIGINTER,
- device shutdown before program exit, and
- periodic status distribution.

2.3 Buffer Manager

Inputs to the action selection process are supplied by the buffer manager, which buffers data that may come from sources external or internal to ASIAGO. Each buffer 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 that 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. These commands 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 that may be issued to ASIAGO externally may be generated internally, and vice-versa. This feature enables event recognizers to perform dynamic configuration and mode changes, or issue motion requests.

The buffer manager is unique because it does not run in its own thread of execution. Rather, it is actually a library of routines which may be called from any process within ASIAGO. This approach was taken to improve efficiency and reduce the possibility of a data bottleneck. In order to implement this scheme, care was taken to ensure that the buffer manager code was reentrant and all critical sections were protected by mutual exclusion (mutex) operators.

The buffer manager provides methods for:

- lookup of buffers by data type and source,
- buffer creation and deletion,
- reading the current record of a buffer,
- adding a record to a buffer,
- registering/removing an interest in updates to a particular buffer, and
- reading the number of records in a buffer.

The buffer manager is also responsible for processing synchronization commands, and for informing processes that new data has arrived in buffers in which they have registered interests.

2.4 Mode Manager

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, but available when needed.

The mode manager supports a special class of modes called 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 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

Table 1: Contents of integrator descriptor record

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`

results in a call to 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.

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. Three types of mode transitions are supported: immediate, linear interpolated, and spline.

2.5 Integrator

The integrator is responsible for taking a set of commands from different buffers and integrating them to produce an action request. The integrator is a generic module which is instantiated in a separate thread within ASIAGO. The method that the generic module uses to generate the action request is governed by the current mode. Each device to be controlled is assigned a single integrator, and this association remains in effect at all times.

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 other fields as described in section 3.2.4. 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. A summary of the fields present in the integrator descriptor record is shown in Table 1.

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

Table 2: Contents of control block

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

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.

LynxOS condition variables are used for process control. The condition variable is an operating system data structure which is used to synchronize access to shared resources between two or more threads. The condition variable is used to indicate the condition (truth value) of a logical predicate. To use a condition variable, a thread evaluates the predicate — if the predicate is not satisfied, the thread sleeps until the condition variable is asserted.

In ASIAGO, the condition variable is stored in the integrator's control block. The logical predicate used to trigger the thread is the `controlBlkCommand` field of the control block. When an action is to be generated, the appropriate party (see Table 3) places a command into `controlBlkCommand` which causes the integrator thread to wake up and process the request.

2.5.1 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 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. Note that other threads, such as event recognizers, may also use the `CB_DEV_REQ` command to initiate action generation when the integrator is running under `AGC_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 the integrator configuration 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 communication 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 communication 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

Table 3: Action generation criteria

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

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 may 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.

2.5.2 Action Selection Algorithm

The result of the action selection process is an action vector **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. For details on the action selection algorithm, refer to [1].

2.6 Event Recognizers

Event recognizers (ERs) are used to trigger mode changes or to issue other commands automatically to 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.

2.7 Network interface

The network interface includes the hardware network connection and a software protocol for delivering commands and data to ASIAGO. The network interface is responsible for extracting the commands and data from the messages received from the network and for passing this data to the communication manager.

The network interface is actually another layer which standardizes the interface between the communication manager and the protocol used for network communication (i.e. TCP/IP or InterAgent).

Table 4: Directory structure

directory	*purpose*
`asiago/`	Main source code for ASIAGO
`asiago/conf/`	Configuration files
`asiago/devices/`	Source code for device interfaces
`asiago/rccl/`	Source code for RCCL device `asiago_rccl_intf`
`asiago/tf/`	Source code for transformation functions

This extra layer was designed to simplify the installation of different network protocols.

2.8 Device interface

In order to interface devices to ASIAGO, a standardized *device interface* is used. 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. The device interface may provide a shared memory interface to another device control program or it may interface to the device directly through a device driver.

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.

2.9 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.

3 Implementation of ASIAGO

ASIAGO consists of approximately 12,000 lines of ANSI C code, running under version 2.1 of the LynxOS operating system [3]. LynxOS is a proprietary UNIX-like real-time operating system, which conforms with POSIX 1003.1 and is designed for deterministic hard real-time response. The network interface was provided by an Ethernet connection. The distributed application environment InterAgent [4] was used to provide ASIAGO with a object-oriented view of command sources.

3.1 Directory Structure

The ASIAGO source code is structured in one main directory, typically called `asiago/`, with several subdirectories of support files. When referring to these support subdirectories, only the directory name (i.e. `conf/`) is given; it is assumed that `asiago/` is the root of the structure. Table 4 illustrates the directory structure.

3.2 Configuration Files

ASIAGO is configured by a set of files in `conf/`. These files are:

- `modes.conf` – Contains a list of mode definitions, read at startup of the mode manager.
- `comm_mgr.conf` – Contains a list of buffer specifications, the contents of which should be periodically distributed as status by the communications manager. Like most other aspects of external status distribution in ASIAGO, this file is not currently used.
- `devices.conf` – Contains a list of available device interface routines.
- `iagent.conf` – Contains a list of InterAgent object names and the associated BDT numbers. These object names are used in defining modes.
- `integrator.conf` – Contains a list of integrators that should be started.

Each of these files will now be described in further detail.

3.2.1 `modes.conf`

The `modes.conf` file provides a local description of modes which may be immediately used by ASIAGO. Currently, this is the only mechanism for defining modes, although it would be straightforward to define a network object that could contain a mode specification and could be parsed using the existing mode specification parser. This would provide an on-the-fly mode creation capability. For details on the theory of mode creation, refer to [1], Section 6.4.

In a mode specification, buffers are specified by giving the source (InterAgent domain) from which the data is distributed, followed by the object type in brackets, followed by a list of properties in parenthesis. The object types are listed in `iagent.conf`.

Since there is only one mode specification file, it may contain many mode specifications. A mode specification may be defined as:

mode_name
$<$*mode_description*$>$
num_inputs num_dofs
dofs
M
δ_1
δ_2
$\vdots$
$delta_{num\_inputs}$

where δ_i is defined as:

transform_function|`NULL` *num_buffers*
  buffername₁
    [*basic data types*]
    homog_transform|`NULL`
  buffername₂
    [*basic data types*]

$homog\_transform$|NULL

$\vdots$

$buffername_{num\_buffers}$

[*basic data types*]

$homog\_transform$|NULL

In a mode specification, whitespace and newlines are ignored. Lines that begin with # are ignored as well. Basic data types are only present if *transform_function* is not NULL.

Lets look at an example mode specification in more detail.

```
1  tball_mode0
2  <teachball control w/obs. avoidance>
3  2 6
4  xtran_rel ytran_rel ztran_rel xrot_rel yrot_rel zrot_rel
5  1.0 1.0 1.0 1.0 1.0 1.0
6  1.0 0.0 0.0 0.0 0.0 0.0
7  NULL 1
8    tamu:tball[eff_cartesian_relative](command)
9       NULL
10 reactive_tf 1
11   tamu:asiago_internal[ST_proximityReading](status)
12      prox_x prox_y prox_z
13      NULL
```

This mode is named `tball_mode0` [line 1]. It allows the robot to be controlled by two control sources, and outputs commands in 6 DOF [line 3]. A short description of the mode is given on line 2. This is more than a comment; it is stored with the mode specification, and displayed in the user interface.

The names of the DOFs controlled by this mode (those it produces as output) are on line 4. Lines 5 and 6 represent M, the blending matrix. Line 5 gives the contribution of the first input to the output (100% on all DOFs). Line 6 gives the contribution of the second input (100% on xtran_rel). Note that this is a deviation from what is stated in [1], that is, the coefficients must sum to 1.0. This rule has been relaxed to more easily support reactive control sources which may only produce some output under particular conditions. To average two inputs, make the weights 0.5 and 0.5; to sum two inputs, use 1.0 and 1.0. In this case we want to sum the inputs.

Line 7 indicates that there is not a transformation function associated with the first input. In such a case, the number of buffers is always equal to one, as shown here. Line 8 indicates that the first input is the `teachball` of `tamu`, which generates `eff_cartesian_relative` commands. Line 9 states that this input does not need to have a homogeneous transform applied to it.

Line 10 says the second input should come from a transformation function, called `reactive_tf`, which has one input. The input is from within ASIAGO (`asiago_internal`), and it is of type `ST_proximityReading` (a proximity sensor simulation). Line 12 says the fields prox_x, prox_y, and prox_z from the `ST_proximityReading` data object should be used as input to the `reactive_tf` function listed on line 10. Line 13 says that no homogeneous transform needs to be applied to the inputs to the transformation function.

3.2.2 `devices.conf`

This file contains a list of available device interface routines. Each device interface is specified by:

> *device_name*
> number n of basic types recognized by interface
> $basic\_type_1$
> $basic\_type_1$
> $\vdots$
> $basic\_type_n$
> `status` [*period*|`async`]
> number m of status datatypes
> $datatype_1$
> $datatype_2$
> $\vdots$
> $datatype_m$

Here is an example of an entry from the `devices.conf` file:

```
1 tamu_merlin
2 19
3 xtran ytran ztran xrot yrot zrot
4 xf yf zf xm ym zm gripper_toggle
5 xtran_rel ytran_rel ztran_rel
6 xrot_rel yrot_rel zrot_rel
7 status 500
8 3
9 eff_cartesian_state
10 force_sensor_state
11 arm_joint_state
```

Line 1 says that this device interface is named `tamu_merlin`. (This name is mapped to the actual code through `devices/dev_info.h`.) Line 2 says that this device has 19 controllable degrees of freedom. Lines 3-6 list the basic types which the device may control. Blanks and newlines are ignored. Line 7 says that status data is to be distributed externally from ASIAGO every 500ms and line 8 says that there are three types of status data. The remaining lines list the data types of status data. Note that external status distribution in ASIAGO is not implemented, so lines 7–11 are not used at present.

3.2.3 `iagent.conf`

`iagent.conf` maps the names of data types to the internal BDT numbers used by InterAgent. Since InterAgent provides no mechanism for looking up this information at runtime, it must be encoded here manually. Each line in the file contains the object name, followed by the hexadecimal object number specified when the object is defined. This file is not parsed using lex, and so it is not as flexible in formatting as the others. Mainly, no comment lines are allowed.

3.2.4 `integrator.conf`

Integrators are generic threads that are attached to a specific device by a configuration file. The integrator is generic because it does not have any hard-coded representation of particular modes or devices. It uses the mode specification data structures to determine how it generates actions. The same code is always used, regardless of the mode, manipulator, or input devices.

`integrator.conf` consists of one or more entries constructed as follows:

> *integrator_name*
> *device_interface*
> `agc_periodic` *period* | `agc_request` | `agc_inform_update`
> *default_mode_name* | `none`
> *failsafe_mode_name* | `none`

Each such line in the file causes an integrator thread to be spawned. Again, whitespace and newlines are ignored, and comments begin with #.

Here is an example of an entry in `integrator.conf`:

```
merlin tamu_merlin agc_periodic 50 tball_mode0 none
```

This file causes an integrator (named `merlin`) to be instantiated. It will generate actions and send them to a device interface named `tamu_merlin`. It will generate an action every 50 ms. It will start out in mode `tball_mode0`. There is no predefined failsafe mode.

3.3 How ASIAGO uses InterAgent domains and addressing

In ASIAGO, all data is keyed according to three criteria – data type, source, and properties. These criteria are used to reference buffers, specify modes, etc. In its current form, ASIAGO uses InterAgent to provide data distribution. The data type, source, and property criteria map directly to InterAgent object types, domains, and properties. The mapping for type descriptors and address values is given in `asiago.h`.

Properties are not handled in a very generic way – only two ASIAGO data properties are supported, `command` and `status`, and these are mapped directly to InterAgent `PT_P_COMMAND` and `PT_P_STATUS` properties.

In order to allow ASIAGO modes to specify the sources of data, the programs which distribute data must signify their functional address. For example, consider a program which generates `eff_cartesian_relative` commands based on teachball data. In the mode specification, the corresponding data buffer may be referenced as `tamu:tball[eff_cartesian_relative](command)`. In order for the data to be recognizable as coming from `tamu.tball`, a domain address must be attached to the data objects. For example, the following code segment would be used in the program generating the data objects:

```
domainAddress = tr_addr_create("FUNCTIONAL:tamu:tball");
HAND_ECR = tr_create_object(EFF_CARTESIAN_RELATIVE, TOBJ_NORMAL);
tr_set_da(HAND_ECR, domainAddress);
```

3.4 ASIAGO Source Code

The ASIAGO source code will now be described in some detail. Each module will be listed, and a short description of each routine in the module will be given. Note that because many of the source files were edited using an MSDOS-based editor, most file names are rather short and cryptic. This means that some of the file names are not as descriptive as they should be (e.g. buffermg.c for the buffer manager). Globally available routines are (often) named beginning with the name of the module to which they belong (i.e. `integratorInit()`). Static routines available only within a particular module do not begin with the module name. Symbols (such as function names, files, directories, data structures, constants, and variables) are signified with the use of a `typewriter` font. Additionally, functions are signified with the `()` suffix and directories are signified with a `/` suffix.

3.4.1 `asiago.c`

Contains the parent thread code for ASIAGO. Note that the entry point is `main_init()` rather than `main()`, due to the use of InterAgent.

Routines:

- `silly()` – A dummy routine which only contains a call to `main()` . This was necessary to force the gcc linker to link in Interagent's `main()`. `silly()` is not called by anybody.
- `main_init()` – The entry point of ASIAGO, which is invoked by InterAgent. It prints a startup message, and calls initialization routines for the communications manager, network interface, transformation functions, buffer manager, integrators, and event recognizers. It also invokes `devIntfReadConfig()`, which reads the `devices.conf` file.

3.4.2 `buffermg.c`

Contains the buffer manager routines. The buffer manager is actually a library of reentrant routines that any of the other threads may invoke. As such, its structures are protected by mutex operators. Each buffer is a linked list, and all the buffers are stored in a linked list called `bufferList`. When a thread is interested in updates of a particular kind of data, it may 'register an interest' in that buffer. To do so, it gives the buffer manager a pointer to its control block, and then sleeps, waiting on the condition variable in its control block. When new data arrives in the buffer, the buffer manager sends a wake-up signal to each thread interested in that buffer. This wake-up signal is sent using the control block's condition variable.

Buffers are automatically created upon receipt of a particular data type from a particular source. Buffers are also created when data types/sources are mentioned in a mode specification.

Routines:

- `signalInterestedThreads()`– When the buffer manager updates a buffer, it calls this routine to signal all threads that have registered an interest in the buffer. *Internal to buffer manager*
- `insertRecord()`– Called by buffer manager to insert a record into a buffer. Calls `signalInterestedThreads()`. *Internal to buffer manager*
- `createBuffer()`– Creates a buffer of a particular type. Buffers may be 'immediate' which means that new data immediately supersedes old data in the buffer (buffer length is always 0 or 1), or they may be 'sequenced,' in which new items are queued up behind old items. The default case is immediate. Implementation of sequenced buffers is incomplete – `insertRecord()`is correct, but there is no mechanism for sequencing the records. *Internal to buffer manager*
- `bufferMgrInit()`– Initializes the buffer manager's data structures. *Called by* `main_init()` *only.*

- `bufferMgrLookup()`– Given a data type/source, this routine returns a pointer to the corresponding buffer. An additional flag tells the buffer manager whether or not to create the buffer if it does not exist.

- `bufferMgrRead()`– Given a pointer to a buffer, this routine returns a pointer to the item at the head of the buffer. Does not remove the item from the buffer.

- `bufferMgrWrite()`– Given a data type/source, this routine looks up the buffer and adds an item to it. If the buffer does not exist, it is created. User-written code should call `commMgrCommandIntf()`, which calls `bufferMgrWrite()`.

- `bufferMgrRecordCnt()`– Given a data type/source, this routine looks up the buffer and returns the current number of records in the buffer.

- `bufferMgrRegisterInterest()`– Given a data type/source, this routine looks up the buffer and registers an interest in it. The calling thread must give a pointer to its control block, the value of the condition variable it will expect in order to wake up, and a flag indicating whether the buffer should be created if it doesn't already exist. After calling this routine, the calling thread should call `waitForConditionVarAssertion()`, which will cause it to sleep until the buffer is updated.

- `bufferMgrRemoveInterest()`– Given a data type/source, this routine looks up the buffer and removes the interest registered by the calling thread. The thread indicates its identity by passing the value of its `controlBlockIndex`.

- `bufferMgrGetCurrTrans()`– Each a buffer is updated, a transaction counter is incremented. This routine returns the value of the transaction counter. The buffer is indicated by use of the buffer pointer.

3.4.3 `cb.c`

This module contains routines which manipulate control blocks. A control blocks is associated with each ASIAGO thread, and are used by the buffer manager to signal a thread when it should wake up and process new data.

Routines:

- `initializeControlBlock()`– Given a pointer to a control block structure, this routine gives the control block a unique index (which is used by the buffer manager to reference the thread) and initializes the mutex and condition variable used for process control. Called by each thread.

- `waitForConditionVarAssertion()`– Called by a thread which has registered an interest in a particular buffer. This routine puts the thread to sleep, awaiting a signal via the condition variable in the thread control block.

- `wakeupCB()`–Called by the buffer manager (`signalInterestedThreads()` to wake up a thread which is sleeping in `waitForConditionVarAssertion()`. Also called by `timerHandler()` for periodic wake-ups (see `timerHandler()`)

3.4.4 comm_mgr.c

Contains the communication manager routines. These routines run in the main thread along with InterAgent.

Routines:

- `timerHandler()`– This routine is a callback used by routines which require a periodic wake-up. For example, an integrator running on `AGC_PERIODIC` will initialize a timer to wake it up according to the specific period using `commMgrScheduleTimer()`. When the timer expires, `timerHandler()` will be called, which then calls `wakeupCB()` to wake up the integrator.

- `commMgrExit()`– Exit point for ASIAGO. Was designed to shut down the integrators, which would then shut down the device interfaces, etc. However, this never worked correctly, and is disabled.

- `commMgrScheduleTimer()`– Called by threads that want a wake-up signal at a specific interval. The thread passes the desired period and the pointer to its control block. Currently implemented with an InterAgent timer, which does not provide very high resolution. Called by integrators running under `AGC_PERIODIC`, and may also be called by event recognizers.

- `commMgrInit()`– Called by `main_init()`. Creates status and message data objects for external distribution of this information. Originally, this routine also installed handlers for `SIGTERM` and `SIGINT` for the kill and interrupt signals, but it is currently commented out due to bugs.

- `commMgrDistribStatus()`– Called by device interfaces and event recognizers to distribute status information internally and externally to ASIAGO. Originally designed to use an external distribution list to see if the data should be sent to network interface, or just issued internally. This is because certain kinds of data may be needed internally at a very high rate, but should not be sent out to the network at the same rate or at all. Currently, status data is issued only to the buffer manager, and not externally. External distribution was not finished in a very nice way. External status distribution must be done manually by the device interfaces and event recognizers. An example of external distribution is in `iagent.c` (`gen_proximity_object()`) which performs status distribution for the `merlintf.c` device interface.

  Status data issued to the buffer manager is treated as any other data received by the system (i.e. `bufferMgrWrite()` is used).

- `commMgrSendProgramStatus()`– Sends a one-line ASIAGO status message to the `xasiago` user interface. Called by various ASIAGO threads, as well as event recognizers and device interfaces.

- `commMgrSendMessage()`– Sends a two-line ASIAGO status message to the `xasiago` user interface. Called by various ASIAGO threads, as well as event recognizers and device interfaces.

- `commMgrSendIntegStatus()`– Sends a integrator status message to the `xasiago` user interface. Called by the integrator threads.

- `commMgrSendModeSpec()`– Sends a mode specification message to the `xasiago` user interface. Called by the mode manager.

3.4.5 `datatype.c`

ASIAGO must deal with data at multiple levels. Data objects arriving from the network interface are often composed of several more basic items such as arrays, floats, ints, etc. ASIAGO must know how to break these objects down into these components in order to integrate the command with other commands. It must also know what these basic data items represent (xtran, ytran, etc.) so it can match them correctly with similar data in other objects. `datatype.c` deals with the three levels of datatypes used by ASIAGO. There are:

1. *Datatypes* – these are InterAgent objects, such as `eff_cartesian_state`. Datatypes are often called 'types.' They are represented with `typeDescriptor` variables.
2. *Basic Data Types* – these are the fields that make up a Datatype, such as xtran, ytran, etc. They are represented with `basicDataTypeEntry` variables. Each basic data type is assigned a sort key. Often, arrays of `basicDataTypeEntry` are used. In this case, they are *always* stored in sort key order.
3. *Primitive Types* – these are `TYPE_FLOAT`, `TYPE_INT`, or `TYPE_STRING` which contain the data of a basic data type. They are represented with `float`, `int`, or `char *` variables, and referred to using the `primitiveDataTypes` enumerated type. Each basic data type is represented by exactly one primitive type.

A table called `typeTable` contains a list of all known datatypes with pointers to routines for manipulating each datatype.

Routines:

- `dataLookupType()` – this routine looks up a datatype (such as `eff_cartesian_state`) and returns a pointer to its `typeTable` entry.
- `dataBuildListEntry()` – takes a pointer to a basic data type structure, and fills in the fields with the basic data type name (such as xtran), the primitive data type, and the data.
- `dataDisassemble()` – given a data type (such as `eff_cartesian_state`), this routine calls the appropriate function to disassemble the object into an array of basic data types.
- `dataTransform()` – given a data type (such as `eff_cartesian_state`), this routine calls the appropriate function to transform the object by the homogeneous transform T.
- `dataPrintList()` – Dumps an array of basic data type entries. Has options to print the type names and to follow each entry with a constant string (such as a space or newline).
- `dataLookupBasicType()` – Given the name of a basic data type (such as xtran) and a basic data type variable, will look up the name, and fill in the fields of the `basicDataTypeEntry` with the proper primitive data type descriptor, sort key, and a zero value.
- `dataCheckBasicDataList()` – This function checks that a list of basic datatypes is properly sorted and contains valid names, IDs, and primitive types.
- `dataCheckBasicDataLL()` – Same as `dataCheckBasicDataList`, except it works on linked lists.
- `dataBasicDataTypeCmp()` – Returns 0 if two `basicDataType`s are the same, non-zero otherwise.

3.4.6 dev_intf.c

These routines are used to manage device interfaces, from initialization to shutdown. A file called `conf/devices.conf` contains a list of known device interfaces. This file indicates the name of the interface, the commandable degrees of freedom, the rate with which status information is distributed, and the types of status data. Unfortunately, this status information is not currently used.

Some of the routines in `dev_intf.c` use a simple lexical interpreter implemented using lex to read configuration files. This allows the configuration files to be much more free-form, and allows better error checking.

Routines:

- `readDataTypeList()` – Reads the list of basic data types which can are commandable by this device interface. Each entry is looked up using `dataLookupBasicType()`, and the entries are stored in a sorted linked list. *Internal, called by* `devIntfReadConfig()`.
- `dumpDataTypes()` – Prints out a linked list of basic data types, as constructed by a call to `readDataTypeList()`.
- `devIntfLookupCode()` – given the name of a device interface (from the `devices.conf` file), returns a pointer to the entry point of the code. The entry points are stored in a static array called `devIntfTable`, which is stored in `devices/dev_info.h`.
- `devIntfShutdown()` – sends a exit command to all devices.
- `devIntfReadConfig()` – builds a list of device interfaces by reading `conf/devices.conf`. Called by `main_init()`.
- `devIntfLookup()` – given the name of a device interface, returns a pointer to the device interface table entry.
- `devIntfDiagnostics()` – dumps the contents of the device interface table. Currently, a call to this routine is in `main_init()`, and is commented out.

3.4.7 er.c

Routines:

- `eventRecognizersInit()` – starts up the event recognizers, by allocating a control block and initiating a thread. Not very generic, currently set up to start one event recognizer called `erTask()`.

3.4.8 iagent.c

This file contains the network interface routines for InterAgent. To use a different network interface, start by replacing these routines. In this module, 'registering an interest' is in the InterAgent sense, not the buffer manager sense.

Routines:

- `dataHandler()` – This is the InterAgent handler for all data InterAgent interests. All InterAgent objects are received by this routine, which then passes the data to the communication manager.

- `lookupObjectNumber()` – This routine takes a textual datatype name (such as `eff_cartesian_state`) and returns the InterAgent base data type number (such as `0x4302`). This is implemented using a lookup table stored in `conf/iagent.conf`. Therefore, all new data types must be added to this file.

- `networkIntfInit()` – This routine reads `conf/iagent.conf` to build a linked list of data type name/number pairs. These are used when registering interests with InterAgent.

- `networkIntfRegisterInterest()` – Called by the mode manager, and other threads that want to register an interest with the network interface in certain data types. When mode specifications are processed by the mode manager, this routine for each data type referenced by the mode, so that the buffers will contain data when that mode is invoked. When the mode is then invoked, the corresponding `bufferMgrRegisterInterest()` call is made. A internal list of interests is maintained to ensure that an interest is not registered twice for the same data.

- `networkIntfCmpAddrs()` – Used to compare two network interface addresses (data sources).

- `networkIntfParseRecordType()` – Given a buffer specification, such as `tamu:motion_tool[eff_cartesian_state](command)` this routine will build a type descriptor from the object type information, (i.e. `eff_cartesian_state` and `command`). The routine can also work on only the object name (i.e. `eff_cartesian_state` – no address, properties, or brackets).

- `networkIntfParseRecordSource()` – Given a buffer specification (such as `tamu:motion_tool[eff_cartesian_state]`) this routine will build a address descriptor from the address information, (i.e. `tamu:motion_tool`). The routine can also work on only the domain name (i.e. `tamu:motion_tool` – no object type or brackets).

- `networkIntfCmpTypes()` – Used to compare two network interface data types.

- `networkIntfCmpProps()` – Used to compare two network interface data type properties.

- `networkIntfInternal()` – Returns true if the given network interface address is internal to ASIAGO, namely `tamu:asiago_internal`, as specified by `ASIAGO_INTERNAL_DOMAIN` in `asiago.h`.

- `gen_proximity_object()` – A temporary routine used to generate status objects for the proximity sensor simulator device. Eventually it should be replaced with a generic routine which would take the name of the data type, a list of basic data types, and a flag indicating whether the data should be distributed externally to ASIAGO.

3.4.9 `integrat.c`

This module contains the integrator thread code. It also contains some code which runs in the main thread, such as initialization routines.

Routines:

- `checkForValidRequest()` – Verify that the integrator was activated with a signal appropriate for the AGC in effect. If not, an error handler is invoked. *Internal – called by the integrator thread.*

- `buildIntegList()` – Build a linked list of integrators by opening `integrator.conf` and building an integrator descriptor for each file entry. Allocates a control block for each. *Internal – called by the main thread, from* `integratorInit()`.

- `buildCMatrix()` – collects the inputs from the appropriate buffers. See [1] (section 6.3.4) for details on the **C** matrix. The **C** matrix is built from the raw data in the buffers, possibly transformed using transformation functions and homogeneous transforms. This is determined by the current mode. Error checking is taken care of at mode specification time, and at mode transition time. *Internal – called by the integrator thread.*

- `calcActionVector()` – given the **C** matrix, and the blending matrix given in the mode specification, this routine calculates the action vector. See [1]. *Internal – called by the integrator thread.*

- `integratorTask()` – the entry point of the integrator thread. Sleeps awaiting wakeup by the buffer manager (for `AGC_INFORM_UPDATE`), from the communication manager (`AGC_PERIODIC`), or from the device interface/event recognizer (`AGC_REQUEST`). When woken, a new action vector is calculated using `buildCMatrix()` and `calcActionVector()`.

- `integratorInit()` – called by `main_init()` to build the list of integrators, initialize the device interfaces, schedule timers (for `AGC_PERIODIC`), and spawn the integrator threads.

- `integratorShutdown()` – Causes all integrators to shutdown their devices and exit.

- `integratorDiagnostics()` – Dumps the contents of integrator data structures to `stdout`. Currently, a call to this routine is in `main_init()`, and is commented out.

- `integratorLookup()` – Given an integrator name (i.e. from `conf/integrator.conf`), returns a pointer to the corresponding integrator list entry (as constructed by `buildIntegList()`).

- `integratorFail()` – This routine is called when an fatal error occurs in an integrator thread. It causes the thread to signal the communications manager that a fatal error has occurred.

3.4.10 `mode_mgr.c`

The mode manager is unique in that it consists of a stand alone thread as well as a set of routines which are meant to be called by the integrator. As such it is a 'library' as well as a 'manager.' Also, a few of the routines are called by the main thread during system initialization. In this section, each routine is marked to indicate the thread in which it runs.

Routines:

- `readLocalModes()` – Reads the local mode specification file (`conf/modes.conf`). Uses a simple lexical analyzer written in lex. See [1] for details on mode specification. See Section 3.2.1 for file format information. *Internal – called by* `modeMgrInit()`.

- `dumpTransition()` – Prints a mode transition specification, which consists of a transition name, a type (i.e. immediate, linear, or spline), and a transition ID. *Internal – called by* `dumpMode()`.

- `dumpDOFList()` – Prints a list of degrees of freedom (a linked list of basic data types). This list is part of the mode specification. *Internal – called by* `dumpMode()`.

- `dumpMode()` – Given a pointer to a mode specification, this routine prints out the contents the data structures.

- `zeroMatrix()` – Returns a matrix with all elements equal to 0. Used during a non-immediate mode transition. See [1] Section 6.4.4. *Internal – called by* `buildTransitionalMatrix()`.

- `composeM()` – Used to combine two blending matrices. Used during a non-immediate mode transition. *Internal – called by* `modeMgrModeChange()`.

- `composeD()` – Used to combine two input maps. Used during a non-immediate mode transition. *Internal – called by* `modeMgrModeChange()`.

- `composeModeDOFs()` – Used to combine the list of DOFs in a starting mode with those of a finishing mode for non-immediate mode transition. *Internal – called by* `modeMgrModeChange()`.

- `buildTransitionalMatrix()` – Builds a transitional blending matrix used during a non-immediate mode transition. *Internal – called by* `modeMgrModeChange()`.

- `modeMgrModeChange()` – Performs a mode change. Takes a transition descriptor, a integrator pointer, a mode pointer, and the duration of the transition. If necessary, builds the transitional matrix necessary for non-immediate transitions. Can also deal with a transition during an existing transition. If integrator is running under `AGC_INFORM_UPDATE`, it informs buffer manager of changes in interests. *Runs in mode manager thread, upon receipt of a* `CB_MODE_TRANSITION` *command.*

- `modeMgrTask()` – Entry point of the mode manager thread. Waits for its condition variable to be asserted, which indicates that a mode transition is needed. The condition variable is asserted by the buffer manager upon receipt of a mode change request command.

- `modeMgrInit()` – Called from `main_init()` to start up the mode manager thread. Informs buffer manager and network interface of interest in mode configuration objects.

- `modeMgrDeleteMode()` – Frees up the memory taken by a mode. *Called by integrator thread.*

- `modeMgrLookup()` – Looks up a mode by name. Returns a pointer to the mode descriptor. *Called by integrator thread.*

- `modeMgrRegisterIntegrator()` – Called by `buildIntegList()` to get a handle to the mode descriptor for each integrator. The mode manager creates these handles so that it can change the modes asynchronously. *Runs in the main thread.*

- `modeMgrGetTransitionState()` – The routine calculates the current state of a non-immediate mode transition. If there was a transition in progress the last time an action was generated, it decides if the transition is complete. Returns `TRUE` (in transition) or `FALSE` (no transition). When the transition is complete `modeMgrGetTransitionState()` performs the final immediate transition, as described in [1], Section 6.4.4. *Called by integrator thread.*

- `modeMgrCopyMode()` – Returns a pointer to a copy of a mode. The calling party is responsible for freeing the memory. The purpose of this routine is to reduce the time the `modeList` mutex is locked, and to simply access to the mode structures by the caller. *Called by integrator thread.*

- `modeMgrCalcLambda()` – Calculates the value λ, which ranges from 0 to 1, indicating the progress of a non-immediate transition. When λ reaches 1, the transition is complete. *Called by integrator thread.*

- `modeMgrDiagnostics()` – Dumps the contents of the mode manager data structures. Currently, a call to this routine is in `main_init()`, and is commented out.

- `modeMgrTransLookup()` – Takes the ID number of a mode transition specification and returns a pointer to the corresponding descriptor. See [1], Section 6.4.6. *Called by* `modeMgrTask()` – *runs in mode manager thread.*

3.4.11 `tf.c`

This module provides the interface between the integrators and the transformation functions provided by the user. When a mode specifies that a particular transformation function is to be used, the transformation function is called by the integrator, who passes a list of input parameters and a command code of `TF_TRANSFORM`. The list of input parameters is determined by the mode and should match the prototype list returned by `tfInit()`; For more details see section 3.5.3.

Routines:

- `tfLookup()` – Looks up a transformation function by name from the table in `tf_table.h`. Returns a pointer to the descriptor record of the transformation function, which includes a pointer to the entry point of the function.

- `tfInit()` – Calls each transformation function, passing the `TF_INIT` command. This will cause the function to return an array of `basicDataTypeEntry`s, which is the expected input to the function.

- `tfDiagnostics()` – Prints a list of all transformation functions and their input parameter lists. Currently, a call to this routine is in `main_init()`, and is commented out.

3.5 Auxiliary Source Code

The following modules are not part of the core of ASIAGO. They serve as examples of how to use ASIAGO to handle certain data types, events, and devices. However, without files of this type, ASIAGO could perform no useful work. These files are linked into the current version of ASIAGO. In this section, we will not describe every routine in detail, since the example code is simple and builds on the routines already described.

3.5.1 `demo_er.c`

This file contains a simple event recognizer used to perform the task described in [1] Section 8.2.4. It implements a simple state machine which coordinates the usage of five different control modes. The primary routine in this module is `erTask()`, which is invoked from `eventRecognizersInit()`. `erTask()` and the other routines in `demo_er.c` illustrate the usage of the following routines:

- `bufferMgrLookup()`,

- `bufferMgrRead()`,

- bufferMgrRegisterInterest(),
- bufferMgrRemoveInterest(),
- bufferMgrWrite(),
- commMgrSendMessage(),
- networkIntfParseRecordSource(),
- networkIntfParseRecordType(),
- networkIntfRegisterInterest(), and
- waitForConditionVarAssertion().

These and other commands listed in the previous sections form the ASIAGO API (Application Programmer's Interface), which can be used to develop other event recognizers. See section 6.

One additional point should be made about demo_er.c. It uses bufferMgrWrite() to place commands into ASIAGO — this is not technically correct. User-written code should instead call commMgrCommandIntf(), which calls bufferMgrWrite().

3.5.2 ecr.c, ecs.c, force.c, prox.c

Each of these files contain information necessary for ASIAGO to make use of a particular data type (eff_cartesian_relative, eff_cartesian_state, force_sensor_state, and proximityReading, respectively.) To add a new data type to ASIAGO, use one of these files as a template. In each .c file, there are up to three routines: disassemble, print, and transform. A lookup table in datatype.h provides a mapping from the type names to these utility functions at run time. Refer to section 7.2 for the procedure for adding new data types.

3.5.3 tf/test_tf.c, fixture_tf.c, reactive_tf.c

The tf/ subdirectory contains the source code for transformation functions. The simplest example is test_tf.c, which accepts a ztran value, and returns -ztran. test_tf.c demonstrates how a transformation is supposed to process the two transformation function commands TF_INIT and TF_TRANSFORM . When passed the TF_INIT command, the transformation function should return an array of basicDataTypeEntry which indicates the input parameters expected by the transformation function. When passed the TF_TRANSFORM command, the transformation function expects an input array of values corresponding to the prototype it returned upon initialization. It then returns a array which contains transformed or derived values.

Transformation functions are referenced in the mode specification by name. A table, called tfTable, in tf_table.h, maps the function names to the corresponding entry point.

tf/fixture_tf.c is a transformation function which causes the robot to move toward a synthetic fixture (a vertical line in this case). A simple potential field approach is used. fixture_tf accepts an xtran and a ytran value which represent the current position of the robot. The position of the synthetic fixture, as well as the range of influence, is hard-coded into the transformation function. The transformation function returns a xtran_relative and a ytran_relative which cause the robot to move toward the fixture.

tf/reactive_tf accepts a proximity reading from a proximity sensor. (In the current implementation, the proximity sensor is simulated in the dev/merlintf.c device interface.) reactive_tf generates relative motions which move the robot away from the obstacle. (Currently, motions are only generated in the x axis.)

For a description of how to create modes which invoke transformation functions, see section 3.2.1. For a description of how to link new transformation functions into ASIAGO, see section 7.3.

3.5.4 devices/merlintf.c

The devices subdirectory contains the device interface for the American Robot AR6500 Merlin. The device interface has one public function, called merlinDevIntf(), which is invoked by the integrator when an action has been requested and generated. The Device interface responds to DEV_INIT, DEV_ACTION, DEV_EXIT, and DEV_NOOP command codes.

Upon initialization, the device interface connects the the device. In the case of merlintf.c, this entails a shared memory connection to a separate RCCL program which controls the robot. The DEV_INIT routine also initializes status distribution using InterAgent calls. This is non-generic, and should be improved. Status data is distributed at 20 Hz, and includes the eff_cartesian_state and force_sensor_state, as well as a proximity reading, based on a simulated proximity sensor. This proximity reading is based on a hard-coded obstacle position, and the current position of the robot.

commMgrDistribStatus() is used to distribute status data. This means that (in the current implementation) status data is only sent to the buffer manager for internal distribution. In the case of proximity data, an additional call to gen_proximity_object() is used to distribute the data externally via InterAgent.

When an action is sent to the device interface, it consists of an array of basic data types. These basic data types are sent directly to the RCCL program through shared memory. Thus, the RCCL program must have the capability to interpret the basic data type structure.

3.5.5 lex.lex, lex.yy.c

lex.lex contains the lex source code for a simple lexical analyzer used by the device interface, mode manager, and integrator to parse the configuration files. lex.yy.c is generated by lex from lex.lex.

3.5.6 Header files

Each of the header files will now be briefly described.

- asiago.h – Included by all ASIAGO core source files. Includes the system headers, defines constants and major data structures, and gives function prototypes for all ASIAGO functions.
- buffermg.h – Definitions local to the buffer manager.
- cb.h – Defines control block commands and definitions local to control blocks.
- comm_mgr.h – Contains some definitions used in communication manager structures.
- datatype.h – Includes all headers for new datatypes, such as ecs.h and ecr.h. Also contains the typeTable, which is used to map the names of objects to the entry points for their disassemble, transform, and print routines. Also contains the basicTypeTable, which is a sorted

list of basic data types (i.e. xtran, ytran) which are referenced in mode definitions, transformation functions, device interfaces, etc. When new objects are added to ASIAGO, these tables must be updated. For more details, see section 7.2.

- `dev_intf.h` – Contains definitions of command codes which are passed from the integrator to the device interface.
- `devices/dev_info.h` – Contains function prototypes for device interface routines, and the `devIntfTable`, which maps device names to the device interface entry point.
- `ecr.h`, `ecs.h`, `force.h`, `prox.h` – Contains prototypes for each data type's disassemble, transform, and print routines. As new data types are added these files can be copied and the routine names changed appropriately. The prototypes should not be changed however, since ASIAGO expects these routines to have a fixed prototype.
- `iagent.h` – Contains local structures used by the InterAgent network interface routines.
- `integrat.h` – Contains local definitions used by the integrator. If new AGCs are needed, their names should be added to this file.
- `mode_mgr.h` – Contains definitions used to issue mode change commands, and local mode manager definitions.
- `tf.h` – Contains definitions of the `TF_INIT` and `TF_TRANSFORM` commands.
- `tf_table.h` – Contains the `tfTable`, which maps transformation function names to their entry points.
- `type_ids.h` – Defines the ID number for each basic data type. These are used mainly as sort keys. All lists of basic data types (such as action vectors, mode specifications, and device interface descriptions) must be built in the order given by the keys in this file.
- `y.tab.h` – Contains definitions of the lexemes returned by the lexical analyzer. Also includes some macros used by ASIAGO to invoke the lexical analyzer.

3.6 Primary Data Structures

Most of ASIAGO's important data structures are defined in `asiago.h`. These are:

- `basicDataTypeEntry` – Represents an instance of a basic data type, including the current value.
- `bufferObject` – Represents a buffer.
- `controlBlk` – A control block.
- `deltaType` – Used to represent the input map Δ in a mode (see [1] section 6.4). Each Δ (of the form $\{\mathcal{F}, \{\beta_1, T_1\}, \{\beta_2, T_2\}, \ldots, \{\beta_k, T_k\}\}$ is represented by a `deltaType`. Each of the $\{\beta_i, T_i\}$ entries are stored in a `subDeltaType`.
- `deviceDescriptor` – Represents a device interface. Contains `devEntry`, the entry point of the device interface, which is called by the integrator.

- `integDescriptor` – Represents an integrator. Contains a handle to the current mode (obtained from the mode manager upon calling `modeMgrRegisterIntegrator()`). Also contains thread information, a control block, a pointer to the attached device interface, and mode transition information.

- `matrix` – not used.

- `modeDescriptor` – Represents a mode. Contains the mode's name and description, the blending matrix, and a pointer to the Δ structure.

- `subDeltaType` – Used to represent a $\{\beta_i, T_i\}$ entry of a Δ of a mode. See `deltaType` above.

- `transDescriptor` – Structure to represent a transition specification. Contains the name and type of the transition (immediate, linear, or spline), and a pointer to transition-specific data.

- `transformFunctionDescriptor` – Represents a transformation function. Contains the function name (as used in a mode specification), a pointer to the entry point of the function, and an array of `basicDataType`s which specifies the expected input to the function.

- `typeTableEntry` – The structure used to hold entries in the `typeTable` (i.e. the type name, and pointers to the disassemble, transform, and print functions).

Data structures defined in other files usually have a more limited usage and scope. They include:

- `asiagoRCCLControlBlk` – (`devices/merltype.h`) A structure contained in shared memory which is used to communicate between ASIAGO and the RCCL device. Also contains fields used to return status data back to ASIAGO.

- `basicTypeTableEntry` – (`datatype.h`) A structure used to describe a `basicDataType`. Contains the type name (i.e. xtran), the primitive data type used store that basic data type (i.e. `TYPE_FLOAT`), and the sort key for this basic data type.

- `bufferInterestListEntry` – (`buffermg.h`) A structure used to represent a thread's interest in a particular buffer. Contains a pointer to the thread control block and the command that the thread expects to be placed in the control block.

- `devIntfTable` – (`devices/dev_info.h`)A table of device interfaces. New devices should be added here. Contains the name of the device interface (as used in `conf/integrator.conf`) and a pointer to the entry point of the device interface.

- `interestListEntry` – (`iagent.h`) Used to record interests that the communications manager has registered with the network interface. This is to prevent multiple interests to be registered for the same data.

- `mmIntegListEntry` – (`mode_mgr.h`) The indirection structure used to associate each integrator with its current and failsafe modes. One of these is allocated for each integrator, and contains a pointer to the integrator's mode.

- `mmTransitionReq` – (`cb.h`) Not used. Rather than using this dedicated structure for mode transition requests, the InterAgent object `asiagoModeChangeCmd` is used as the structure passed through the buffer manager to the mode manager.

- `nameMapEntry` – (`iagent.h`) Used to store the datatype/BDT pairs which map datatype names used in mode specifications with the InterAgent object numbers needed to register interests.
- `rcclDataEntry` – (`devices/merltype.h`) A structure used to represent one command of one degree of freedom issued from ASIAGO to the RCCL device. An statically allocated array of these objects is passed using the `asiagoRCCLControlBlk`.
- `tfTable` – (`tf_table.h`) The table of transformation functions, containing the function name (as used in the mode specification) and a pointer to the entry point of the function.
- `typeTable` – (`datatype.h`) The table of datatypes known to ASIAGO. Contains the datatype name, and entry points to functions to disassemble, transform, and print objects of that datatype.

Linked lists in ASIAGO are represented using Jiri Matas' `LL` library, version 8.1.

3.7 Process Creation and Initialization

ASIAGO runs as a single multi-threaded process under LynxOS. The communication manager handles thread creation and initialization. Figure 3 shows the flow of control under a typical situation of one event recognizer and one integrator. The primary functions which are called are also shown. The functions are listed according to their dynamic execution order, not according to their lexical scope. For example, in the second call to the event recognizer, the figure shows that a call to `commMgrCommandIntf()` may be made. In turn, `bufferMgrWrite()` would be called, which would call `bufferMgrLookup()`, and so on. In other words, the event recognizer does not directly call the functions listed below `commMgrCommandIntf()`.

The integrator is shown running under the `AGC_INFORM_UPDATE` protocol. Thus, the integrator is activated by receipt of any data object referenced in its current mode. If the integrator were running under `AGC_PERIODIC`, a few differences would occur. First, the main thread (in `integratorInit()`) would register a timer wakeup for the integrator, using `commMgrScheduleTimer()`. Then, according to the period of the timer, `timerHandler()` would be invoked, which in turn calls `wakeupCB()` to wake up the integrator.

Note from the figure that the buffer manager calls may be made in any thread (shown in the event recognizer thread and in the main thread). This is because the buffer manager and the other utility functions are not a separate thread, but simply a reentrant library.

4 The RCCL Device

The `merlin` device interface (in `devices/merlintf.c`) provides a generic shared memory interface to a separate RCCL [2] program. This RCCL program (`rccl/asiago_rccl_intf`) controls the robot and returns status data to ASIAGO. Figure 4 illustrates this process. In this section, a brief overview will be given on `asiago_rccl_intf`.

In RCCL robot control is divided into two levels — the planning level and the control level. These levels run in separate threads, with the control level running at a high priority. In this case, the planning level is used to perform communication with ASIAGO. The planning level accepts commands from ASIAGO though the shared memory interface and updates control variables accordingly. The control level, in turn, controls the end effector of the robot in order to obtain the desired forces

Figure 3: Time line of function calls in ASIAGO, showing one event recognizer, one integrator (running under `AGC_INFORM_UPDATE`), and the mode manager. The figure illustrates the primary functions invoked during thread creation and during nominal processing. The event recognizer is depicted issuing a mode change command.

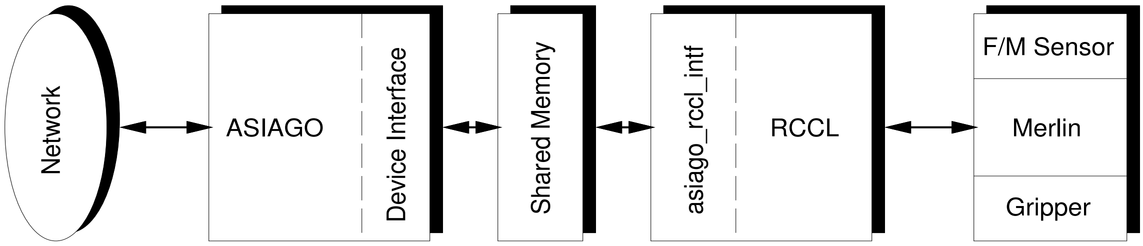

Figure 4: Block diagram illustrating the RCCL device interface

and positions. A condition variable in the shared memory segment is used to synchronize ASIAGO with the planning level.

The planning level expects commands from ASIAGO to be in the form of an array of basic data types. Therefore, it includes ASIAGO's `type_ids.h` file, and several other files specific to ASIAGO in order to parse these commands. Changes in the `basicDataType`s used in ASIAGO must be reflected in `asiago_rccl_intf`.

The robot is controlled using RCCL's `transEval()` function, which binds a transformation to the position of the arm. Each servo cycle, this transformation is re-evaluated to determine the current arm position. Thus, as commands arrive from ASIAGO, all that must be done is to modify the transformation accordingly, causing the robot to move. Control level functions are used to transform desired forces into positional offsets using a simple proportional control law.

The planning level initiates a separate RCCL monitor function using `runMonitorFxn()`, which samples the force sensor in each servo cycle. These force readings are used to during force controlled operations, and are also made available for status monitoring. A separate status thread is also created by the planning level. This status thread periodically places the current robot position, force readings, and joint positions into the shared memory segment. The `merlin` device interface contains code for retrieving this status data and distributing it appropriately. The period of the status function is hard-coded at 25ms.

Here is a short description of each file which comprises the RCCL device:

- `rccl/asiago_rccl_intf.c` – Contains the RCCL planning level code. Connects to ASIAGO through the shared memory interface, starts up the control level thread, the force sensor monitor, and the status distribution thread.

- `rccl/asiago_rccl_intf.h` – Contains some constants used in communication between the planning and control levels. Also includes the definition of a shared memory object used to communicate between the two levels. This is not the same as the shared memory segment used to communicate between `asiago_rccl_intf` and ASIAGO.

- `rccl/asiago_rcclCtrl.c` – Contains the control level functions which are called during each servo cycle.

- `rccl/status.c` – Contains the status gathering functions. The rate of status gathering is hard-coded in a `usleep()` call, and is currently set at 25ms.

5 ASIAGO Usage

To start ASIAGO, three steps must be performed:

1. Start an InterAgent router. This may be done on a different machine than on which ASIAGO is running. (type `router -f &`)
2. Start `asiago_rccl_intf`. (Type `rccl/asiago_rccl_intf` on the machine where ASIAGO is to run.)
3. Start ASIAGO. (On the same machine on which `asiago_rccl_intf` is running, type `asiago -- -s`*hostname*, where *hostname* is the name of the machine where the router is running.)

If there are problems connecting ASIAGO to `asiago_rccl_intf`, try deleting any existing shared memory segments using `smems -r asiago_dev_intf` and then retry steps 2 and 3.

6 The ASIAGO Application Programmer's Interface (API)

The ASIAGO API consists of a library of routines which may be called by user supplied routines, such as event recognizers and device interfaces. Table 5 lists the API functions. The API routines are also used by the ASIAGO core, and are described in more detail in section 3.4. See section 3.5.1 for a discussion of an example module which uses these functions. For details on the input and output parameters of these functions, refer to the function prototypes in `asiago.h` and to the code described in section 3.5.1.

7 Customization

This section describes how to customize ASIAGO. This includes adding new modes, data types, transformation functions, event recognizers, and device interfaces.

7.1 Mode Specification

New mode specification does not require any modification or re-compilation of ASIAGO. Refer to section 3.2.1 for instructions on specifying new modes.

7.2 Data Type Specification

New data types may be added to ASIAGO. This enables it to process new network objects which may be developed. The steps for adding a new datatype:

1. Add the type name and number to `conf/iagent.conf`.
2. Add basic types (such as xtran) which are unique to this type to `type_ids.h`.
3. In `datatype.h`:
   (a) Add a `#include` for the datatype being added.
   (b) Add an entry in `typeTable` listing the type name, and entry points to the disassemble, transform, and print functions for this type.
   (c) Any basic types added in step 2 must be added to `basicTypeTable`. The basic data type IDs must be in sorted order.

Table 5: The ASIAGO Application Programmer's Interface

function	*description*
`bufferMgrGetCurrTrans()`	Returns the value of a buffer's transaction counter
`bufferMgrLookup()`	Given data type/source, returns a pointer to a buffer
`bufferMgrRead()`	Returns pointer to item at head of buffer
`bufferMgrRecordCnt()`	Returns the current number of records in a buffer
`bufferMgrRegisterInterest()`	Requests notification of new data
`bufferMgrRemoveInterest()`	Removes notification request
`commMgrCommandIntf()`	Places data into ASIAGO (issues to buffer manager)
`commMgrDistribStatus()`	Distribute status information internally
`commMgrExit()`	Exit point for ASIAGO
`commMgrScheduleTimer()`	Request periodic notification
`commMgrSendMessage()`	Sends status message (2-line) to `xasiago`
`commMgrSendProgramStatus()`	Sends status message (1-line) to `xasiago`
`dataBasicDataTypeCmp()`	Compares two `basicDataType`s
`dataBuildListEntry()`	Initializes a `basicDataType` entry based on pointer
`dataCheckBasicDataLL()`	Checks integrity of linked list of `basicDataType`s
`dataCheckBasicDataList()`	Checks integrity of array of `basicDataType`s
`dataLookupBasicType()`	Initializes a `basicDataType` entry based on name
`dataLookupType()`	Looks up datatype by name and returns pointer from `typeTable`
`dataPrintList()`	Dumps an array of `basicDataType` entries
`devIntfDiagnostics()`	Dumps the contents of the device interface table
`integratorDiagnostics()`	Dumps the contents of integrator data structures
`lookupObjectNumber()`	Takes a datatype name and returns the InterAgent BDT
`modeMgrDiagnostics()`	Dumps the contents of the mode manager data structures
`networkIntfCmpAddrs()`	Compares two network interface addresses (data sources)
`networkIntfCmpProps()`	Compares two network interface data type properties
`networkIntfCmpTypes()`	Compares two network interface data types
`networkIntfInternal()`	Checks if a data source is internal to ASIAGO
`networkIntfParseRecordSource()`	Builds address from textual information
`networkIntfParseRecordType()`	Builds type descriptor from textual information
`networkIntfRegisterInterest()`	Places request on network for certain data type
`tfDiagnostics()`	Lists all transformation functions and their input parameters
`waitForConditionVarAssertion()`	Causes thread to sleep until event occurs

4. Create the header file mentioned in step 3a. The only requirement is that it give function prototypes for the disassemble, transform, and print functions. These prototypes must be as follows:

   ```
   void
   my_disassemble(void *rawdata, basicDataTypeEntry *data);
   void *
   my_transform(void *rawdata, double *T);
   void
   my_print(void *rawdata);
   ```

5. Create a `.c` file containing the disassemble, transform, and print functions.

6. Add the `.c` file to the ASIAGO makefile.

7. Now use the new datatype by creating appropriate entries in `conf/modes.conf`.

For an example, refer to `ecs.c` (`eff_cartesian_state`).

7.3 Transformation Function Specification

A transformation function has the following function prototype:

```
void *
myTF(int tfFxnCode, void *tfData);
```

A transformation function must process the two transformation function commands `TF_INIT` and `TF_TRANSFORM`, which are passed in `tfFxnCode`. When passed the `TF_INIT` command, the transformation function should return an array of `basicDataTypeEntry` which indicates the input parameters expected by the transformation function. When passed the `TF_TRANSFORM` command, the transformation function expects an input array of values in `tfData` corresponding to the prototype it returned upon initialization. It then returns a array which contains transformed or derived values. To add a transformation function to ASIAGO:

1. In `tf_table.h`:
   (a) Add a `#include` for the transformation function being added.
   (b) Add an entry in `tfTable` listing the function name and its entry point.

2. In the directory `tf/`, add the header file mentioned in step 1a. The only requirement is that it give a function prototype for the entry point.

3. Create the `.c` file containing the transformation function. Look at one of the other `.c` files in `tf/` for a template.

4. Add the `.c` file to the `libtf` makefile in `tf/`.

5. Type `make` in the `tf/` directory.

6. Type `make` in the `asiago/` directory.

7. Now use the new transformation function in a mode specification.

For an example, refer to `tf/reactive_tf.c`.

7.4 Device Interface Specification

A device interface has the following function prototype:

```
int
myDevIntf(int devFxnCode, void *devData);
```

The Device interface must respond to `DEV_INIT`, `DEV_ACTION`, `DEV_EXIT`, and `DEV_NOOP` command codes, which are passed in the `devFxnCode` parameter of the function prototype. The device interface should return 0 if there is no error, or a non-zero value otherwise. In case a non-zero value is returned, ASIAGO will print an error message and the returned value, and exit.

When called with `devFxnCode` = `DEV_INIT`, the device interface should initialize communication with the device and perform any other necessary initializations. This call will occur just prior to the integrator thread being created by the communication manager.

When called with `devFxnCode` = `DEV_ACTION`, the `devData` field will contain an array of basic data types, which should be issued to the device as commanded values. When called with the other function codes, `devData` should be ignored. This call occurs when an action vector has been requested of, and generated by, the integrator.

When called with `devFxnCode` = `DEV_EXIT`, the device should be shut down safely. When called with `devFxnCode` = `DEV_NOOP`, the device interface should return 0. This is used in the `devIntfDiagnostics()` function to verify that the device interface is properly linked. To add a device interface to ASIAGO, the following steps must be performed:

1. In `devices/dev_info.h`:
   (a) Add a `#include` for the device interface.
   (b) Add an entry in the `devIntfTable` for this device, listing the device name and entry point.
2. Create the `.h` file for the device interface. Look at `devices/merlintf.h` for an example. The only requirement is that there is a function prototype for the device interface's entry point. The integrator expects the function prototype as shown in `devices/merlintf.h`.
3. Create the `.c` file containing the device interface code.
4. Add an entry to `devices/makefile`.
5. Type `make` in `devices/`.
6. Type `make` in the ASIAGO directory.
7. Add an entry to `conf/devices.conf` describing the device interface. The format of this file is described in section 3.2.2.
8. Use the device in a integrator specification in `conf/integrator.conf`. The format of this file is described in section 3.2.4.

7.5 Event Recognizer Specification

The procedure for adding an event recognizer (ER) is slightly more flexible than for the other modules. The ER is started up in `eventRecognizersInit()`; however, the function is hard-coded as `erTask()`. If this name is acceptable, all that must be done is to replace the entry in the ASIAGO makefile which refers to `demo_er.c` (the current location of `erTask()`) with a different module containing an `erTask()`.

An event recognizer has the following function prototype:

```
void
erTask(void *arg);
```

`arg` is a pointer to the ER's control block. This function is called during the initialization phase of ASIAGO (see figure 3). The ER must perform any initialization it requires (such as registering interests with the buffer manager, getting pointers to buffers, or registering a timer wakeup from the communication manager), and then it must call `waitForConditionVarAssertion()` to sleep.

When woken, the ER should examine the `controlBlkCommand` field of its control block. If it contains `CB_EXIT`, the ER thread should exit. If `controlBlkCommand` contains `CB_DATA_READY`, the ER should perform whatever processing it needs, and then possibly issue commands to ASIAGO using the ASIAGO API. For example, it may place data objects into the buffer manager (using `commMgrCommandIntf()`), issue status messages, or shut down the system. Refer to Table 5 for a list of ASIAGO API calls.

To add a transformation function to ASIAGO:

1. Create a `.c` file containing `erTask()` as described above.

2. Replace `demo_er.c` in ASIAGO's makefile with the new `.c` file.

3. Type `make` in the ASIAGO directory.

To start up multiple ERs, changes must be made to `eventRecognizersInit()` so that each ER is passed a unique control block pointer.

8 xasiago

`xasiago` is a MOTIF user interface to ASIAGO. `xasiago` was designed as a general interface for status monitoring, mode creation, mode selection, transition specification, and execution control of ASIAGO. As with all other communication to and from ASIAGO, InterAgent is used to interface `xasiago` to ASIAGO. Though unfinished, the following capabilities exist:

- As modes are parsed by the mode manager, a mode specification message is issued. This includes the mode name and description. These modes are listed in the `xasiago` mode selection window.

- As integrators are started, each sends a message to `xasiago` stating the integrator name.

- Mode change requests may be issued from `xasiago` to ASIAGO, using the list of modes and integrators.

- Certain error messages are issued to the operator via `xasiago`.

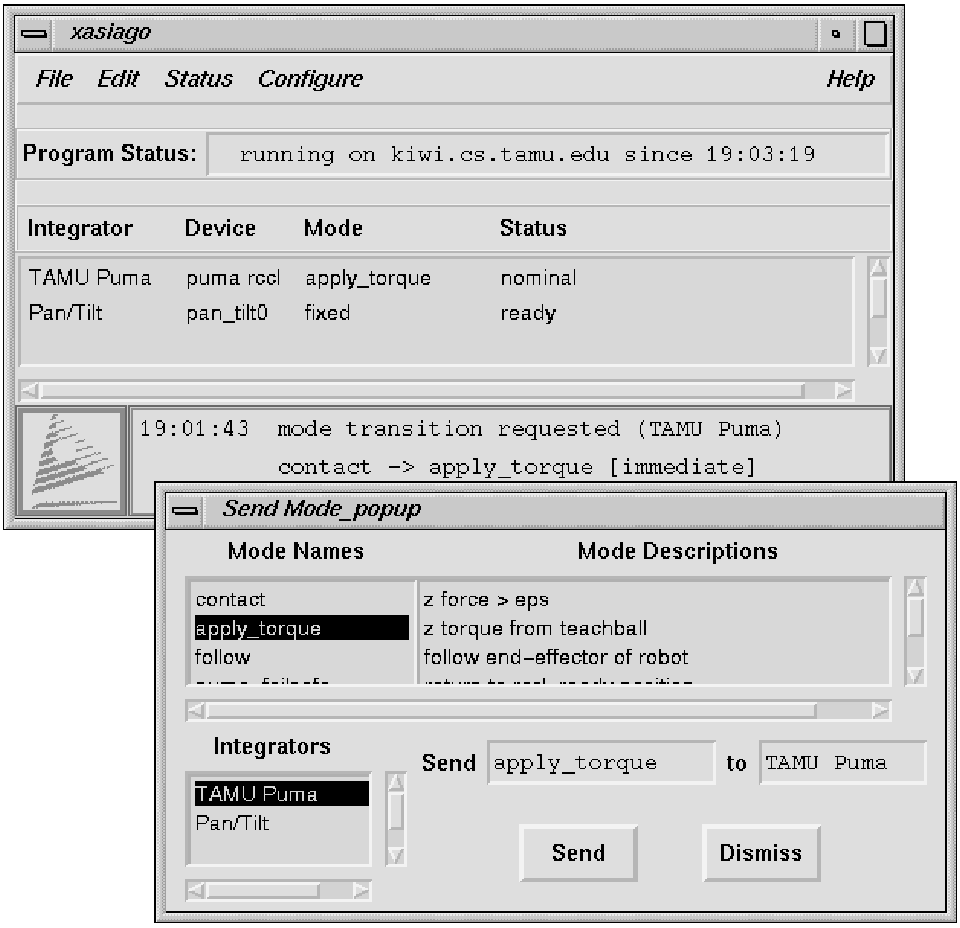

Figure 5: A typical session of `xasiago`

- User-supplied modules may also issue messages for display by `xasiago`.

A typical session of `xasiago` is shown in Figure 5. The only menu item which actually works (besides File|Exit) is Configure|Modes, which causes the smaller window shown in the figure to appear. In this window, the user is allowed to select a mode and an integrator and send a command to ASIAGO instructing it to perform a corresponding mode change. No technique for specifying the transition is given.

To start up `xasiago`, type `xasiago -- -s`*hostname*, where *hostname* is the name of the machine where the InterAgent router is running. This should be the same router to which ASIAGO is connected.

9 Things left unfinished

There are numerous functionalities which are not fully implemented. The user should be aware of these limitations, and is encouraged to complete any of the following that sound interesting:

- **Spline transitions** – The ASIAGO design calls for the capability to perform mode changes using immediate, linear interpolated, and splined transitions. A splined transition would allow the user to design a transition profile which was non-linear, and could generate some interesting effects. Currently, the data structures in the mode manager support splines, but there is no code to specify the spline in `xasiago`, or to generate λ accordingly.

- **Sequenced buffers** – ASIAGO was designed to support two types of buffers: sequenced and immediate. An immediate buffer is similar to a memory location, where an update will immediately supersede a previous value. A sequenced buffer would have a time value associated with it, where each command would have a certain lifespan. This would be useful for sending complete 'trajectories' of commands to ASIAGO. There could also be looping or branching constructs to allow complex command execution ordering. Again, the data structures support such buffers, but there is not a method for specifying that a buffer should be sequenced, or of specifying the lifespan of a command. New code would have to be added to the buffer manager to cause it to age commands in the buffers, so that a call to `bufferMgrRead()` would return the current 'live' command.

- **External status reporting** – As stated on numerous occasions in this manual, external status reporting is not generalized. The configuration files and data structures support the specification of the types and periods of data that should be send out on the network as status. However, the communication manager does not use this information. It would be fairly easy to establish another thread specifically for this purpose, which would read specific buffers and distribute their contents externally.

- **On-the-fly mode specification** – One of the advantages of a generalized mode representation is that it can be used to handle many situations not previously envisioned. However, there is no current method for sending new modes to ASIAGO as it is running. This could implemented using an ER which would register an interest in special mode specification objects, and would then make calls to the existing mode manager functions to define the modes.

- **Other primitive data types** – The only primitive types which are currently supported are `TYPE_FLOAT`, `TYPE_INT`, and `TYPE_STRING`. This means that no basic data types can be defined using other types of data, such as homogeneous transforms, quaternions, etc. These kinds of data must now be represented using an array of `TYPE_FLOAT`. It would be useful to define other primitive types. This would require changes to `datatype.c` and `integrat.c` (specifically in `calcActionVector()`) to handle these other types.

- **Complete `xasiago`** – `xasiago` should be expanded to allow the examination of buffers, transitions, event recognizers, etc. Mode creation capabilities should be added.

- **Event recognizer startup** – Currently, one ER is supported with a fixed name. A method similar to that used for transformation functions should be devised so that ASIAGO core code is not affected by adding new ERs.

- **Synchronization** – The ASIAGO design describes a method of providing buffer synchronization ([1] section 6.7.2). This should be implemented.

- **Buffer Priorities** – It may be useful to devise a buffer priority scheme, so that the data from multiple buffers can be selected based on priority.

- **Dynamic linking** – Dynamic linking should be used to add user modules.

References

[1] B. S. Graves, "A generalized teleautonomous architecture using situation-based action selection," Ph.D. dissertation, Texas A&M University, College Station, TX, 1995.

[2] 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.

[3] Lynx Real-Time Systems, Inc., *LynxOS Version 2.1 Reference Manual*, Lynx Real-Time Systems, Inc., Los Gatos, CA, July 1992.

[4] Modulus Technologies, Inc., *InterAgent Toolkit Reference Manual*, Modulus Technologies, Inc., Houston, TX, 1993.

Index

N

P

R