The HKU Behavior Server for Atlas
Wyatt Newman
April 9, 2013

1) **Introduction**: Direct control of Atlas through ROS is via publications of joint-level angles, joint velocities, and/or joint efforts (torques). To perform useful tasks (e.g., "open the door") will require introducing multiple layers in a hierarchy and abstracting from joint-level commands to task-level commands. Although there is a single gateway to control of the robot (via 28 body joints and 2, 12-jointed hands), there will be a potentially large number of "skills" (e.g., turning a valve, turning a steering wheel, turning a door handle, climbing, crawling, walking, operating a power tool, …). Skills may involve coordinating all (or most) joints (e.g., for entering/exiting a vehicle), or only a few (e.g., moving the right arm towards a hand goal location, or tilting the sensor head). Some tasks are best performed in joint space (possibly vehicle ingress/egress, possibly climbing) and some are best performed in task space (e.g. reaching for an object at sensed Cartesian coordinates). Some tasks will need to be performed concurrently, such as steering with the hands/arms while operating brake/gas with foot/leg, where gas vs. steering would be treated more conveniently as separate tasks.

A "behavior" should abstract away necessary, lower-level details to perform some higher-level function. An example is to command the body joints to achieve some desired pose. At the higher level, the appropriate level of abstraction is to think in terms of desirable outcomes (e.g. a desired pose). To achieve this outcome effectively, one must provide a stream of intermediate goals that result in the robot gracefully converging on the desired pose. From the perspective of the higher level, one may think in terms of a coarse sequence of destination poses (e.g., pre-position the right arm to an appropriate pose to approach an object of interest with the right hand, move monotonically towards the object until an appropriate pre-grasp pose is achieved, grasp the object, …). Each of these intermediate goals must be broken down into a sufficiently fine granularity of joint commands that are updated at a sufficiently high rate (e.g. 10Hz to 100Hz). The higher-level code should not be concerned with the trajectory-generation details, but it does need to know when a sub-goal has been achieved, so a new goal can be introduced. Typically, this cannot be based on pure timing, but would be based on outcomes.

More generally, even low-level behaviors (such as performing interpolation for smooth joint-space moves) are associated with "events." An implicit event is reaching a specified goal, resulting in the expected outcome (the arm is now where I asked it to be). More generally, a behavior may terminate reactively in response to environmental input—e.g., move the arm until the hand feels contact. Other events may include time-out (to prevent deadlocks when there is failure to progress), a reflexive halt to prevent impending collision, reaching a singularity or joint limits, extending full reach without a (Cartesian) goal achievable, a user-invoked Emergency Stop or pause/resume, vision-based recognition of an object or fiducial of interest, etc. Thus, we may think of "behaviors" as actions coupled with "events."

Designing higher-level "skills" will require coupling events and actions, where (typically sensor-based) event triggers will influence the actions. A low-level behavior server must be able to accommodate future, as-yet unforeseen skills, and thus introducing new "events" needs to be a simple, extensible process. Further, recognizing that events may be the result of substantial perceptual processing, event detectors should be implemented as separate nodes, decoupled from the behavior server.

It is also recognized that there is a need for a control layer beneath that of the behavior server, operating at lower level of abstraction but at a higher loop rate. Notably, feedback for maintaining balance or for performing gentle, compliant manipulation will require high update rates (e.g.

1kHz). Maintaining balance and behaving compliantly may be modulated from a higher level by specifying parameters, but the high loop-rate computations must be performed at a low level of abstraction.

This document describes our current (ground-up) approach to constructing initial layers for a behavior server. A low-level joint controller resides between a "behavior server" and direct control of Atlas's joints. A behavior server communicates with one or more higher-level behavior clients, which specify actions to be performed coupled with events to be monitored. Separate event-detector nodes encapsulate the necessary signal processing to recognize events of interest, and results of such nodes are broadcast to be available to any process or specific behavior. The behavior server offers joint-space or Cartesian-space motion options involving an arbitrary collection of joints. The behavior server supports execution of multiple, simultaneous behaviors (provided they do not require use of conflicting joints). The behavior server also coordinates hand control with body control, treating control of fingers as additional behaviors, which may be executed sequentially or concurrently with body motions.

The behavior-server architecture is described herein in terms of multiple, interacting packages of: low-level joint controller, behavior server, event detectors and behavior clients.

2) **The Low Level Joint Controller**: The low-level joint controller resides in the low_level_joint_controller package. At present, this is merely a skeleton, passing inputs on to Atlas. The low-level joint controller receives joint commands from the behavior server as published on the topic "/lowLevelJntCmd", using the (locally defined) message type "hku_msgs::lowLevelJointController." *Only* the behavior server should publish to this topic, or results may be unpredictable and dangerous. The behavior server is responsible for coordinating multiple, concurrent behaviors involving disjoint sets of joints. Such coordination should be performed by one entity (the behavior server), and only this entity should send commands to the low-level joint controller.

The low-level joint controller publishes its output to "/atlas/joint_commands," which is the topic used for sending commands to the physical (or simulated) robot. *Only* the low-level joint controller should publish to this topic, as this layer is the gateway to physical motion.

Although the current low-level joint controller is merely a pass-through of joint commands from the behavior server to the robot, it will grow in sophistication to incorporate high-speed control issues, including balance control, force control, protection against joint-limit violations or self collisions, and potentially reflexive collision avoidance with objects in the world. The low-level joint controller is central to issues of dynamics and control theory. While it is expected to grow dramatically in sophistication, the current input/output interfaces are not expected to change significantly. Thus, developments in the low_level_joint_control package should occur simultaneously with higher-level software developments without conflicts.

3) Event and State Publishing and Listening: Perceptual processing is expected to be the largest proportion of code in the system. To accommodate ongoing change, it is desired to decouple perceptual processing from motion control as much as possible. At the same time, execution of tasks that are responsive to sensation of interactions with the environment (e.g., touching an object, moving towards a visualized goal, performing vision-based steering, etc) require coupling perception with action. The present architecture approaches this challenge through the notion of "event detectors." We presume that perceptual processing will be performed in nodes that are distinct from the behavior server, but which must communicate with the behavior server. Ultimately, there may be a large number of specialized event detectors, and the architecture should

support a simple means to incorporate them as they are developed.

The present approach to managing multiple, concurrent, asynchronous event detectors is inspired by ROS's "transform listener." The transform listener receives data published to a single topic by a potentially large number of transform publishers. The nodes that introduce and update individual parent-child frame relations may come or go unpredictably, new transforms may be introduced dynamically, and the transform data may be published chaotically. The transform listener is responsible for acquiring available data (as published to the transform topic), organizing this data, and providing services to any consumers needing results of operations on this data.

In our approach, the "state_trigger" package defines classes of "broadcaster" and "listener" for events and states. In some instances, a behavior may care about the "state" of some status (e.g., is the force on my hand greater than 20N?). In other cases, a transient "event" may be of concern, where an "event" is typically a time-derivative of a state (e.g. "tell me when the hand makes initial contact").

The broadcaster class provides a designer with the means to communicate a state or event of interest that is compatible with a complementary listener. A broadcaster instance would be included in any event-detector node. At present, the only user of the event listener is the behavior server (although use of the event listener is an option for higher or lower levels as well, if desired).

Distinct event detectors broadcast their results using unique (string) names. It is not necessary for the event listener to be altered (e.g., no changes to header files, editing code, nor recompilng) in order for it to process new broadcast events. As long as an event-detector node uses the provided event (and state) broadcaster—using a unique name for the signal of interest—the listener will accept the new data and make it available to any interested consumers.

The events_and_states package should be included in the manifest of any packages that use an event broadcaster (e.g., an event-detector node) or event listener (e.g., the behavior-server node).

The events_and_states package merits further explanation of its theory of operation, use and capabilities, but this documentation is deferred for now.

4) Event Detectors: Although a potentially large number of specialized event detectors is anticipated, there is only one example implemented at present. This resides in the package "event_detectors," which includes the (only) node "hand_contact_sensor." The source file for hand_contact_sensor illustrates how to build an event detector. This simple (and certainly not optimized) event detector is intended to broadcast if the force on the right hand exceeds some threshold (hard-coded, in this example, to 15.0N), as well as provide the ability to detect transitions from low to high (below the threshold to above the threshold) or high to low. Since the input signal is noisy, it was first low-pass filtered, with cut-off frequency tuned by the value of the variable "a_filt." (A value closer to 1.0 results in a lower cut-off frequency, and a value closer to zero makes the filter have less influence).

This node gets its input from the topic "/atlas/force_torque_sensors/". This is typical; most event detectors will need to subscribe to some source of sensory information, which may then be processed appropriately by the detector node to determine if an event of interest has occurred.

The event detector uses an instance of state_trigger::Broadcaster to broadcast its results.

In addition to using an event/state broadcaster, this example also publishes the analog value of the low-pass filtered force data to the topic "filtRhandForce"; this is not necessary but is useful for

visualizing data, monitoring operation or tuning filter and threshold values (e.g. using rxplot).

Use of the event listener involves C++ 11 extensions. To incorporate these extensions, the Cmakelists.txt file must include the line: #comment: Use C++11 standard SET(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++0x").

Also, the manifest for an event-detector package must include dependency on the state_trigger package: <depend package="state_trigger"/>. If new event detectors are added to the existing event_detector package, these details are already incorporated in the CMakeLists.txt and manifest files.

5) The Behavior Server: The current implementation of the behavior server is also skeletal, but it is anticipated to grow in sophistication. At present, the behavior server only performs trajectory generation for smooth, coordinated movement of selected sets of joints—for the body and/or for fingers of the left and right hands. Anticipated extensions include Cartesian motion of hands, feet, elbows or knees, and coordinated body motions that shift center of gravity while maintaining no-slip conditions at support points (e.g., grasped ladder rungs, or knees and wrists during crawling). Additionally, coordinated single-limb motions—useful for crawling, walking or climbing—may become implemented behavior modes within the behavior server.

The behavior server uses the "action library" within ROS. The action library is intended to support invocation of and feedback from "actions." Actions may be considered at high levels of abstraction (e.g., "do the dishes") or low levels (e.g. "move the right hand to the following pose"). The present behavior server implementation is at a low level (limited to simple motion goals).

The action library supports multiple action clients and concurrent action executions. An action client communicates with an action server through an interface defined by an "action" file. In our package "behavior_server_v2", there is a directory "action" that contains a single file, "behavior.action." This simple file has a format like that of ROS message definitions, except that it is divided into three fields: goal, result and feedback. A programmer defines the contents of each of these fields. The "goal" field specifies all of the information to be sent by an action client to the action server in requesting a new action. In the present behavior-server implementation, a goal is specified either in joint space (with an array of desired joint angles specifying a joint-space destination, plus an array of booleans specifying which joints are involved) or (anticipated) in Cartesian space, via a stamped pose. Additionally, a (variable-length) vector of "events" may be specified for conditions that should invoke a halt (e.g., sensed contact). An arbitary number of events may be considered for any one action. A "behavior type" element specifies the type of action to be performed. At present, functional actions are JOINT_SPACE_MOVE, RIGHT_HAND_MOVE (control of the fingers of the right hand) and LEFT_HAND_MOVE. These codes are specified in the "include" file "behavior_codes.h". Packages referring to this definition file may include it with:
#include <behavior_server_v2/behavior_codes.h>. The list of functional behavior codes will expand in the future.

The behavior.action file also specifies components of the "result" field. An action server communicates back to an action client by specifying these values in a "result" message when the corresponding action concludes. An action may conclude normally (e.g., reaches its goal destination) or with an error (e.g., out of reach or impending collision), E-stop, or cancellation (as requested by the client), or due to one of the terminating events firing (e.g., detected contact). At present, the information returned to the client by the server only contains an integer return code and (if terminated by an event), the name of the responsible event. The return codes are defined in

behavior_codes.h, at present limited to COMPLETED_MOVE_ON_SCHEDULE or EVENT_FIRED. Additional codes (e.g. “user E-stop” or “reflexive safety halt”) are anticipated. The return message may also be expanded to include more detailed information about the conditions upon conclusion.

A third field of behavior.action is provided for feedback to the action client while the action is in progress. As with the “goal” and “result” messages, it is up to the user to define and populate these fields. The goal message gets transmitted when the client requests an action, and the result message gets transmitted back to the client when the corresponding action concludes. For intermediate feedback, the user must explicitly include in their implementation publication of feedback status. The initial implementation of the behavior server does not use feedback, although it is illustrated in the code how to do so with a dummy message. (see line: feedback.status = 1234; //dummy for test).

Use of the action server requires specific additions to the manifest (<depend package="actionlib"/> and <depend package="actionlib_msgs"/>) and to the CmakeLists.txt, including:

```
rosbuild_find_ros_package(actionlib_msgs)
include(${actionlib_msgs_PACKAGE_PATH}/cmake/actionbuild.cmake)
genaction()
rosbuild_genmsg()
```

See the CmakeLists.txt and manifest files in the behavior_server_v2 package for an example.

Threaded support for callbacks also requires specifying links with the “boost” library. (See the CmakeLists.txt file for behavior_server_v2).

Behaviors are requested by clients through specification of the elements within the “goal” field of the behavior.action file, as described above. The behavior server interprets the request in the context of the specified behavior_type item, and it considers termination in terms of “normal” completion (reaches a specified goal) or occurrence of one of the associated events.

Within the behavior-server code, new behavior types may be included by editing three regions of the code. A new behavior code would add a new “case” to the switch/case statement, e.g. as: case JOINT_SPACE_MOVE: The new behavior should include code in: a block for initialization (computation of values or conditions to be used in incremental updates); normal (non-event) termination; and (primarily) how to update the action incrementally each iteration of the behavior-server loop (at frequency “loopFreq”, currently set to 100Hz). In the existing code, this is accomplished through member functions of special-purpose objects, such as: smoothJointSpaceMove.update(selectedJntArray, qOut, qDotOut); for interpolated joint-space moves.

6) Action Clients: Behaviors are initiated by action (or behavior) clients communicating with the behavior server. Examples are included in the behavior_server_v2 package.

A simple example is “action_client_nodder.cpp”. This example defines an instance of an action client, “ac”, which communicates with an action server named “hku_action_server” via the line:

```
actionlib::SimpleActionClient<behaviorAction> ac("hku_action_server", true);
```

A goal message, as defined in the behavior.action file, is declared as: behaviorGoal goal;
This goal is then populated with the elements selectedJoints[], jointAngles[], behavior_type, and duration. The fields for a vector of terminating event conditions and Cartesian pose are left empty,

since the desired action does not depend on any event detectors.

The prescribed action is then requested of the behavior server through the member function:
ac.sendGoal(goal, &doneCb, &activeCb, &feedbackCb);

The callback functions inform the client of if/when the action is started, any intermediate feedback provided by the action server, and the result message transmitted by the server upon conclusion of the behavior. In the present example, the client requests motion exclusively from the neck joint (with all other values of selectedJoints set to "false"). It requests that the head tilt down (to an angle of 1.0 rad) over a duration of 3.0 seconds. It waits for news of completion of this action, then initiates a request to return the head upright (also over 3.0 seconds). The process is repeated in an infinite loop, resulting in the robot nodding its head up and down indefinitely.

As an illustration of ability to run concurrent behaviors, additional action clients may be run simultaneously. The action client "action_client_lhand_grabber" causes the left hand to open/close its fingers alternately. This node may be rosrun at any time, resulting in the hand motion being executed at the same time as the head nodding.

A third example, action_client_interactive, may also be run (individually or simultaneous with other action clients). This example prompts the user to enter a joint number and a joint-angle value, then executes the specified joint motion (over a default 2.0 seconds).

A more sophisticated example is "action_client_w_sequencer". This example is based on the DRC qualifying task 2, in which the robot needs to grasp a drill and deposit it in a bin. This example uses a "behavior sequencer," which initializes a (circular) queue of behaviors. The action client "pops" behaviors from the queue and requests that behavior of the action server. Each new behavior is invoked upon being informed that the prior behavior has concluded. The third behavior in this example includes responsiveness to the right-hand contact event detector. The intent is that the robot approach the drill with its right hand, stop when contact is detected, close the hand, then lift the drill. This example is *not* tuned to successfully manipulate the drill. Approach angles, speeds, event-detector filtering and thresholding all need to be tuned. Additional motions are also required to move the grasped object above the bin and release it.

7) Launch Files: Running the behavior server with its supporting nodes is tedious to start up manually. Two launch files can be found in the behavior_server_v2/launch directory. The launch file "behavior_server.launch" will start up the low-level joint controller and the behavior server. With these two nodes running (plus the drc simulator), one can rosrun any of the example action clients.

The grabber.launch file starts up four nodes: the low-level joint controller, the behavior server, the right-hand contact event-detector node, and the "action_client_w_sequencer" action client.

Running: roslaunch atlas_utils qual_task_2.launch
together with: roslaunch behavior_server_v2 grabber.launch
will result in the Atlas robot attempting to pick up the drill. (With parameter tuning of this example code, the robot should perform this task successfully).

8) Conclusion: The current status of the behavior server is primitive, limited to joint-space motion control of body and finger joints. Only a single example of an event detector exists. The low-level joint controller merely passes joint commands from the behavior server to the atlas joint controller. The action clients are simple examples invoking crude behaviors. Nonetheless, the structure of the architecture is illustrated with these elements, and contributions can be added by emulating these

examples. It is intended that each of the elements of this architecture will support extensive growth.