CWRU ARIAC 2018 software architecture
Wyatt Newman
4/8/17

Robot Behavior Action Server and Robot Behavior Interface:

To help simplify our ARIAC solution, robot-specific code has been encapsulated in a robot motion action server. This action server needs to be started separately, e.g. with:

rosrun kuka_move_as kuka_behavior_as

This robot action server is specific to the Kuka IIWA 14 robot. However, it responds to generic part-manipulation commands.

The objective of the robot action server (called kuka_behavior_as.cpp and node kuka_behavior_as, in package kuka_move_as) is to abstract the robot, so that the higher levels can focus on operations on parts, independent of robot specifics. This also supports substituting alternative robots with maximum code re-use.

The behavior action server takes goals that are based exclusively on part manipulations. The robot action server is blind, i.e. does not have access to any cameras or other sensors, besides its own body parts (such as joint-sensor values and gripper states). The robot action server is responsible for controlling all joints in feasible, collision-free trajectories and for controlling the vacuum gripper (and for sensing attachment/release and dropped parts). Collision avoidance depends on specifics of the robot and specifics of the environment, so this action server can be quite lengthy and tedious in its design. However, it is hopefully easy for higher levels to use.

Commands to the robot action server are via goals, as defined in RobotBehavior.action in the kuka_move_as package. This action message contains goal fields to hold: an action code, a timeout value, and "Part" descriptions for a sourcePart and a destinationPart. Not all action codes require all arguments. Example action codes include: PICK_PART_FROM_BIN, PLACE_PART_IN_BOX_WITH_RELEASE, ADJUST_PART_LOCATION_WITH_RELEASE, PICK_PART_FROM_BOX, DISCARD_GRASPED_PART_Q1, and RELEASE. A "timeout" value can be included in the goal. If no timeout value is provided, the default timeout is
const double MAX_BEHAVIOR_SERVER_WAIT_TIME = 30.0; //to prevent deadlocks
as defined in the KukaBehaviorActionServer.h header file. However, the user does not need to be aware of these internal details, since a simplified interface class has been provided.

The class/library "RobotBehaviorInterface" can be used by higher levels to communicate with the robot behavior action server. This interface is responsible for packaging "goal" messages appropriately, sending them to the action server, and providing callback functions for "result" messages to be returned. Functions available in the robot behavior interface library include:

bool pick_part_from_bin(Part part, double timeou);
bool place_part_in_box_no_release(Part part, double timeout);
bool place_part_in_box_with_release(Part part, double timeout);
bool release_and_retract(double timeout);
bool discard_grasped_part(Part part, double timeout);
bool release(double timeout);
bool adjust_part_location_no_release(Part sourcePart, Part destinationPart, double timeout);

bool adjust_part_location_with_release(Part sourcePart, Part destinationPart, double timeout);
bool pick_part_from_box(Part part, double timeout);

These functions are prototyped in "RobotBehaviorInterface.h" corresponding to the library "RobotBehaviorInterface". By instantiating an object of this library, a higher-level program can invoke robot behaviors, e.g. with calls such as:
robotBehaviorInterface.place_part_in_box_with_release(part);

The "Part" message is a CWRU-defined message (defined in package "inventory_msgs") that includes the part name (a string), a location code (e.g. a bin or box ID), and a geometry_msgs/PoseStamped pose. The pose is typically expressed with respect to the "world" frame. (If expressed in any other frame, this is noted in the frame_id, and the pose can be transformed to the world frame). By passing such an object to one of the robot-behavior functions, the robot can know where it must send its gripper, including knowing whether it must carefully insert the end effector through shelving to reach a part. Part properties, such as part thickness, are also known to the robot action server. This enables the robot to adjust pickup and dropoff locations to account for part heights.

Fulfilling an order would typically consist of pick and place operations. Once a desired part is found in inventory, its description can be used in the **pick_part_from_bin()** behavior. If this operation is successful, the next behavior to invoke is **place_part_in_box_no_release().** It is then useful to evaluate the part in terms of the corresponding quality sensor, to assure it is good or not. If the part does not pass the quality inspection, it can be discarded by invoking the **discard_grasped_part()** behavior. If the part passes the quality test, its pose can be evaluated before releasing it, using **adjust_part_location_no_release().** Once the part is assured to be good and to be properly placed, it can be released with **release_and_retract().** This process can be repeated for each part required to fill a shipment.

All of the behaviors return a bool, indicating success or failure of the operation. These return values should be inspected to guide error handling.

BinInventory class/library:
The "bin_inventory" package includes a library, "bin_inventory" (source code: bin_inventory.cpp) that defines the class "BinInventory", which performs inventory assessment of parts in bins. The main function is:
bool update();
This function attempts to get a camera update from every bin camera, which it uses to update the bin inventory. If the cameras do not respond within time:
const double BIN_INVENTORY_TIMEOUT=2.0;
the **update()** function returns "false", and the inventory remains unchanged.

A program that instantiates an object of type BinInventory, e.g. as:
BinInventory binInventory
can invoke an inventory update with:
binInventory.update();
This will return "true" if all cameras responded, in which case the inventory will be updated. Otherwise, the previous inventory will still be present in memory.

The a complete copy of the current inventory can be obtained with the call:

inventory_msgs::Inventory inventory_msg;
binInventory.get_inventory(inventory_msg);

Keeping a copy of the inventory can be useful in the event of camera "blackouts". The update() function can be attempted, and if it returns "true", then a fresh copy of the newly-assessed inventory can be requested. However, if the camera updates are unsuccessful, processing can still continue using the copy of inventory. In this case, when parts are picked from bins, the inventory in memory should be updated automatically by deleting the picked part, which can be done with:

binInventory.remove_part_from_inventory(part_id, partnum_in_inventory,current_inventory);

The "partnum_in_inventory" value is necessary to specify which item of type "part_id" should be removed from the inventory list (std::vector). This partnum is available from requests for a part type. Specfically, calling the function:

binInventory.find_part(current_inventory, part_name, pick_part, partnum_in_inventory)

will return "true" if the part type specified by "part_name" exists within the "current_inventory." If so, this function will fill out a Part object, "pick_part", and it will also specify which part from inventory was selected (partnum_in_inventory).

Note that it would have been simpler to choose the first part from inventory of the specified type. However, if there are problems grabbing this part, one could get stuck in a loop retrying the same part repeatedly. Thus find_part() chooses one of the available parts at random.

Typically, a BinInventory object would be used via the functions "update()" and "find_part()" (and remove_part_from_inventory(), in the case of blackouts). The program using a BinInventory object thus should not be concerned with details of cameras, coordinate transformations, or choosing parts. These details are encapsulated, with the result that one can easily obtain the location/pose of a part of interest in any one of the bins by merely specifying the desired part's name.

BoxInspector class/library:

The BoxInspector is responsible for comparing a shipment description to the observations from a camera (either at quality-inspection station 1 or 2). It recognizes parts that are present and precisely located, are present but imprecisely located, are missing, or do not belong. This information can be used to make corrections, including repositioning parts, removing orphaned parts, and confirming status with the shipment-filling operation.

Order Manager class/library:

This class needs much work. It subscribes to the topic "/ariac/orders", on which new order requests will be sent during the competition.

The order manager queues up incoming orders. Note, however, that it is expected that order requests will seldom need queueing. Also, it should be recognized that orders that arrive before a current order is filled are "priority" orders. Orders that arrive prior to fulfillment of an existing unfilled order are queued up in a separate array of "priority" orders (although it is anticipated that this queue will never have more than one order).

The order manager instantiates an object of type BinInventory, which it uses to evaluate fillable orders. Unfillable orders are placed in a separate queue ("vector", really). In practice, it is expected that the system will seldom have more than one order at a time, and thus a sub-optimal (unfillable) order may be the only available option.

The order manager picks an order to be filled, which may be comprised of multiple "shipments." In choosing shipments to be filled, the order manager should choose priority shipments over non-priority shipments.

In an additional challenge, an existing order could receive an "update". At present, this is *not* handled by the orderManager.

ShipmentFiller:
The ShipmentFiller class is a library of utility functions useful for manipulating parts. The primary value of this library is the function:
shipmentFiller.get_part_and_place_in_box(current_inventory,place_part)
which returns "true" if the desired part pick-and-place is successful (else, false). This function takes arguments of the current inventory and the desired part-placement outcome (place_part). The function will look for the desired part type in the inventory, select one from inventory, perform a "pick" operation to acquire the desired part, then attempt a "place" operation to achieved the desired outcome. If there are errors along the way, this function will re-try until it is successful, or until there are no more options left in inventory.

If get_part_and_place_in_box() returns successfully, then it is assured that a part of the desired type is placed in the box. It is *not* assured that this part has passed the quality inspection, nor that the part is precisely located. These tests should be performed after the desired manipulation, which only guarantees that a part of the desired type is in the box (and is still grasped).

The get_part_and_place_in_box() uses member functions of the classes BinInventory (to find a desired part in inventory) and RobotBehaviorInterface (to perform part manipulations).

The ShipmentFiller class should be extended with more helper functions, including handling part inspections and part relocations, as well as possible part removal/replacement at quality-inspection station 2.

Additional functions in ShipmentFiller are used to control the conveyor and call a drone with a specified shipment label (which invokes scoring).

Conveyor-control actions should be extracted from ShipmentFiller and implemented as an action server, as described next.

Conveyor Action Server:
This does not yet exist, but elements of it are implemented in the ShipmentFiller class. Control of the conveyor should be spun off to a separate action server, which can advance boxes as desired and call the drone when boxes arrive at the loading dock. Further, the conveyor action server could monitor the trial time, and conclude with shipping any boxes with partial shipments (to maximize score as time is expiring). Possibly, the conveyor action server could send an "interrupt" (e.g. via a service hosted by the top-level program) that time is nearly up, filling should cease, and any partial shipments should get sent out. It is at least necessary that the arm is withdrawn from boxes and ceases to attempt more part placements.

The conveyor action server will need to be sensitive to camera dropouts. Some limited control could be done with dead reckoning, but there are some critical poses of the boxes that can only be known adequately with camera inspection.

Example top-level control node: simple_shipment_filler
The simple_shipment_filler node starts the competition (via a service call to an OSRF service), then begins to receive and fill orders. To help do so, it uses functions from the classes: OrderManager, RobotBehaviorInterface, BinInventory, BoxInspector, and ShipmentFiller.

The current simple_shipment_filler is capable of: receiving an order, attempting to fill it, to the extent possible, sending the result to the loading dock, and calling a drone to deliver the shipment. This currently scores a total of 2 points, with a shipment containing a single item that is correctly placed.

Much more logic needs to be written to create a more sophisticated shipment filler.

Running the simple shipment filler:
First, start the ARIAC simulation:

(on Atlas6)
rosrun osrf_gear gear.py -f `catkin_find --share --first-only osrf_gear`/config/quals/qual2a.yaml ~/ariac_ws/ariac-docker/team_config/team_case_config/qual2_config.yaml

start the robot behavior server:
rosrun kuka_move_as kuka_behavior_as

start the shipment filler:
rosrun shipment_filler simple_shipment_filler

The shipment filler will run to completion and the resulting score will be shown in the window from which the simulator was launched.