A Client/Server Approach to Open-Architecture, Behavior-Based Robot Programming

Wyatt Newman, Adam Covitch and Ryan May
Department of Electrical Engineering and Computer Science
Case Western Reserve University
Cleveland, Ohio 44106
wnewman@case.edu

Abstract:

This paper describes our progress in the development of a behavior-based, stimulus-response robot control software architecture that is expressly designed to program and execute interactive tasks, including assembly, multi-robot collaborative manipulation, and exploration. The system has been designed to assure stability and safety while maintaining flexibility and achieving expert performance. The architecture incorporates a reactive controller as a behavior server, and applications are written as client programs that can operate either locally or across a network. This organization has been demonstrated to be sufficiently open to support teleoperation tasks as well as human-guided supervisory control and full autonomous functionality.

1. Introduction:

Extending human reach and achieving sustainable presence in space will require dramatic advances in robotics. Robots are expected to precede humans to Mars, where they would perform a variety of sophisticated tasks, including: preparing landing sites, constructing thermal and radiation barriers, assembling power systems and communications systems, prospecting for *in situ* resources (e.g. minerals containing extractable oxygen and hydrogen), assembling and operating factories for resource extraction, assembling habitats, and continuously performing inspection, maintenance and repair of facilities. These tasks are currently beyond the capabilities of robots operating on Earth, much less in space. In contrast, current Mars rovers operate by moving only centimeters per command, and each command has a time lag of approximately one hour between updates from human operators on Earth. At these rates, no significant landing-site preparations could be performed within the lifetime of a robot.

Achieving the required robot competence will depend on advances in robot autonomy. Rather than expecting incremental motion commands from Earth, robots will have to receive much higher-level, more abstract goals. The robots will have to behave competently to achieve specified subgoals while simultaneously taking responsibility for self preservation as well as protection of critical and possibly delicate components and systems with which they interact.

In pursuit of more autonomous and more competent robot behavior, we are constructing a behavior-based, stimulus-response robot control software architecture that is expressly designed to program and execute interactive tasks, including assembly, multi-robot collaborative manipulation, and exploration. This system is based on "schemas", comprising collections of stimulus-response behaviors that are simultaneously relevant in some context. Figure 1 shows an analogy between creature-like behavior and a behavior-based robot program. In this analogy, a creature is engaged in an initial action, but it is simultaneously monitoring myriad conditions that may induce a switch to an appropriate new action. In the analogous robot program, the initial action is to move towards a wall. The "event" of sensing contact with the wall causes the robot to change its action to move along the wall. During this operation, the robot controller should be monitoring and interpreting a variety of sensory conditions to select appropriate action sequences, to protect itself and objects in contact, to recognize and respond to abnormal conditions, and to guard against deadlock.

While the desired behaviors could be programmed in conventional if-then-else constructs, such an approach is tedious and error prone, particularly as the number of conditions to be monitored within a given context grows. Instead, it is useful to think of a collection of state machines and event detectors, and to program via selection of a state machine appropriate to a given context, and specification of event-driven transitions among state machines. The present work describes a system that supports programming in this manner.

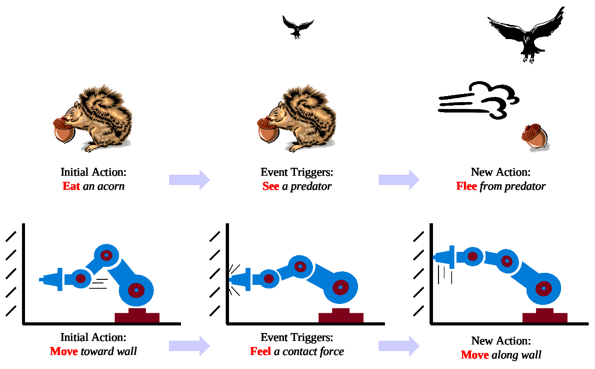

Fig 1: Analogy between creature behavior and the robot behavior-server architecture.

2. Interaction Dynamics and Assembly:

It is noteworthy that the competencies required of space exploration robots would also be of great value in conventional manufacturing. At present, industrial robots are seldom utilized in tasks that require regulation of interaction dynamics, such as mechanical assembly, grinding, polishing, window-washing, and manipulation of fragile or elongated objects. Further, reprogramming industrial robots is tedious and costly, which constitutes a barrier to greater robot employment. Achieving higher robot autonomy, easy and effective reprogrammability from a distance, inherent self preservation and greater task-performance expertise would serve space exploration needs as well as open up new markets for terrestrial robots.

In this presentation, we emphasize issues of compliant-motion interaction dynamics, notably mechanical assembly. Competence in mechanical assembly will be required for constructing, maintaining and repairing essential systems robotically.

It has long been recognized that sensation of and responsiveness to contact forces is crucial in performing mechanical assembly tasks competently (see, e.g. [1]). In some cases, the mapping from sensation of forces to physical response can be encoded mechanically, as in the successful "remote center of compliance" device [2]. However, such a static mapping is limited to niche applications. A more flexible, programmable stimulus-response behavior is required for accomplishing mechanical assembly more generally.

A variety of methods have been reported in this area of research, including use of fuzzy logic, neural nets, logic programming, and hybrid dynamical systems. These techniques may be employed within the context of behavior-based robotics (see [3] for an excellent overview). The approach most relevant to the present work is that of hybrid dynamical systems, as exemplified by McCarragher in [4].

In our previous research, collaborative between CWRU, Ford Motor Co., MicroDexterity Systems and the National Center for Manufacturing Sciences, we were able to demonstrate the capacity for an impedance-controlled robot to perform relatively difficult mechanical assemblies—components of automotive transmissions (see, e.g. [5,6,7]). These components could not be assembled by conventional position-controlled industrial robots. In a laboratory setting, six assembly examples were performed robotically with results competitive with human manual assembly [7]. The use of force controlled robotic assembly systems is now increasing within Ford Motor Company. There are currently four force-controlled robotic applications and another planned for the immediate future [8]. This research demonstrated the capacity for compliantly-controlled robots to perform difficult mechanical assemblies, and it offers hope for addressing some of NASA's daunting challenges.

While force-controlled robots are making inroads in industry, implementation efforts have also exposed a weakness: each application had to be painstakingly hand coded and tuned in a respective custom program. The level of difficulty, software development expense and programming expertise required present obstacles to greater industrial utilization. This weakness is an even greater barrier for robots acting remotely in unstructured environments, where they must be reprogrammed from a distance, and where programming errors may not become apparent on Earth before significant damage is done.

A viable robot control architecture and programming interface should support encoding manipulation skills that exploit compliant-motion capability. At the same time, the controller should be capable of achieving subgoals competently while assuring safe operation. Instead of communicating desired coordinates, an application program should invoke sequences of behaviors.

3. Behavior-Based Control for Manipulation

Compliant-motion control offers the possibility of making robots perform interaction tasks more like a creature than like a machine tool. Force-control capability also introduces new safety and effectiveness concerns. These include: joint or workspace constraints; contact force/moment constraints; velocity constraints; system fault detection and response; and stalling (failure to progress).

For a conventional position-controlled system, joint and workspace limits may be evaluated by simulation prior to execution to assure constraints are not violated. Also, excessive contact forces or system failures can be detected by monitoring servo errors in real time. The conventional response to excessive servo errors is to automatically shut the system down.

Under compliant-motion control, however, the actual trajectory of the robot is an emergent property of the situated agent, resulting from the interaction of virtual forces from the compliant attractor trajectory and physical forces from environment interaction. Since the resulting joint and workspace excursions depend on the environment, advance simulation is inadequate; instead, joint and workspace constraints must be monitored at run time. Similarly, endpoint forces and moments must be monitored directly, since large errors between a compliant attractor and the robot end effector may be intentional or emergent. Under compliant motion control, unlike position control, the actual trajectory of the robot may also fail to progress (e.g., due to an unexpected obstruction that prevents the robot from reaching a goal state). To prevent program deadlock, use of compliant motion should also incorporate watchdog timers to detect and respond to failure to progress.

We thus see that the benefits of a behavior-based, compliantly-controlled interactive system also incur costs in terms of additional burdens to monitor safety and progress. New interfaces should enable a programmer to exploit compliant-motion capability easily, should incorporate means to monitor safety and progress, and should be sufficiently flexible to accommodate timely and appropriate responsiveness to multiple stimuli.

Our system consists of three layers of control, shown schematically in Fig 2. The lowest level is the impedance-

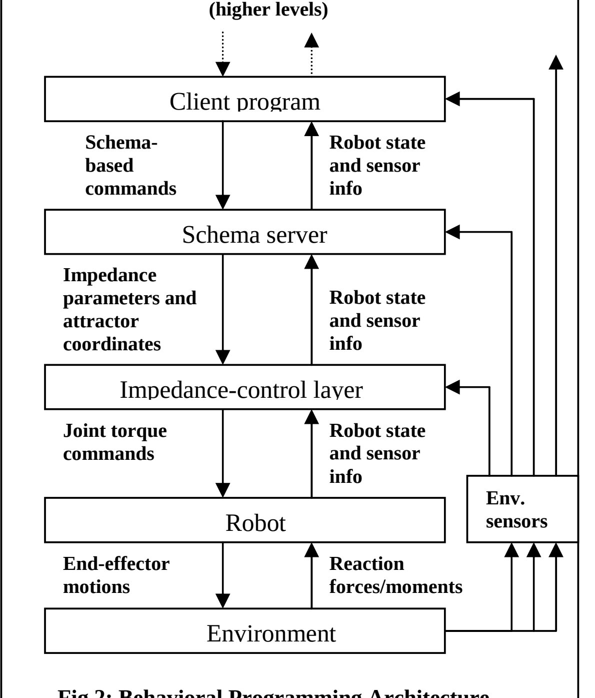

Fig 2: Behavioral Programming Architecture

control layer, which achieves gentle compliant motion with guaranteed interaction stability (see e.g. [9-12]). The second layer is a "schema server", the heart of our system, which is responsible for accepting and encoding parameters of a reactive controller that interacts with the impedance-control layer. The third layer is the deliberative layer, which incorporates the programmer's logic and communicates incremental schemas to the schema server for nominally sequential execution of low-level, event-driven reactive behaviors.

4. Primitive Action Functions

In constructing a reactive system, an essential issue is the definition of the building-block elements, including how to implement and interface to sensory processing and actuation.

Our compliant-motion layer interface helps to frame the design problem for the stimulus-response layer of control. Specifically, a higher level must specify commands to the compliant-control layer in terms of impedance parameters and an attractor trajectory. We have approached this problem by defining a set of action primitives, describable in these interface terms.

To encode an assembly skill, we attempt to describe the skill as a collection of simpler action "primitives." To gain some insights into how to identify and organize such primitives, we have analyzed instrumented human demonstrations of assembly operations [13]. While no two instances of human demonstrations are identical, repetitions are subjectively similar. In our interpretation of such data, in the spirit of Brooks [14], we presume that the complexity observed emerges as a result of a simple underlying strategy interacting with a complex environment. In development, we propose and test simple action functions to evaluate if these candidate primitives are consistent with human performance. Those primitives that are found to be useful are installed as options within the controller.

A trivial but useful action is "sleep" or "idle". This action maintains a constant command to the underlying impedance controller. Idle may be persistent for a specified duration, e.g. while waiting for transient dynamic effects to decay before proceeding with a subsequent action. Idle is also useful while waiting for satisfaction of preconditions before advancing to a subsequent action.

Some action primitives are conceptually "atomic", since they can be executed in the robot controller within a single iteration. Examples include: opening or closing a gripper (e.g., a one-shot digital output command to a solenoid); setting the virtual impedance values (spring and damper parameters) and relieving reaction forces (by immediately setting the attractor coordinates equal to the end-effector coordinates). These actions are analogous to reflexes, since they do not persist beyond excitation of the stimulating event.

More interesting action primitives correspond to fixed-action patterns [15]. Such actions are commonly initiated by a transient sensory event, as is the case with a reflex action, but the fixed-action pattern persists after the stimulus has ceased. The pattern may persist for a fixed duration, or it may be terminated by another relevant sensory event. To describe these primitive actions in our system, we design corresponding functions that may be called repetitively (at a relatively high, fixed frequency), producing incremental action updates resulting in smooth evolution of robot states.

A useful fixed-action pattern is to command motion of a soft attractor at a constant speed along a straight-line path (e.g., in 6-D for a typical robot arm). For simplicity, we assume that the impedance parameters are held constant for the duration of each action-primitive function. We have found that the apparent manipulation skills utilized by humans in performing many useful assemblies can be adequately modeled with a coarse sequence of straight-line attractor trajectories [13]. Note that while the attractor trajectory of an action primitive is linear, the corresponding motion of the robot itself may be much more complex, due to dynamic interaction with the environment (e.g., while complying with a kinematic constraint).

The action function for a compliant-motion command requires specification of the instantaneous attractor coordinates, desired attractor speed, and desired attractor direction vector. A call to the compliant-motion action function increments the soft attractor coordinates along the desired direction vector by a step size proportional to the desired speed and the update period.

A slightly more complex action, the "blind search" has also been found to be useful [5,6,7], e.g. for vertical-stack assembly. In this action, the robot applies a preload by pressing a grasped component against a work surface. The robot slides the part in a spiral search pattern while oscillating the part about a vector normal to the surface. This technique is frequently effective in hunting for insertion coordinates when the part grasp and sub-assembly fixturing uncertainty exceeds the assembly clearance. Such uncertainty is inevitable in unstructured environments, such as assembly in space or on remote planets.

By constraining consideration to the above primitive action functions (a set which may be expanded, as necessary), we define a framework for encoding robot skills that consists of identifying and prescribing the following unknowns:

- How many primitive actions to invoke and in what sequence
- What parameters (e.g. direction, speed, and impedance values) to assign to each primitive-action function
- How to decide when to switch from action *i* to action *j*.

Transitions among actions are invoked by recognition of sensory events; this is performed at the second layer, the "schema server" layer.

5. The Schema Server

Rumelhart et. al. [16] trace use of the term "schema" from Kant (1787), Bartlett (1932), Piaget (1952) to more modern usage. Arkin [3] reviews additional historical usages, and adopts a working definition: "A schema is the basic unit of behavior from which complex actions can be constructed; it consist of the knowledge of how to act or perceive as well as the computational process by which it is enacted." (see [3], pg 43). This definition is consistent with our present usage.

Our "schema server" processes behaviors encoded as sets of action primitives associated with sensory events. Simultaneously, the schema server constantly monitors safety and progress. The schema server is instrumental in making the system reactive, but it does not incorporate strategic and deliberative processing (which occurs at higher levels). The schema server refreshes and processes sets of stimulus-response pairs, as prescribed by a higher level. In our system, only one schema is active at any instant. However, each schema may include many stimulus-response pairs that are simultaneously pending.

At the schema-server level, there is always a single schema in context. Consistent with the current schema, there is always one and only one enabled primitive action (which may be the "idle" action). This primitive action may be atomic or may be persistent. This is in contrast to systems in which multiple actions from multiple simultaneous stimuli are blended (e.g. additively, as in superposition of potential functions).

A prioritized list of stimulus-response pairs is contained within a schema specification, constituting a reaction table. Each stimulus-response pair requires specification of:

- How to process a sensory signal to determine if an event of interest has occurred
- From where to fetch the relevant schema in response to an event trigger

By default, we include a list of high-priority events responsible for monitoring safety and progress. These include events that trigger when exceeding joint or workspace constraints, events that trigger when measured force or torque values exceed safety constraints, and an event corresponding to a watchdog timeout, indicating failure to progress. The default response to these events is to halt the machine, relieve interaction forces, and appeal to a higher level for error recovery.

By handling safety and progress concerns implicitly, the programmer may focus on specifying each intentional stimulus-response association for performing a specific task. Typically, this will consist of a sequence of actions and expected sensory responses. Each deliberate action is encoded within a separate schema. The schema server processes each schema and advances to the next schema, as driven by the sensory events. The source of the next schema is specified for each sensory event. Sources include: pre-encoded schemas in memory (equivalent to innate reflexes and fixed-action patterns); an emergency communications channel (required for responding to conditions for which there are no pre-encoded contingencies); and a command buffer (which is fed from a higher level with schemas to be performed sequentially under normal conditions).

Each time the schema server fetches a new schema, it installs a new current action (typically parameterized), a new focus of attention (sensors of interest), new alarm levels (conditions for sensors to trigger events), and new associations paired with sensory events (respective sources for a schema to be installed in response to the event).

Given an installed schema (including a currently-active action primitive), the schema server loops, performing the following operations:

1. Block (suspend) pending a timer signal
2. Check high-priority channel and respond to any interrupts from higher levels.
3. Read all sensors and update a table of sensor values and sensor interpretations
4. Scan the list of sensor values of current interest, in order of importance, evaluating the highest-priority sensory event (if any) that has occurred. (This is a "triggering event")
5. In response to the triggering event (if any), install a new schema from the source associated with the event.
6. Perform one iteration of the current primitive action.
7. Loop back to step (1)

Step 1 establishes the repetition rate of the schema server, as regulated by a timer signal from the real-time operating system. In our implementation, the schema server repeats its loop at 200Hz.

Upon receiving the timer signal, the schema loop proceeds in step 2 to check for messages from a high-priority communications channel (implemented as a Unix socket). Via this channel, higher levels can exert commands such as E-stop, abort, or may respond to fault conditions. This interrupt channel may also be used for direct human intervention. The schema server interprets commands from this source as schemas, encoded in XML format. If the schema server receives a schema packet via the high-priority channel, this new schema is immediately installed in the reaction table, overwriting the previously active schema. Further processing within the schema server is subsequently under the control of this new schema.

In step 3, the schema server samples all relevant sensors and updates the values of these sensors in a table. Some of these sensors may be hard-wired to the robot controller, and others may interact as network services. Alternatively, the sensor-table refresh operation may be performed asynchronously by parallel processes, e.g. through shared memory. Such processes may include relatively complex signal processing (e.g. pattern matching). Regardless of implementation, the objective of this step is to update the interpretation of all sensors of interest.

An unusual but useful event that may be included in a schema is the "one-shot" event. The status of the one-shot virtual sensor is initialized to FALSE upon installation of a new schema. During step 3, the status of one-shot is changed to TRUE. This event is useful for executing a schema that contains an atomic action primitive that should be performed once only.

In step 4, the sensor-value table is evaluated to determine if any signal-processing results have satisfied triggering conditions (e.g., by exceeding respective specified thresholds). All sensory events are prioritized. If any events are observed to have occurred during the current execution cycle of the schema server, the highest-priority event is defined as the triggering event.

In step 5, if there is a triggering event, the schema associated with that event is installed. This step involves listing a new set of prioritized stimulus-response pairs, resetting all event flags, and installing a new current action function. The response to a one-shot event is treated the same way; installing a new schema due to a one-shot event enforces only a single iteration of an atomic action.

The source of the next schema to be installed is crucial in determining the behavior of the resulting system. One may define the source of the next schema to be a location in memory where predefined schemas exist. Sensory events described in these schemas may, in turn, point to other hard-coded schemas. In this manner, one may encode arbitrarily complex finite-state machine behaviors. Alternatively, events may point to an input command buffer as the source of the next schema. In that case, the robot controller would behave similar to a simple peripheral, executing commands sequentially as determined by some higher level "client."

Note that the schema server is never a complete slave to its client. Although normal sensory events may point to the command buffer for new schemas, the schema server also continues to monitor its safety and progress conditions. If one of these higher-priority events occurs, the schema server will redirect its actions per the associated schema sources. For example, the force sensor may trip an excessive-force event. We associate this event with a hard-coded schema that performs the atomic action of setting the attractor position equal to the end-effector position. This action causes the low-level controller to rapidly relieve the force between the robot and its environment. The one-shot virtual sensor is listed within this emergency-recovery schema, and the schema source associated with the one-shot event is the high-priority queue. Thus, the system responds by ignoring pending schemas in the command queue, processing a schema that performs an emergency corrective action, then appealing to a higher level for further error-recovery instructions.

Between these extremes (automaton vs slave peripheral), finite-length linked schemas may be pre-encoded and utilized. An event may point to an "innate" schema (hard-coded in memory), and execution of this schema may invoke successive links through additional hard-coded schemas, but ultimately point back to the command queue. In this manner, one can encode a relatively complex skill consisting of a sequence of actions, and this skill may be invoked as part of a sequence within a strategy. Such encoding is analogous to sophisticated but non-cognitive behaviors executed in the spinal cord or brain stem.

In step 6, the schema server commands execution of one increment of the current action function. This may be as simple as invoking an atomic action (e.g., open gripper or set impedance parameters), or it may advance the attractor incrementally along a specified vector. Note that a schema containing an atomic action would be installed in step 5, and execution of that action would occur in step 6. On the next iteration of the schema-server loop, the "one-shot" virtual sensor would be set to TRUE, invoking an event leading to installation of a new schema in step 5. Using this processing logic, atomic and persistent actions may be encoded in a common schema format.

As noted, the schema server may draw its commands from linked lists of pre-encoded schemas. If these lists are constructed cleverly enough, the resulting reactive system may perform interesting and useful operations, potentially robustly. However, a system consisting of a static set of linked schemas would be difficult to debug, inconvenient for implementing adaptation (learning) and inappropriate for performing higher-level processing (e.g. planning). Higher-level layers are needed.

6. The Deliberative Layer

Our third layer (and final layer, in the current system) is implemented as a client of the schema server. In our client/server construction, the client program may run within any computing environment, independent of operating system or programming language, provided it can connect to the server via internet protocol. The deliberative-layer client communicates with the schema server by encoding and transmitting schemas in XML format. The schema-server receives and processes schemas, and it transmits sensory and status data back to its client. The client program may transmit schemas either to the server's input command buffer, where they will be executed sequentially, or to the high-priority channel, where the server will execute each new schema immediately, pre-empting commands in the queue.

Our client programs have been developed and executed on Windows-based PC's. Wrapper functions have been written to ease the task of converting schemas into XML format and transmitting the schemas to the schema server. A set of Application Programming Interface (API) functions for specifying common schemas eases program development.

An example API function is: *moveTo (pose)*
This function constructs and transmits a schema that causes the robot to move to the specified desired position. A default speed is assumed (which may be overridden), and the most recently imposed set of impedance parameters is used during the move. In execution, the attractor is moved at constant speed from the robot's initial position to the desired final position along a straight line in Cartesian/orientation space. This is encoded as a persistent attractor speed along a specified vector. The normal triggering event that terminates this action occurs when the attractor position passes a threshold value.

This function typically behaves similar to the corresponding conventional position-controlled robot command. However, its implementation differs in its reactive behavior. The robot is attracted towards the goal pose, but it may be impeded by an obstruction. If the obstruction is near the goal location, then the robot will respond by exerting a relatively low force against the obstruction. If the reaction force exceeds a safe value, the schema-server will observe triggering of an excessive-force event and will react appropriately. Such error detection and reaction is implicit by default.

A more obviously reactive API that is useful in compliant-motion programming is:

moveToTouch(dir_vec)

This function encodes and transmits a schema that causes the robot to move compliantly from its current pose along the direction vector "dir_vec" until contact with the environment exceeds a threshold force. The speed of approach and the threshold force are optional arguments with nominal default values.

7. An Assembly Example

Figure 3 shows the ParaDex robot and an assembly task. The task consists of inserting 6, 15mm-thick aluminum geometric shapes into corresponding recesses in a baseplate. The assembly clearance for these parts was +/-0.05mm, and the parts were not chamfered, making assembly challenging.

The client program for inserting these shapes consists of repetitions of the following sequence of calls:

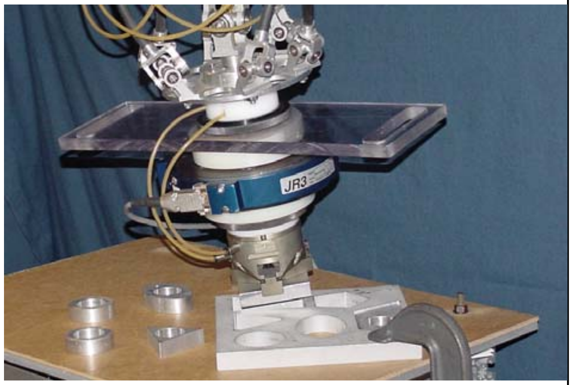

Fig 3: The ParaDex Robot and Example Assembly Task

```
moveTo(above_part);// approach part pick location from
                              //above
moveToTouch(dir_down); // approach the part vertically
                              //from above; stop upon contact
grasp(); // atomic command: grasp part
moveTo(above_part); // depart vertically w/ part
moveTo(above_assem_loc); // move to location above
                    // the approximate assembly location
moveToTouch(dir_down); // approach the sub-assembly
                    // from above; stop on contact
blind_search(); // with vertical pre-load, move part in a
          // search pattern terminating when insertion is
          // detected
release(); // atomic command to release grasped part
moveTo(above_assem_loc); // depart vertically; ready
   // for next assembly command
```

Each of these function calls within the client program results in the construction of a schema, its encoding in XML format, and its transmission to the schema server running on the robot controller. Parameters may be altered to tune the assembly performance (e.g., as in [6,7]).

The nominal part-acquisition location and approximate insertion coordinates may be pre-taught (if feeding and fixture coordinates are at least approximately known), or these coordinates may be obtained from sensors (e.g., machine vision).

The above program was successful in assembling all 6 geometric shapes. Using a blind search, although successful, was relatively slow. With an initial error of 0.3mm in translation and 0.02rad in rotation, blind-search assembly times averaged approximately 7 seconds for all parts. As the uncertainty in assembly location increased, search times also increased, roughly doubling at 1.2mm/0.05rad initial error. Using a more sophisticated strategy incorporating a sequence of 4 triggered behaviors, all parts were assembled with a mean time less than 3 seconds over the above range of uncertainties. This latter strategy was roughly 1 second slower than human assembly rates.

8. Conclusions

Although behavior-based robotics is often associated with basic research in artificial life, techniques from behavioral programming can be applied fruitfully to very pragmatic robot programming problems. This paper has presented our approach to behavioral programming for exploitation of force control in constrained manipulation. The system presented responds to the needs for safety, effectiveness and ease of programming. Continued progress in this direction will help to move behavior-based robotics research into demanding applications.

Acknowledgments: This work was supported by ABB Inc., US Corporate Research Center and by NASA Glenn Research Center under grants NAG3-2578, NAG3-2799 and NNC4AA12A. Experimental work on the ParaDex manipulator was made possible through equipment loans from Ford Motor Co.

References:

1. Whitney, D. E., "Historical Perspective and State of the Art in Robot Force Control," Int. J. of Robotics Research, Vol 6, No 1, Spring 1987, pp 3-14.
2. Whitney, D.E., “Quasi-Static Assembly of Compliantly Supported Rigid Parts,” J. Dyn. Sys. Measurement and Control, V104, 1982, pp62-77.
3. Arkin, R. C., Behavior-Based Robotics, MIT Press, Cambridge, MA, 1999.
4. McCarragher, B.J.; "Force sensing from human demonstration using a hybrid dynamical model and qualitative reasoning", ICRA ’94, pp 557 -563 vol.1
5. Hebbar, R., W. Newman, et al., *Flexible Robotic Assembly for Powertrain Applications*; ICRA 2002 video proceedings. (best video award)
6. Morris, D.M.; Hebbar, R.; Newman, W.S. *Force guided assemblies using a novel parallel manipulator,* ICRA 2001, Page(s): 325 -330 vol.1
7. Wei, Jing and Newman, W. S., *Improving robotic assembly performance through autonomous exploration,* ICRA 2002,Page(s): 3303 -3308.
8. personal communication w/ David Gravel, Ford Motor Co., AMTDC, 8/29/04
9. Newman, W. S., and Zhang, Y., “Stable Interaction Control and Coulomb Friction Compensation Using Natural Admittance Control,” *Journal of Robotic Systems,* Vol. 11, No. 1, pp. 3 - 11, 1994.
10. Newman, W. S., “Stability and Performance Limits of Interaction Controllers,” *Transactions of the ASME Journal of Dynamic Systems, Measurement, and Control,* Vol. 114, No. 4, pp. 563-570, 1992.
11. Dohring, M.; Newman, W., *Admittance enhancement in force feedback of dynamic systems,* ICRA '02, pp638-643.
12. Hebbar, R. and Newman, W. S., *Passivity Analysis of Sampled-Data Interactive System,* ICRA 2003 pp 3309-3314.
13. Newman,W.; Birkhimer,C.; Hebbar,R.*; Towards automatic transfer of human skills for robotic assembly* IROS 2003,Pages:2528 – 2533

14. Brooks, R. A., Cambrian Intelligence, MIT Press, Cambridge, MA, 1999.
15. Smith, W. J. The Behavior of Communicating: An Ethological Approach, Harvard University Press, Cambridge, Ma, 1977.
16. Rumelhart, D.E. et. al., “Schemata and Sequential Thought Processes in PDP Models”, in Parallel Distributed Processing, Vol 2, MIT Press, Cambridge Ma, 1986.