Using a Robot Control Architecture to Automate Space Shuttle Operations*

R. Peter Bonasso and **David Kortenkamp**
Metrica Inc., Texas Robotics and Automation Center
NASA Johnson Space Center – ER2
Houston, TX 77058
bonasso@mickey.jsc.nasa.gov

Troy Whitney
North Dakota State University
Fargo, ND

Abstract

This paper describes preliminary results from using an AI robot control software architecture, known as 3T, as the software framework for a procedure tracking system for the space shuttle Remote Manipulator System (RMS). The system, called 3TPT, is designed to track the expected steps of the crew as they carry out RMS operations, detecting malfunctions in the RMS system from failures or improper configurations as well as improper or incomplete procedures by the crew. Scheduled for a ground demonstration in February 1997, and a test flight the following fall, 3TPT, was employed this past fall to track the RMS checkout procedures on a space shuttle mission. It successfully carried out its task because the reactive nature of the architecture allowed it to stay synchronized with the procedures even in the face of intermittent loss of telemetry and unexpected crew actions.

Introduction

The Orbiter Upgrade Program is a significant effort to streamline space shuttle operations in all phases by moving flight controller assistance from the ground to on-board the spacecraft. Such assistance will take the form of automatic procedure tracking and verification, caution and warning monitoring and malfunction procedure execution, and fault isolation detection and recovery. All of the orbiter main functions, e.g., propulsion, guidance, navigation and control, communications, etc. are to be automated in this manner.

The first such function is the Payload Deployment and Retrieval System (PDRS) which uses the shuttle's Remote Manipulator System (RMS). This system, known as RMS Assistant, is to provide the procedural and analytical knowledge of the PDRS flight controller to the on-board crew in the form of a collective set of functions including automatic procedure tracking and verification, activity logging, fault isolation and recovery, and payload operations replanning.

The RMS is a teleoperated robot, so it was decided to use an existing intelligent robot control architecture called 3T to implement the automatic procedure tracking (PT) and verification portion of the RMS Assistant project, called 3TPT. This decision has the important advantage of allowing for increased autonomy in RMS operations in the future. We first give a brief overview of the 3T intelligent robot control architecture, show how it is being applied to the RMS Assistant project, and describe how it was used during a recent live shuttle mission.

The 3T robot control architecture

The three tiered robot control architecture (3T) has been in development in one of several forms since the late 80s (Firby 1987; Gat 1992; Connell 1992; Bonasso *et al.* 1995). Space does not allow for an extended history of this architecture (a useful historical summary can be found in (Gat 1997)). Its development stemmed from the frustration of AI researchers at the failure of "sense-plan-act" architectures to bring AI reasoning to bear on real-world robots. Brooks claimed that for many tasks, robots did not need traditional AI reasoning (Brooks 1991) and backed that claim with robust autonomous robots that used essentially a "sense-act" architecture. But others sought to integrate traditional reasoning systems with the reactive style, a kind of "plan-sense-act" approach. What emerged was a software control architecture with a lowest layer of reactivity, a topmost layer of traditional AI planning, and a middle layer which transformed the state-space representation of plans into the continuous actions of the robot.

The 3T version of this architecture has been used at NASA Johnson Space Center since 1992 in a variety of space robot research programs (see (Bonasso *et al.* 1997) for a work summary). 3T, shown in Figure 1, separates the general robot intelligence problem into three interacting pieces. First, there is a set of robot specific, real-time reactive skills, such as grasping, ob-

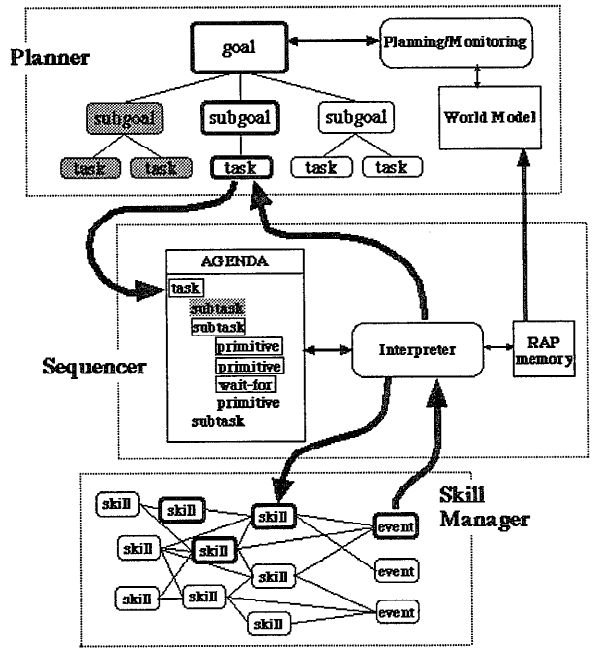

Figure 1: The 3T architecture

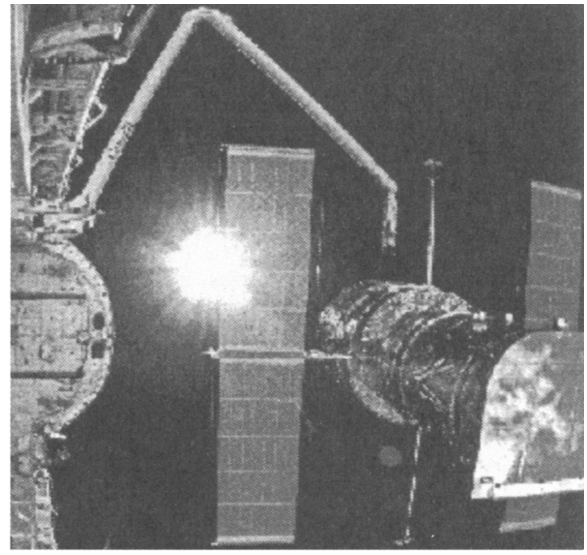

Figure 2: An image of the shuttle's remote manipulator system

ject tracking, and local navigation, which are tightly bound to the specific hardware of the robot (Yu, Slack, & Miller 1994). The next layer up is a sequencing capability which can activate the reactive skills in order to direct changes in the state of the world and accomplish specific tasks. For example, exiting a room might be activating and deactivating sets of reactive skills for door tracking, local navigation, grasping, and pulling. We are using the Reactive Action Packages (RAPs) system (Firby 1995) for this portion of the architecture. At the top layer there is a deliberative planning capability that reasons in depth about goals, resources and timing constraints (Elsaesser & MacMillan 1991). This top tier was not used in this initial implementation of 3TPT. It will be used in pre-flight mission planning and for on-board mission replanning in later flights.

Applying 3T to the RMS

The RMS, shown in Figure 2, is a six degree of freedom arm, fifty feet long and 15 inches in diameter, with a capturing device on the end. It is controlled from the shuttle's aft flight deck with two hand controllers and a set of switch panels.

In applying 3T to the RMS we had several different testbeds, each of which was used for different purposes. First, there is a high-fidelity simulation of the RMS developed at NASA JSC. This simulation runs on Silicon Graphics workstations and has displays similar to what the astronauts use. In addition, it uses the same TCP/IP-based protocol as is used by the actual RMS, so connecting to simulation data is no different than connecting to real RMS data (see the next subsection). The RMS simulation was useful in debugging our software and it also allowed us to experiment with autonomous control of RMS operations, something that will not be possible on the real system until the orbiter is upgraded. The second testbed that we used was a playback system which generated telemetry from log files of previous shuttle missions that used the RMS. The resulting telemetry is identical to that of actual shuttle missions and let us check for problems with data dropouts and noise, which are not modeled in the simulation. However, we cannot control the execution in order to test different execution paths as we can with the simulation. Finally, we ran the system in real-time against actual shuttle data during shuttle mission STS-80. This offered the opportunity to test the system in a situation similar to its delivery environment. In the next two subsections we briefly describe how we connected 3T to the testbeds described above.

RMS skills 3T's hardware controlling skills are C subroutines that perform some action or monitor sensor values. There are three types of skills in 3T: blocks control a piece of hardware or perform a specific computation; events monitor sensor values and detect and report key state changes back to the middle tier; and queries interpret sensory data as requested by the mid-

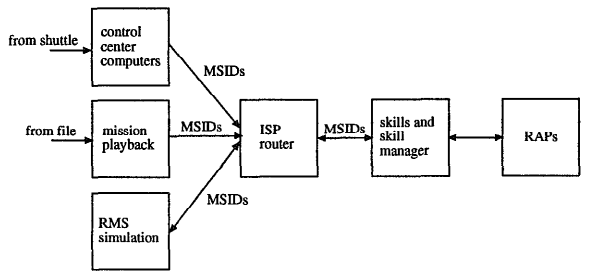

Figure 3: Communications network for distributing MSIDs.

dle layer. A *skill manager* program is responsible for moving data among skills using shared memory, for scheduling CPU access by the skills and for communicating with the middle tier of the architecture via a TCP/IP socket.

Examples of RMS skills we implemented are blocks: set-brakes, select-power, activate-cameras, drive-joint-to, and test-joint; events: cameras-on, joint-switch-at-position, correct-joint-response, arm-not-moving, and joints-within; and queries: camera-state, brakes-status, joint-switch-position, rms-mode, and joint-angles. Of course, when the 3TPT cannot control the RMS autonomously, blocks are replaced by requests to the operator, however, all events and queries can be used in either autonomous or teleoperated mode.

To send commands and receive data, the skills use a NASA JSC publish/subscribe communication protocol called Information Sharing Protocol (ISP, see Figure 3). Each piece of space shuttle information is assigned a unique message identifier called an MSID. Processes subscribe to MSIDs in which they are interested and publish MSIDs that control pieces of hardware. ISP distributes MSIDs to all processes that need them. During shuttle missions, NASA JSC's Mission Control Center (MCC) publishes for use by authorized computers actual MSIDs generated by the general purpose computer (GPC) on the shuttle. Each of our testbeds uses the same MSID tags for the same information. Thus, the simulation publishes MSIDs as if it were the actual RMS GPC. Connecting to the simulation is then no different from connecting to the real space shuttle; only the source of the process generating MSIDs is different.

RMS RAPS When a shuttle mission calls for the RMS, the crew first performs a series of functional tests designed to ensure that the RMS and all of its related controls are working properly. We have automated these tests in the second level of 3TPT with a set of RAPs which include testing various switches and responses on the control panels, testing that the RMS joints can be driven in a variety of control modes, and testing the hand controllers for proper operation. We explain the operation of these RAPs by discussing our implementation of the single drive test procedure in detail in the next section.

The Single Drive Test

The single drive test involves a crew member driving each of the six joints via the on-board GPC to determine that all joints can be commanded individually. The crew member first places the RMS in single drive mode, then, using switches on the control panel, selects and drives each of the joints in both the positive and negative directions, while watching a rate meter on a switch panel.

Because 3T was designed for autonomous operations it will be useful to first describe its operation for this procedure as if the RMS was fully autonomous. The RAP for this procedure, in a shortened form of the LISP-based RAPs language, is shown below:

```
conduct-single-drive-test
  succeed:       (all-joints-tested)
  preconditions: (rms-power-up procedure complete)
                 AND
                 (on-orbit-initialization
                   procedure complete) AND
                 (rms-at pre-cradle-position)
  method standard-procedure
    task-net
      sequence
        T1: set-parameter-switch-p A8 joint-angle
        T2: set-brake-switch-p A8 off
        T3: set-mode-p A8 single
        T4: verify-no-arm-motion-p
        T5: test-all-joints 10 45
        T6: set-brake-switch-p A8 on
```

When the above RAP is invoked it is placed on the RAP agenda for execution, and the succeed clause is checked by a deductive query of the RAP memory. If it is true, the goal is achieved, otherwise the truth of the preconditions are checked in the RAP memory before proceeding. RAP memory is updated by forms known as memory rules which will be discussed later, but such rules fired after previous procedures have been completed will have established the truth of the first two propositions in the preconditions. The third proposition is actually a function which invokes a query skill which returns the six joint angles of the RMS.

Primitive RAPs and skills

Assuming the preconditions to be true, the RAP interpreter then selects a method, here the standard-procedure method being the only one, and executes the plan of that method by posting each step to the

agenda and invoking the RAP each represents. Plan steps whose names end in "-p" are primitive RAPs. The first one, `set-parameter-switch-p` is defined as follows:

```
set-parameter-switch-p (agent value)
  succeed: (parameter-switch-position agent value)
  method autonomous
    primitive
        enable:     :set_parameter (:value value)
        wait-for:   :param_switch (:setting value)
                     (:timeout 15)
                     actual-value result
                    :succeed (actual-value
                                result)
        disable :above
```

When this RAP is invoked from the previous RAP, `agent` is bound to `A8`, the name of the control panel, and `value` is bound to `joint-angle`. Our 3T architecture can control and coordinate multiple agents, and in the 3TPT prototype, we have used this ability to group the skills for efficiency into those associated with the panel switches (A8), those associated with the arm motion (RMS), and those associated with the end effector (EE).

In this RAP we see the use of all three types of skills. The succeed clause is a primitive query to the RMS system for the state of the parameter switch, and `:set_parameter` and `:param_switch` are a primitive action and a primitive event skill respectively. If the parameter switch is at the joint-angle position, this RAP succeeds and the `set-parameter-switch-p` goal will be removed from the agenda. If the switch is in any other position, the RAP system simultaneously enables the action and event skills. These skills are C routines in the skill level of the architecture whose names are `set_parameter` and `param_switch` and whose arguments are `value` for the action and `setting` and `timeout` for the event. The `set_parameter` skill will command the state of the selected parameter in the RMS GPC, and the `param_switch` event will watch for the selected parameter value to change.

Assuming no other RAPs are waiting to be decomposed, the RAP interpreter will loop waiting for the `param_switch` event to fire. If the value does not change within the 15 second timeout, the event will return the setting of the parameter switch and the result `timeout`. Otherwise it will return the new setting and the result `okay`. After the event fires, it and the primitive action are disabled in the skill layer.

Memory rules can be optionally written for the start, finish and events of a RAP. Their left hand sides match on the returned values of the event's `:succeed` clause and their right hand sides can make changes to the RAP memory or can execute arbitrary LISP functions.

The operation of the RAP interpreter is designed for reactivity. If the switch is already in the correct position, no action is taken. If the event returns a timeout condition, the subsequent succeed query will notice that the RAP has not yet been successful and will enable the action and event skills again (the number of retries is a user-setable parameter; we use two). If this RAP fails, the top-level single-drive-test RAP method will fail causing the interpreter to try it again, thus also re-invoking the primitive RAP again. The combination of primitive queries and action retries makes for a robust system in the face of data loss or other related problems. If the reason the event times out is because of a temporary data loss, the RAPs system will eventually get the switch thrown if the data is re-established before the retry/query mechanisms are exhausted.

Iterative operations

Returning to the `conduct-single-drive-test` RAP, we see that in the `standard-procedure` method, primitive RAPs similar to `set-parameter-switch-p` are invoked for the brake switch and for setting the mode. `Verify-no-arm-motion-p` uses a no-op action and invokes an event that insures that no arm joint is moving. Then the heart of the test is invoked, the RAP to test all the joints as shown below:

```
test-all-joints (duration timeout)
  succeed:  NOT ((class-of ?jt joint ) AND NOT
                 (tested-both-directions ?jt))
  preconditions: (parameter-switch-position
                    A8 joint-angle) AND
                  ((current-rms-mode A8 single) OR
                   (current-rms-mode A8 direct))
  repeat-while: (new-axis-direction-has-been
                   tested?)
  method standard-procedure
  task-net
    T1: select-and-test-joint-p rms duration
                                  timeout
```

This RAP is also used for the direct drive test where the joints are driven directly from the power supply rather than through the GPC. As select-and-test-joint-p executes, memory-rules associated with its actions post `tested-both-directions` propositions to memory. The succeed clause is true when no item of class `joint` is in need of testing. The `class-of` relations are established as part of the RMS initialization. The `repeat-while` clause is a kind of recomputable precondition that insures that as long as there are new axis directions being successfully tested, this RAP will continue to execute until its succeed clause is true.

The real action takes place in the primitive RAP which is shown below:

```
select-and-test-joint-p (agent duration timeout)
  method auto+
```

```
  context: (next-joint-for-test ?j) AND
           ((joint-tested ?j -) OR NOT
            (joint-tested ?j))
  primitive
    enable:    :test_joint (:joint ?j)
                           (:direction +)
                           (:duration duration)
    wait-for:  :correct-joint-response
                 (:timeout 15)  joint direction
                                result
               :succeed (joint direction result)
    disable    :above

method auto-
  context: (next-joint-for-test ?j) AND
           (joint-tested ?j +)
  primitive
    enable:    :test_joint (:joint ?j)
                           (:direction -)
                           (:duration duration)
    wait-for:  :correct-joint-response
                 (:timeout 15)  joint direction
                                result
               :succeed (joint direction result)
    disable    :above
```

This RAP has two methods; if the context clause of one of the methods is true, that method is a candidate for selection. In the `auto+` method the context is true if the next joint to be tested has either already been tested in the negative direction or has not been tested at all. The `next-joint-for-test` query is a memory function that examines all the items of class `joint` to determine which ones still need to be tested in both directions. If more than one method is eligible, then other mechanisms not discussed, such as prioritization, will make the selection. In this case, the two context clauses are mutually exclusive, so only one method will be eligible each time the RAP is invoked depending on the testing status of the next joint to be tested.

The context clause binds the variable `?j` for use in the primitive skills invoked in the method. The `:test-joint` skill will drive the selected joint in the given direction for the specified duration and then stop. The `:correct-joint-response` skill will wait for a joint to start moving and will then check for the proper response, returning the joint that moved, the direction it moved, and a result of `okay` or `timeout` (if no joint moved within the 15 second timeout), or the joint malfunctions of `stalled`, `reverse tach` or `sluggish`. As discussed above, event memory rules will record the appropriate propositions to allow the parent and primitive RAPs to step through all the joints, as well as to record the malfunctions.

Variable autonomy

While the 3T architecture is well suited for autonomous operations as shown above, complete autonomy for the RMS Assistant is only envisioned for the far future. A key contribution of 3T to this effort is it's ability to handle variable autonomy. In particular, the very fact that our design begins with the assumption of autonomy, makes it directly amenable to variations on that autonomy. Below we show the `select-and-test-joint-p` RAP with the changes made for variable autonomy:

```
select-and-test-joint-p (agent duration timeout)
  method auto+
    context: (level-of-autonomy
               select-and-test-joint-p
               autonomous) AND
             (next-joint-for-test ?j) AND
             ((joint-tested ?j -) OR NOT
              (joint-tested ?j))

... etc.

  method auto-
    context: (level-of-autonomy
               select-and-test-joint-p
               autonomous) AND
             (next-joint-for-test ?j) AND
             (joint-tested ?j +)
... etc.

  method tele-operation
    context: (level-of-autonomy
               select-and-test-joint-p
               teleoperation)
    primitive
    enable:     tell-user "There are ~a more
                           joints to test."
                           (compute-joints-left)
    wait-for:   :correct-joint-response
                  (:timeout 15) joint direction
                                result
                :succeed (joint direction result)
    disable     :above
```

A `level-of-autonomy` query has been added to each context and is designed to allow RMS operators to declare which primitives should be run autonomously or in teleoperation mode. It also allows a semi-autonomous setting which generates an interactive query as each method is invoked to allow the user to vary the autonomy on the fly. The `level-of-autonomy` query and teleoperation methods were developed for all the primitives of the procedure tracking system. Yet, except for the general `tell-user` interface function, *RAPs above the primitive level remain unchanged.*

The key item to note is that we have *added no new skills* but have used the existing event skills to monitor the human's activity. The human is not told which joint to move or in what direction. In discussions with the crew it was determined that they want the RMS Assistant to be an non-intrusive monitor of humans as-

sumed to be experienced in the task. So in autonomous operations, the RAPs system determines the order of joint and direction testing, but in teleoperation mode, the `:correct-joint-response` skill serves to inform the RAP system of the order of the human activity. As long as progress is being made in the test, the RAP system will act in the same manner as in the autonomous mode.

A Summary of 3TPT Prototype Development

The 3T Procedure Tracking (3TPT) prototype development was started in mid-September 1996 and was ready for the STS-80 flight following by mid-November of the same year. A total of 35 RAPs and 75 skills were developed, and five staff months were expended in the effort. Two staff months were expended for the RAPs development, two staff months for the skills development and integration with the ISP architecture, and one staff month for knowledge engineering with a flight controller with 8 years' experience in RMS operations. This rapid development was made possible in part because the RMS simulation and the appropriate ISP services were already in place, but primarily because 3T is a mature robot control system for intelligent agents which had been used in several projects (Bonasso *et al.* 1997) and thus had a body of documentation (see URL http://tommy.jsc.nasa.gov/er/er6/mrl/projects/archs/) and example code available. This made it possible for all members of the RMS team – up to 12 other members are involved with other aspects of the RMS Assistant program – to become familiar with 3T and to support the information requirements necessary for the rapid development.

Results of Flight Following STS-80

The RMS is not used on every shuttle flight. Two flights were scheduled in our development time frame, STS-80 in November 96, and STS-82 in late February 97. Since it would be beneficial to have data available on 3TPT's performance with real time flight data for the February prototype, management decided to attempt to flight follow the portions of the RMS Checkout planned for February as well as various RMS joint movements during payload deployment and retrieval. In this test, obviously all operations would be in teleoperated mode.

STS-80 was a 17 day flight which involved the use of the RMS to deploy and retrieve the Wake Shield and the SPAS satellite. We employed 3TPT during all the daytime RMS operations. The flight following succeeded beyond our expectations, showing the utility of the system under circumstances which it was not explicitly designed to handle. The key distinction between flight following and the intended use for the RMS Assistant is that the crew would be actively using the RMS Assistant on board, but for the flight following, the crew was unaware of the 3TPT activity. Thus, there was no attempt on the part of the crew to tailor their actions based on the monitoring activity of 3TPT. Nevertheless, as discussed below, the reactive nature of the architecture allowed 3TPT to stay synchronized with the crew activity even in the face of the transient data losses normally experienced during a flight.

Getting started

We had at our disposal the flight plan for STS-80, so we knew when the RMS operations should be taking place. But these schedules were adjusted often throughout the flight. Since we were not in communications with the crew (3TPT was run out of the Robotics, Automation & Simulation Division building, not the MCC building), a data display client program was made available to us which allowed us to watch the settings of the switches which cued us that certain operations should be taking place. For example, the RMS Select switch, normally in the off position, would be switched to `port` prior to RMS operations. To monitor the RMS Power Up procedure, we would wait for that switch to change and then invoke the appropriate RAP. As the flight progressed, we automated this process using monitor RAPs, since there were several hour long delays in the flight activities.

RMS checkout operations

The five RMS Checkout procedures to be tracked were Reconfigure to Primary Power, Single Drive Test, Backup Drive Test, Direct Drive Test, and the Hand Controller Test. The system successfully tracked all five tests with the exception of the Hand Controller Test, though, as explained below, it did the best it could in that test, given the crew actions. Reconfigure to Primary Power is used to power up the RMS or to switch from Backup Power to Primary Power. It involves turning the power on, establishing I/O communications between the RMS and the GPC, and canceling the automatic safing system. Once the RMS select switch was turned to PORT, 3TPT straightforwardly tracked the reconfiguration by either querying the status of the switches or by watching for the events to occur, such as the I/O from the GPC being established.

When the RMS was at the pre-cradle position (see the discussion on monitoring arm movement below), we started the single-drive test. The tracking went well until we noticed that each joint was reported by

3TPT as being stalled. It turned out that the skills were receiving zero's for the tachometer readings because our MSID assignments were not synchronized with that being used by the MCC. Later in the flight, once we used the correct MSID assignments, all skills using the tachometers worked properly. About half way through the joint test, several data dropouts occurred and 3TPT thought the RMS had gone out of single mode. It automatically informed us of this and started waiting for the mode to change. When we started getting data again, 3TPT noted the return to mode and began tracking the crew activity, but by that time the crew had finished the test and had moved on to another procedure.

For the backup drive test, wherein the crew tests the joints under a backup power configuration, 3TPT successfully followed the reconfiguration into backup power. After that, because of the reduced telemetry during backup operations, the system waited until the RMS Select switch was moved to "off" signaling the start of reconfiguring to primary power. The system noted the switch change and again successfully followed the crew reconfiguring to primary power.

Once the RMS was at the direct drive test position, 3TPT successfully tracked this test to completion. The RMS is configured such that the brakes must be on in the direct drive mode, not off as is the case in single drive mode. The official procedure calls for the crew to test this when going into direct drive mode (leave the brakes off and see if a joint can be driven) and when switching to back to single mode (see if a joint can be driven with the brakes on). The crew did not perform these tests, so 3TPT timed out on both of them, but was able to resynchronize to the subsequent joint testing that was conducted.

The hand controller tests involved the crew moving the hand controllers in test mode (not actually commanding the RMS joints) to see if proper deflection signals were generated. During this test, the crew moved the hand controllers faster and held them for less time than was called for in the official procedure. As a result, 3TPT stopped with a constraint error after three axes were tested since the repeat-while clause being used detected no more progress when the crew had actually completed the procedure. Fortunately, we were able to retry the hand-controller test during STS-82. The crew on that flight conducted the test more in accordance with the official procedures and 3TPT successfully tracked it to completion.

Monitoring arm movement

Another cue for when an operation was taking place was that the RMS had to be moved to certain positions – the pre-cradle position, the direct drive test position and the hand controller test position – in single drive mode using a certain ordering of the joints. Though it was not a formal procedure, we wrote a RAP for doing this and executed it when we saw the RMS placed in single drive mode. 3TPT successfully tracked these movements for all three positions. This was possible even though the crew was not responding to the promptings of the system because for long joint moves, there was plenty of time to synchronize the RAP steps, and for shorter joint moves, often the RMS would be at the new joint position before the RAP commanded it. In the latter case the RAP simply computed the next joint to be moved to and pressed on. In one case, after commanding two of the required four joint moves, the RMS was already at the final position, so the RAP finished, announcing success.

The Wake Shield and the SPAS satellite were retrieved/deployed not in joint mode but with the hand controllers in a mode that moved the point of resolution (POR) of the RMS, be it the end effector or a point on the payload. In a few days' time we wrote a RAP with the appropriate skills which could monitor the progress of the POR to any designated position as well as watch for the correct joint response during such movements. In this manner, armed with the planned positions of the payloads, 3TPT successfully tracked the RMS as it deployed and retrieved the Wake Shield and retrieved the SPAS satellite. In one instance, 3TPT reported sluggish tachometers on two of the joints. The readouts on the data display system verified that the tachometer readings were indeed below the demand from the hand controllers on those two occasions.

Discussion

3TPT showed that it could follow crew operations successfully even in the face of loss of data and an "unresponsive" crew. After some reflection it was clear why 3TPT did so well despite the fact that the crew was unaware of its activity. 3T was designed not only for controlling autonomous robots but with the assumption that things will go wrong in the real-world. Such mechanisms as event timeouts, alternate methods, succeed, preconditions and constraint queries, and multiple retries are a direct result of that assumption. In this light, the flight following effort essentially required 3TPT to command (via speech only) and monitor an errant robot – i.e., a robot that was executing tasks either too fast or too slow while having trouble with its communications (something that has happened to us with several robots in other projects). In that regard, 3TPT performed as all 3T implementations: it accom-

plished whatever the real-world would allow, failing gracefully, and always keeping the user informed as to the events which caused its actions.

The failure of 3TPT to successfully track the hand controller test is an indication of placing the control of such activity at too high a level. The 3T philosophy dictates that activities with smaller time constants be situated lower down in the architecture. The crew's movement of the hand controller during the hand controller test, because they were not connected with the physical movement of the RMS, took place 5 to 10 times faster than during analogous joint motion. So we are considering redesigning the `test-all-joints` RAP so that the `correct-hc-response` event skill would not return to the RAPs level unless there was a problem with one of the hand controller axes. Otherwise it should simply monitor the joints and return when they were all tested. This in effect allows the faster running event skills to watch for the selection and testing of all axes rather than have RAPs intervene after each axis is tested.

A final note is in order with regard to knowledge engineering. We have found that knowledge engineering intelligent control, as opposed to intelligent analysis – like analyzing the stock market, seems to be made easier by the fact that one is constrained by the laws of the physical system. The physical system is designed to operate in a certain manner; any other manner will result in failure. Thus, once we had been tutored on how each major portion of the system functioned – the arm, the end-effector, and the switch panels – it was relatively easy to examine an official procedure and discern the rationale behind each step. In some cases, we identified redundant or superfluous steps and found that those steps were indeed being eliminated for the Orbiter Upgrade.

Conclusions

We believe this work and the early results as demonstrated during STS-80, have shown that 3T and related three-layer agent control architectures are emerging as a promising framework for the automation of any computer controlled machine in deployed applications. 3T's robustness, designed originally for autonomous robots in unstructured environments, is required in any real world environment involving humans and computer controlled machines. Its ability to smoothly allow human intervention and control for all or any part of the task makes it especially suited to applications where there is gradual movement toward less human-in-the-loop operations or toward full autonomy. In the RMS Assistant application, 3T is allowing the capture and use of both procedural (RAPs) and task execution/monitoring knowledge (skills) in different forms (task-nets or C functions) that allow for that knowledge to be applied in the most appropriate way, i.e., as state-based or continuous activity. Using the framework of 3T, NASA will realize its RMS Assistant automation goals in a cost effective manner and in a time frame which should see it in operation before the end of the century.

References

Bonasso, R. P.; Kortenkamp, D.; Miller, D. P.; and Slack, M. 1995. Experiences with an architecture for intelligent, reactive agents. In *Proceedings 1995 IJCAI Workshop on Agent Theories, Architectures, and Languages.*

Bonasso, R. P.; Firby, R. J.; Gat, E.; Kortenkamp, D.; Miller, D.; and Slack, M. 1997. Experiences with an architecture for intelligent, reactive agents. *Journal of Experimental and Theoretical Artificial Intelligence* 9(2).

Brooks, R. A. 1991. Intelligence without representation. *Artificial Intelligence* 47.

Connell, J. H. 1992. SSS: A hybrid architecture applied to robot navigation. In *Proceedings IEEE International Conference on Robotics and Automation.*

Elsaesser, C., and MacMillan, R. 1991. Representation and algorithms for multiagent adversarial planning. Technical Report MTR-91W000207, The MITRE Corporation.

Firby, R. J. 1987. An investigation into reactive planning in complex domains. In *Proceedings of the National Conference on Artificial Intelligence (AAAI).*

Firby, R. J. 1995. The RAPS Language Manual. Animate Agent Project Working Note AAP-6, University of Chicago.

Gat, E. 1992. Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In *Proceedings of the National Conference on Artificial Intelligence (AAAI).*

Gat, E. 1997. On three-layer architectures. In Kortenkamp, D.; Bonasso, R. P.; and Murphy, R., eds., *Artificial Intelligence and Mobile Robots.* Cambridge, MA: AAAI/MIT Press.

Yu, S. T.; Slack, M. G.; and Miller, D. P. 1994. A streamlined software environment for situated skills. In *Proceedings of the AIAA/NASA Conference on Intelligent Robots in Field, Factory, Service, and Space (CIRFFSS '94).*