A Procedure Representation Language for Human Spaceflight Operations

David Kortenkamp, R. Peter Bonasso, Debra Schreckenghost
TRACLabs Inc., 1012 Hercules, Houston TX 77058
korten@traclabs.com, r.p.bonasso@nasa.gov, ghost@ieee.org

K. Michael Dalal
Perot Systems, NASA Ames Research Center, Moffett Field CA
mdalal@arc.nasa.gov

Vandi Verma
Jet Propulsion Laboratory, Pasadena CA
vandi@jpl.nasa.gov

Lui Wang
NASA JSC ER2 Houston TX 77058
lui.wang-1@nasa.gov

Abstract

Procedures are at the heart of all crewed spacecraft operations. They are the accepted means to command systems. Current procedure representations do not lend themselves to automation. A new procedure representation language (PRL) is being proposed to NASA's Constellation program to allow for procedures to be automated whenever possible. PRL is similar to many other procedural languages that have been developed over the past decade such as RAPS and PLEXIL. Its main advantage is in representing information that humans will need to perform a procedure in addition to information that the autonomous system will need. This allows for implementation of adjustable autonomy in which either a human or a computer can execute different parts of a procedure. Examples from International Space Station and robotic procedures are given in this paper.

1. Introduction

Procedures are the accepted means of commanding spacecraft. Procedures encode the operational knowledge of a system as derived from system experts, testing, training and experience. NASA has tens of thousands of procedures for Space Shuttle and the International Space Station (ISS), which are used daily by both flight controllers and crew. As an example, there is a procedure on ISS to perform a smoke alarm test. For ISS operations, procedures are represented in an XML format that is presentation-based. That is, the format describes how the procedure should look when displayed. ISS procedures are then executed manually using system displays. For Constellation programs, NASA is pursuing an operations concept in which procedures can begin with manual execution then, over time, shift to more automated execution as situations and experiences warrant. To support this new operations concept we are developing a new representation for procedures called a Procedure Representation Language (PRL). PRL includes support for automating procedure execution without taking away any of the current information for human operators contained in the existing ISS procedures. Thus, it allows for a gradual and seamless increase of automation.

2. Motivation

PRL is one of the first execution languages designed for both humans and automated systems. Almost all previous execution languages are built for fully automated systems in which humans have well-defined roles that overlap very little with the role of automation. In the systems we are designing there is a significant overlap between the tasks that can be performed by a human and those that can be executed

by a computer. In most cases, tasks can be done by either. At first a task will be performed almost entirely by humans; gradually, portions of it will be automated with the human always kept informed and able to intervene. This is called adjustable autonomy [5] and allows for gradual introduction of autonomy. This allows for a fallback to manual operations in specific situations and allows the level of automation to be set based on what is appropriate for each situation.

3. Structure of PRL

PRL is currently defined as an XML schema that defines a variety of tags that can be used to describe a procedure. At the highest level is a *procedure* tag that marks the beginning of a new procedure. Each procedure consists of *steps* that describe smaller tasks within the procedure. Steps themselves have *blocks* that are containers for *instructions* that provide explicit detail about commanding a system. Each of these components can have *automation data* that controls their execution status. In this section we describe the main components of PRL.

3.1. Procedures

A procedure is the top-level entity in PRL. Each procedure has a human-entered name and number. Each procedure also has a unique identifier. A procedure can contain a block of "meta-data" with information about the procedure such as the author, comments, revisions, etc. Each procedure can contain parameters that are passed into the procedure at execution time. A procedure can also contain local variables that can be used within the procedure. All procedures can have Automation Data (see Section 3.5) that controls when and how they are executed. A procedure has as its body one or more steps.

3.2. Steps

A step has a specific purpose or goal within the procedure. Each step has a human-entered name, a number that is generated sequentially and a unique identifier. Each step has an optional information statement, which is human-readable text that can provide additional information to a human performing the step. Each step must end in one of three ways: 1) A conditional branch in which Boolean expressions are paired with step identifiers and execution branches to the first step whose Boolean expression evaluates to TRUE; 2) A goto-step in which execution continues at the step identified in the goto-step; and 3) An exit procedure in which execution ends. Each step can have Automation Data controlling its execution (see Section 3.5). Each step has as its body one or more blocks.

3.3. Blocks

Blocks are wrappers that contain the instructions necessary to accomplish the step. The most basic block is an Ordered Block, which contains one or more instructions that are executed one after the other. An Unordered Block contains one or more instructions that can be executed in any order. Other block constructs offer control over execution flow. These are: 1) If Then block in which a Boolean expression is matched to a block. If the Boolean expression evaluates to TRUE the block is executed. An optional Else block is executed if the Boolean expression evaluates to FALSE; 2) For Each block in which a list and a block are given and the block is executed for each item in the list; and 3) While block in which a Boolean expression is matched with a block and the block executes while the Boolean expression evaluates to TRUE. Each block contains another block or a group of instructions.

3.4. Instructions

Instructions are the atomic actions of PRL. There are a wide variety of instructions, which will be detailed in this section. A Command Instruction issues a computer command (possibly with parameters) to the underlying system. A Verify Instruction compares a specific telemetry item to a target value. If the comparison is TRUE the instruction succeeds and execution continues. If the comparison is FALSE then execution halts and the procedure fails. An Ensure Instruction is similar to a Verify Instruction except that if the comparison fails a command is given to the underlying system. A Wait Instruction halts execution either for a specified period of time or until a Boolean expression becomes TRUE. The Call Procedure Instruction calls another procedure using the procedure identifier and passes any required parameters. The Call Procedure Instruction can be blocking, meaning that the calling procedure halts execution until the called procedure is finished or non-blocking, in which case both procedures execute simultaneously. The Call Function Instruction calls a user-defined function running on the underlying system. A Manual Instruction is used for commands that need to be performed by a human, that is, they require manual contact. This instruction simply contains text that is displayed to the user. An Input Instruction acquires data from a user (or other source) and assigns it to a local variable in the procedure. Each instruction can also have Automation Data that controls its execution.

3.5. Automation Data

Automation Data is used to control execution in PRL. Automation Data includes a pair of gating conditions called PreConditions and PostConditions. These are

Boolean expressions that must evaluate to TRUE before execution of the procedure, step or instruction begins and after execution ends or the procedure execution fails. Automation Data also includes a pair of wait conditions called StartConditions and EndConditions. These are also Boolean expressions that must evaluate to TRUE before execution can begin and end. If these conditions evaluate to FALSE then execution waits until they become TRUE. Automation Data also includes InvariantConditions that are Boolean expressions that must remain TRUE throughout execution of the procedure, step or instruction or execution fails. Automation Data also includes a description of the resources necessary to execute the procedure, step or instruction. This can be used by a planner to resolve conflicts in concurrent execution. Automation Data can apply at the procedure, step or instruction level.

4. PRL and other execution languages

PRL represents operations for both humans and automation. It was developed after extensively studying existing execution languages that were developed in automated robotics research, and also procedures currently used in human space flight operations. In particular, many PRL constructs come from execution languages such as PLEXIL or RAPS. We have also implemented translators from PRL to PLEXIL and RAPS. In this section we compare PRL with two other popular execution languages.

4.1. PLEXIL

The Plan Execution Interchange Language (PLEXIL) [1] is a language for representing plans for automation. It has been used to operate rovers in a simulation of a lunar robotic site survey, and has been used to demonstrate automation for International Space Station (ISS) operations. Like PRL, PLEXIL's core form is represented in XML, and XML-based technologies are used extensively in its specification, implementation, and verification.

PLEXIL's structure is simpler than that of PRL. In PLEXIL a plan is represented as a tree of uniform structures called nodes, with only one means of composition: internal *list* nodes simply aggregate their children. In contrast, PRL has a heterogeneous mix of structures, which includes a variety of composable *blocks* and a large *instruction* set, with a specific nesting order. Yet both languages represent a plan as a hierarchical decomposition of tasks: high level tasks are closer to the root node (or procedure level), while leaf nodes (or instructions) are primitive actions such as assigning to a local variable or sending a command to hardware.

A major difference between PRL and PLEXIL is found in their execution logic. The execution of PLEXIL is entirely condition driven. Each node contains a set of conditions that enable and govern the node's execution and outcome. These condition-driven semantics, which allow a natural representation of concurrency and event-driven monitoring, were adopted by PRL, but only as a secondary control mechanism. The primary means of control in PRL is conditional branching at the step level. Each step specifies either the next step(s) to which control may transfer, or a mode in which to immediately exit the procedure.

The simplicity of PLEXIL's structure and control logic extends naturally to both its execution semantics and implementation. The execution of PLEXIL is defined by an elegant formal framework, which also proves useful properties of the language [2]. The essence of the PLEXIL executive (plan interpreter) is implemented in several hundreds of lines of C++ code. At the same time, PLEXIL lacks PRL's human-centric expressiveness of plans, which includes an application domain oriented ontology, features for interacting with visual displays, and support for adjustable autonomy. Unlike PRL, PLEXIL is not a practical user programming language for substantial procedures or plans, and is more suitable as a target language for planners or translators.

With much success, PRL procedures have been automatically translated into PLEXIL plans and executed by the PLEXIL executive, thus making the two languages highly complementary. The translator, which is written in the XML transformation language XSLT, performs at its highest level the straightforward mapping of PRL constructs into PLEXIL nodes shown in Figure 1. The complexity of translation resides mainly in two areas, the translation of PRL's control flow, and the "instrumentation" of the PLEXIL plan needed to support adjustable automation.

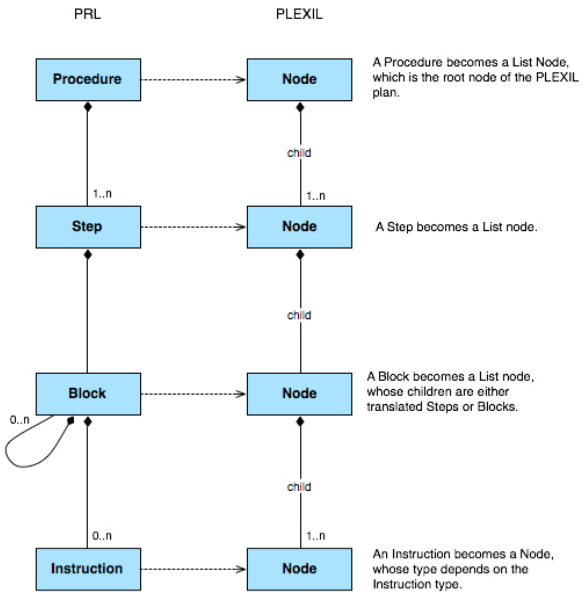


Figure 1: PRL to PLEXIL translation

PRL's branching (which occurs between steps only) is translated into a node enabling and disabling mechanism wherein a node representing a given step has a start condition formed in part from a composition of the branching conditions in all steps that branch to the given step. Furthermore the node's execution is disabled at all but these occasions to fully heed the procedure's specified control flow. Figure 2 illustrates the resulting PLEXIL node structure.

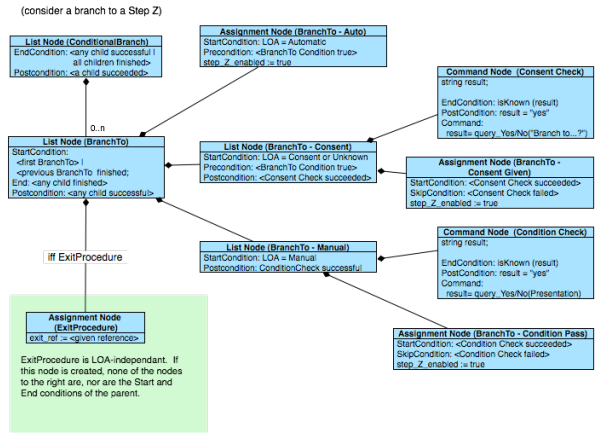


Figure 2: PRL branching in PLEXIL

Adjustable autonomy is achieved by executing procedure elements with respect to their assigned *level of autonomy*, or LOA. The translation from PRL to PLEXIL adds constructs to acquire LOA in a manner similar to any other operation performed by PLEXIL. In particular, the PLEXIL plan sends queries to an LOA server for a given node and executes the node accordingly. For example, when executing a node X representing a given step, if the LOA for this step is *consent*, another node that issues a command querying the user for consent is executed, and a confirmation enables the execution of X. Figure 3 illustrates the resulting PLEXIL node structure.

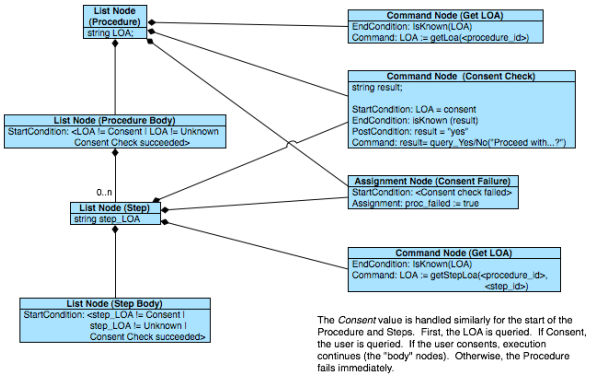


Figure 3: Adjustable autonomy in PLEXIL

4.2. RAPS and APEX

The Reactive Action Package system (RAPs) [3] and Apex [4] were designed to execute predefined AI plans reactively. RAPs and Apex are similar enough for the following discussion that we will use the RAPs system as the exemplar. Reactive AI plans are decomposable into executable code, have expected world states to be achieved, preconditions and post-conditions for each task, and detect when the plan is failing and invoke contingency plans. A procedure is a simple plan, that is, a set of steps executed in a defined order to accomplish a goal. As such, PRL can represent such plans and a RAPs (or Apex) exec can execute them. Indeed, we have developed a PRL to RAPs translator and have translated and executed a number of PRL procedures based on actual NASA procedures.

The one drawback of PRL in this regard is that the semantics of procedure execution require that all steps be tied to the same parent procedure. In RAPs, any group of instructions is a procedure (a RAP). So at a choice point in RAPs, each branch invokes another RAP and the parent RAP is finished. In order to keep the parent-child continuity that PRL requires, the top level RAP resulting from a PRL translation is a large execution tree of all the paths that could result from all of the choice points. This requires that the RAP

interpreter maintain this tree until the final path through it has been realized.

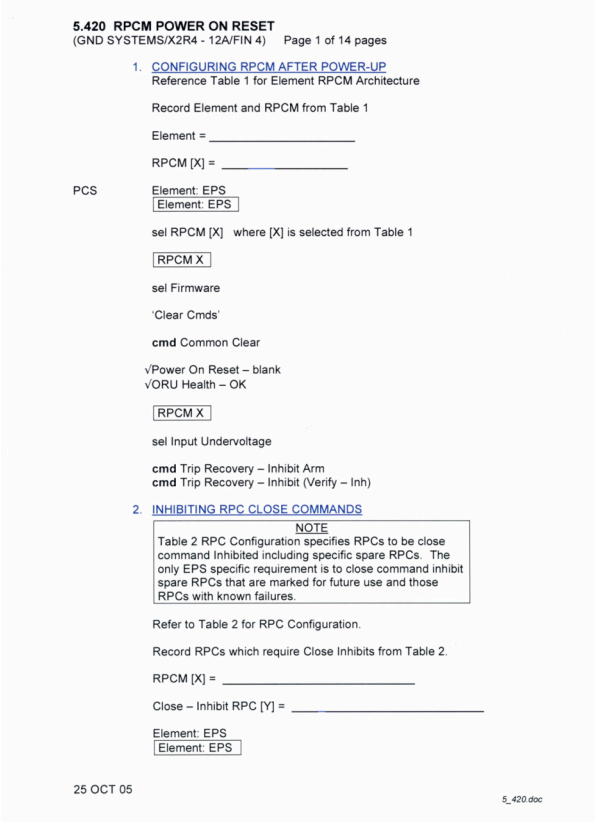
Beyond this problem, there are two aspects of executing AI plans reactively that stretch the capabilities of the current PRL language to be executed by a RAPs exec. Both have to do with the fact that beyond executing linear plans, RAPs is reactive, that is, the order of execution of any instruction is dependent at any moment on the dynamic state of the world.

The first aspect of reactivity that is difficult to achieve in PRL is the ability to handle concurrent execution. AI plans are usually only partially ordered and the final execution order of their steps is situation-dependent. The current PRL makes unordered execution difficult to manage. This is especially apparent with regard to failure. RAPs has the ability to detect failures and to invoke alternative plans based on the type of failure. For example, if a step fails by timing out before achieving its goal, RAPs may simply re-invoke the procedure. RAPs can also encode more than one way (method) to achieve a result. If the first method fails, the RAPs interpreter will invoke an alternative depending on the cause of the failure. There is little support in the current PRL for this kind of reactive control.

The other aspect of reactivity involves the way one develops plans to be executed by RAPs. One usually starts by defining a set of primitives that command objects in the physical world and obtain telemetry from them, for example, a robot_move to a set of coordinates with a data monitor that watches the robot's position. From these we can develop a higher-level procedure for the robot to visit a series of waypoints. Assuming the robot has an arm with an end-effector, one might also have primitives to move the arm to a pose and to track the arm's movement as well as to grasp and ungrasp objects. From these one could develop a pick-and-place procedure. Combining the two higher-level procedures, one could then build a top-level procedure that goes to various locations to collect soil samples.

But each sub-procedure with which the top-level procedure is composed is reactive. If the pick-and-place sub-procedure fails, the top-level procedure can try again, skip the object and move to the next waypoint or fail altogether. While it is possible to construct procedures from other procedures in PRL, it does not provide the degree of control over the activation of sub-procedures that is provided by RAPS.

Finally, both RAPs and Apex support the concept of monitors, that is, procedures which “wakeup” periodically, check the world for certain states -- e.g., a fire monitor -- and go back to “sleep”. These procedures never "finish" in the traditional sense of procedure completion. PRL has as of yet no constructs for easily defining such monitors.



5.420 RPCM POWER ON RESET
(GND SYSTEMS/X2R4 - 12A/FIN 4) Page 1 of 14 pages

1. CONFIGURING RPCM AFTER POWER-UP
   Reference Table 1 for Element RPCM Architecture

   Record Element and RPCM from Table 1

   Element = _____

   RPCM [X] = _____

PCS Element: EPS
Element: EPS

sel RPCM [X] where [X] is selected from Table 1

RPCM X

sel Firmware

'Clear Cmds'

cmd Common Clear

√Power On Reset – blank
√ORU Health – OK

RPCM X

sel Input Undervoltage

cmd Trip Recovery – Inhibit Arm
cmd Trip Recovery – Inhibit (Verify – Inh)

2. INHIBITING RPC CLOSE COMMANDS

   NOTE
   Table 2 RPC Configuration specifies RPCs to be close command Inhibited including specific spare RPCs. The only EPS specific requirement is to close command inhibit spare RPCs that are marked for future use and those RPCs with known failures.

   Refer to Table 2 for RPC Configuration.

   Record RPCs which require Close Inhibits from Table 2.

   RPCM [X] = _____

   Close – Inhibit RPC [Y] = _____

   Element: EPS
   Element: EPS

25 OCT 05 5_420.doc

Figure 4: An ISS EPS Procedure

5. PRL in action

We have used PRL in two domains at NASA Johnson Space Center. The first domain is procedures to control the International Space Station. The second domain is procedures in the cockpit of a lunar surface robot.

5.1. ISS Application

The ISS has thousands of procedures that are used for control and troubleshooting. Figure 4 shows the firswt page of an actual ISS procedure for configuring an electrical power system component. This particular procedure takes in a single parameter (X) that is the specific component that is to be configured. Commands (**cmd**) are issued on the component. Verifies (√) are performed on telemetry coming from the component. The second step issues a series of commands to the component in a loop (not shown in the figure). On ISS this procedure is executed manually by crew members who issue commands through a command and control interface (the various

'sel' instructions in the procedure are navigations through the interface).

We have re-authored this particular ISS procedure into PRL. This requires embedding the actual computer commands and telemetry in the procedure so that it can potentially be executed autonomously. Here is PRL pseudo-code for the first step of the procedure:

```
Procedure
  Parameters
    id="X" externalType="RPCM"
          parameterType="In"
  LocalVariables
    id="Y" externalType="RPC"
  ExitModes
    id="exit_success" Message="Procedure exited
          successfully"
    id="exit_verify_failed" Message="Procedure failed on
           verify"
  ProcedureTitle
    name="RPCM Power On Reset" number="5.420"
          id="proc_5420"

  Step id="step_1" title="Configuring RPCM after powerup"
    OrderedBlock id="block_1"
      CommandInstruction id="instr_1"
        Description "cmd [X] Common Clear"
        CommandIdentifier X.RPCMCommonClear
      VerifyInstruction id="instr_2"
        ExitModeReference="exit_verify_fail"
        Description "Verify [X] Power On Reset -- blank"
        Value X.PowerOnReset
        Operator "equal"
        TargetValue "blank"
    GotStep stepRef="step_2"
```

This PRL file contains a parameter passed into the procedure of type RPCM and a local variable of type RPC. An RPCM is a power module on ISS and an RPC is a switch in that power module. There are dozens of RPCMs on ISS that can all be configured with this same procedure. The parameter, X, specifies which RPCM is being configured in this particular instance. Exit modes provide messages out of the procedure when it exits. In this case there are two, one for a successful exit and one if a verify fails. The pseudo-code shows one step of the procedure. The stem has one command instruction and one verify instruction. The command instruction has an identifier that needs to be created at run-time since the command needs to go to the correct RPCM. The verify instruction compares a telemetry value with a target value using an operator.

As an experiment, the full PRL file representing ISS procedure 5.420 was translated into PLEXIL and executed using the PLEXIL Executive (see Section 4.2) against a simulation of the ISS Electrical Power System (EPS). A procedure display (see Figure 5) and associated tools provided execution context to the end user.

5.2. Robotic Application

PRL has also been used to represent procedures running on a cockpit to command autonomous robots at a distance. In these cases the PRL specified various start-up and shut-down procedures and served to orchestrate robot activities. Low-level, closed-loop control of the robot was on-board and activated by the procedures. Here is an example of a robotic PRL in pseudo-code:

```
Procedure
  Parameters
    id="X" dataType="real" parameterType="In"
    id="Y" dataType="real" parameterType="In"
   id="A" dataType="real" parameterType="In"
  ExitMode
    id="exit_succeed" Message="Succeeded"
    id="exit_failed" Message="Failed"
    id="exit_cancelled" Message="Cancelled"

  ProcedureTitle
    name=" DRIVE TO XYA" number="100"
          id="proc_100"

  Step id="step_1" title="Send a command to the robot to
                          move to the specified location"
    OrderedBlock id="block_1"
          CommandInstruction id="instr_1"
            AutomationData
                  EndConditions timeout-"20 seconds"
                     CommandQueueStatus = 2 OR
                     CommandQueueStatus = 5 OR
                     CommandQueueStatus = 6 OR
                     CommandQueueStatus = 7
                  PostConditions
                     exitModeReference="exit_failed"
                     CommandQueueStatus = 6
            Description "Send CCmd::DriveToXYA() then
                              wait for command to succeed"
            CommandIdentifier DriveToXYA
                                  Parameters X, Y, A

    ExitProcedure exitModeReference="exit_succeeded"
```

This procedure has three parameters – the (X,Y) location for the robot to drive to and its final orientation (A). There is one step with one ordered block that has a single instruction. The instruction has Automation Data. Specifically, there is an end condition that says that this instruction is not finished until the value of CommandQueueStatus is either 2, 5, 6, or 7. This means that the execution status of this instruction is not complete until this end condition is met or until the timeout of 20 seconds passes, after

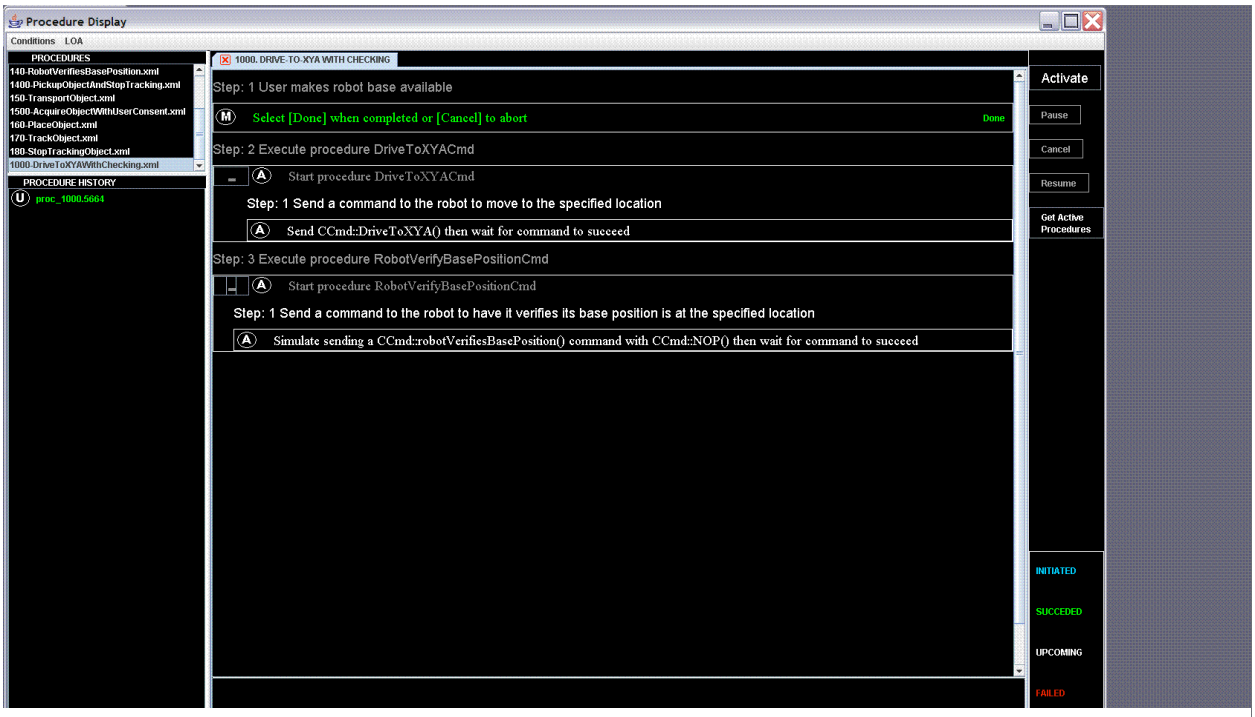


Figure 5: The Procedure Display for PRL

which the instruction fails. There is also a post condition that the CommandQueueStatus be equal to 6 or the instruction (and procedure) fails with exitMode failed. The single command instruction calls a system command (DriveToXYA) with the given parameters. After Step 1 is finished the procedure exits with exitMode succeeded.

This (and other) procedures were executed on the cockpit for the Centaur robot at NASA Johnson Space Center. The cockpit user saw a procedure display as shown in Figure 5. The display could be used to select procedures for execution and to monitor their status. The executive dispatched commands to the robot over a network connection and monitored for telemetry.

6. Conclusions and future work

PRL is in the early stages of being investigated by the Constellation program of use in future missions. The feedback has been positive so far. One improvement we are considering is creating a more modular language. Currently there is one schema that contains all of the PRL constructs. For some simple applications (e.g., early Orion operations) not all PRL constructs will be necessary. Also, robotic operations often require different constructs than single vehicle operations. Our users want the ability to load in only the PRL constructs they need for their specific application. We are examining how we can create a core PRL that is then extended to accommodate different operational scenarios.

We are also exploring adding a greater ability to deal with concurrency in PRL. Since many current procedures are performed by a single human concurrency is not a real requirement. However, has missions become more complex and automated having multiple procedures being executed at the same time will be more likely. In these cases we need to address resource contention and timing constraints between procedures.

PRL is only as useful as the tools that are available for it. We are building a Procedure Integrated Development Environment (PRIDE) for authoring procedures in PRL, executives that execute PRL (or translations of PRL), displays that show a PRL file to a human and show its execution status and procedure support services that handle human interaction with the procedure during execution. PRL's advantage is that it provides a single representation can be used by all of these tools, thus streamlining the authoring, verification, validation and execution of procedures.

7. Acknowledgments

A large group of people of contributed to PRL including Robert Phillips (L3Com/NASA JSC), Michel Izygon (Tietronix/NASA JSC), Wes White (Tietronix/NASA JSC), Arthur Molin (SKA/NASA JSC), Tam Ngo (NASA JSC), Ari Jonsson (NASA ARC), Chuck Fry (Perot Systems/NASA ARC), Jeremy Frank (NASA ARC), Scott Bell (SKA/NASA JSC), Tod Milam (SKA/NASA JSC), Bebe Ly (NASA JSC), Ken McMurtry (Tietronix/NASA JSC), Kevin Kusy (SKA/NASA JSC), Cesar Munoz (NASA LaRC), Tony Barrett (JPL) and Chad Keeton (Tietronix/NASA JSC).

8. References

[1] V. Baskaran, M. Dalal, T. Estlin, C. Fry, M. Iatauro, R. Harris, A. Jonsson, C. Pasareanu, R. Simmons, V. Verma, "Plan Execution Interchance Language (PLEXIL) Version 1.0", NASA Technical Memorandum, Nov 2007.

[2] G. Dowek, C. Munoz, C. Pasareanu, "Formal Semantics of a Synchronous Plan Execution Language", *Workshop on Planning and Plan Execution for Real-World Systems: Principles and Practices for Planning in Execution at the International Conference on Automated Planning and Scheduling (ICAPS)*, 2007.

[3] Firby, R.J., "An Investigation into Reactive Planning in Complex Domains," in *Proceedings of the National Conference on Artificial Intelligence (AAAI),* 1987.

[4] Freed, M.J, "Managing Multiple Tasks in Complex, Dynamic Environments," in *Proceedings of the National Conference on Artificial Intelligence (AAAI),* 1998.

[5] Schreckenghost, D., R. P. Bonasso, D. Kortenkamp, S. Bell, T. Milam, C. Thronesbery, "Adjustable Autonomy with NASA Procedures," in *Proceedings International Symposium on Artificial Intelligence, Robotics and Automation for Space (i-SAIRAS)*, 2008.