Developing and Executing Goal-Based, Adjustably Autonomous Procedures

David Kortenkamp, R. Peter Bonasso and Debra Schreckenghost
NASA Johnson Space Center/TRACLabs Inc.
Houston Texas 77058
kortenkamp@jsc.nasa.gov

This paper describes an approach to representing, authoring and executing procedures during human spaceflight missions. The approach allows for the explicit incorporation of goals into procedures. The approach also allows for adjustably autonomous execution of procedures. That is, the procedure can be executed by a computer, by a human or in any combination of human and computer. A new procedure representation is described as are tools for authoring procedures in the new representation. An execution engine interprets the representation and executes it appropriately. An end-user procedure display provides guidance to the human as to execution status. Experiments were performed using actual International Space Station procedures being executed against a high-fidelity space station simulation. The new approach increases the efficiency of crew and ground controllers while executing procedures and reduces errors in procedure execution.

I. Introduction

Procedures are the accepted means of commanding spacecraft – both crewed and uncrewed. 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) and they are used by both flight controllers and crew. Procedures are inherently goal-based, that is, they are intended to place the system in a particular mode, to diagnose a particular failure or to recover from a malfunction. However, procedures as they are encoded today rarely make explicit the goal they are achieving, instead relying on human operators to choose the correct procedure and then verify that it had the intended effect. Also, procedures today are executed manually by humans reading the procedures and then interacting with the system through command and control interfaces. The work described in this paper addressed these two issues by defining an explicit representation of procedures and their goals and by using that representation to allow procedures to be flexibly executed by either humans or computers. We also address the issue of developing and verifying goal-based procedures.

I.A. Motivations

Procedures can be very long and complex. Many run to several and even tens of pages. This creates several problems. First, procedure execution is very time-consuming. Second, manual execution of procedures can be error-prone. Finally, verifying that a procedure is correct is expensive and human-intensive. An explicit representation of procedure goals and actions can help in each of these cases. Automation can be used to reduce the time necessary to execute routine procedures. Additional software can monitor manual procedure execution and warn of deviations. Verification of procedures can be done autonomously against simulations to determine if the procedure achieves the goal. Because automation can introduce its own risks, we have developed an adjustably autonomous approach. This provides a comfort-level to the operator that they can automate only those portions of a procedure with which they feel comfortable. The level of automation can change depending up on the context in which the procedure is being executed. This provides a path towards increased automation that is appealing to human operators.

I.B. Goals in human spaceflight

Goals in human spaceflight, as in uncrewed missions, involve putting systems into particular states. The difference is that humans on-board the vehicle can both assist in achieving system goals and can be given goals themselves (e.g., to exercise). Having humans on-board means that many goals can be implicit. For example, if a human sees a fire they will put it out without needing to be explicitly assigned a goal. Or, if a procedure says to verify that the temperature is 100 there is an implicit goal that if the temperature is not 100 make it so. These implicit goals are products of both training and common sense. Human training and significant sensory skills allow for fairly expansive goal definition – e.g., "land the vehicle where it is not too bumpy" can be a goal given to a pilot who relies on training and visual sensing to determine the correct action whereas an automated pilot will need more precise definitions of "bumpy" and appropriate sensing modalities.

II. Representing procedures

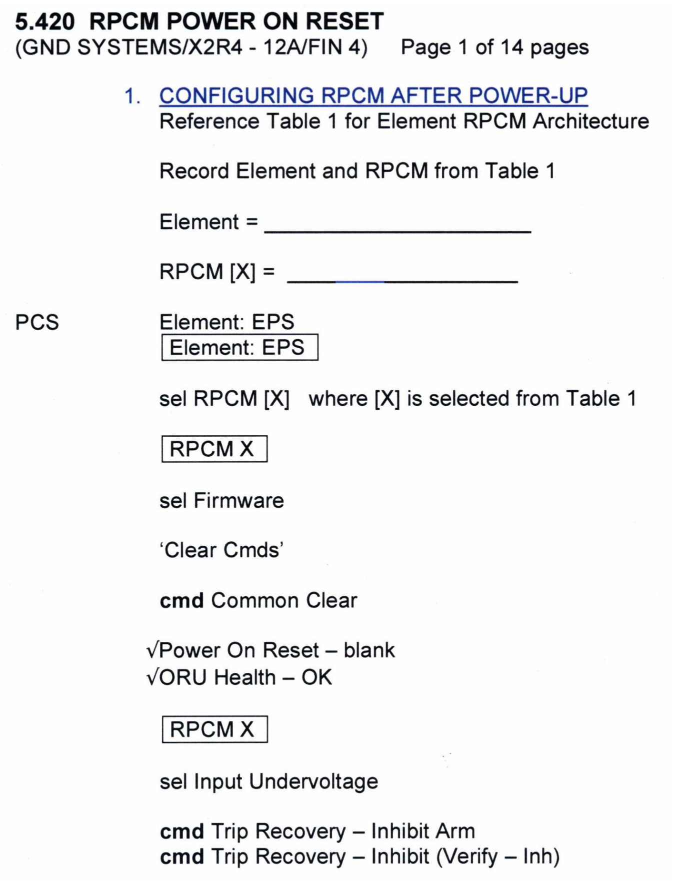
Procedures are currently represented in natural language on a human-readable display. They are intended for human consumption not computer consumption. Goal information is often vague if it is there at all. A small number of ISS procedures contain an "Objective" statement at the beginning. For example one procedure has the objective: "The purpose of this procedure is to enable automated water venting." In addition, telemetry and commanding information is not encoded in the procedure. Also, procedures tend to be lengthy and very specific to hardware configurations. The current representation is not sufficient for goal-based operations nor adjustably autonomous execution.

We have been developing a new procedure representation called the Procedure Representation Language (PRL). This language keeps the user friendly display format of current procedures but augments it with content-based information (e.g., goals, resources, pre-conditions, post-conditions, etc.) required for more autonomous execution. PRL also allows for procedures to be written in a more modular fashion, with larger procedures composed of small procedure "fragments" that accomplish specific tasks. In this way, new procedures can be easily created and verified as hardware configurations change. Our procedure representation language is an integration of an automated execution language developed at NASA called PLEXIL[1] and the existing ISS procedure representation. Both of these representations and our new PRL written as eXtensible Markup Language (XML) schemas.

II.A. Existing procedures

Figure 1 shows the first step in a two step International Space Station (ISS) procedure that configures a Remote Power Controller Module (RPCM) after it is powered up. This figure shows exactly what the operator sees when executing the procedure. The procedure is executed using a command and control interface. This procedure is for a generic RPCM, so the first part of the procedure records where the RPCM is located (Element) and on which RPCM the procedure is being done. Then that specific RPCM is selected from a graphical display of the station Electrical Power System (EPS). Then a "Firmware" button is selected on that display. The 'Clear Cmds' area has to be located and within that area there is a command button called "Common Clear", which is to be pressed. Then the operator verifies that a telemetry field called "Power On Reset" is "blank" and that another telemetry field called "ORU Health" reads "OK". Two other commands are also issued in this step of the procedure. Note that an RPCM is similar to a circuit breaker box. It has multiple Remote Power Controllers (RPCs) that power downstream loads. Like circuit breakers these RPCs are designed to "trip" open if the load is too large (or too small). Unlike circuit breakers in homes these RPCs can be electronically commanded open or closed to effect power flow.

Figure 2 shows the second step in a two step procedure that configures an RPCM after it is powered up. This step requires the operator too look in a table (attached to the procedure) to determine which RPCs need close inhibit commands. Close inhibit commands prevent an RPC from being accidentally closed – not all RPCs will need to be close inhibited. After the operator creates a list of those RPCs that need to be close inhibit they repeat the "Close Cmd – Inhibit" command for each of those RPCs and verify the result.


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
$\boxed{\text{Element: EPS}}$

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

$\boxed{\text{RPCM X}}$

sel Firmware

‘Clear Cmds’

cmd Common Clear

√Power On Reset – blank
√ORU Health – OK

$\boxed{\text{RPCM X}}$

sel Input Undervoltage

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

Figure 1. First step of a power on reset procedure.

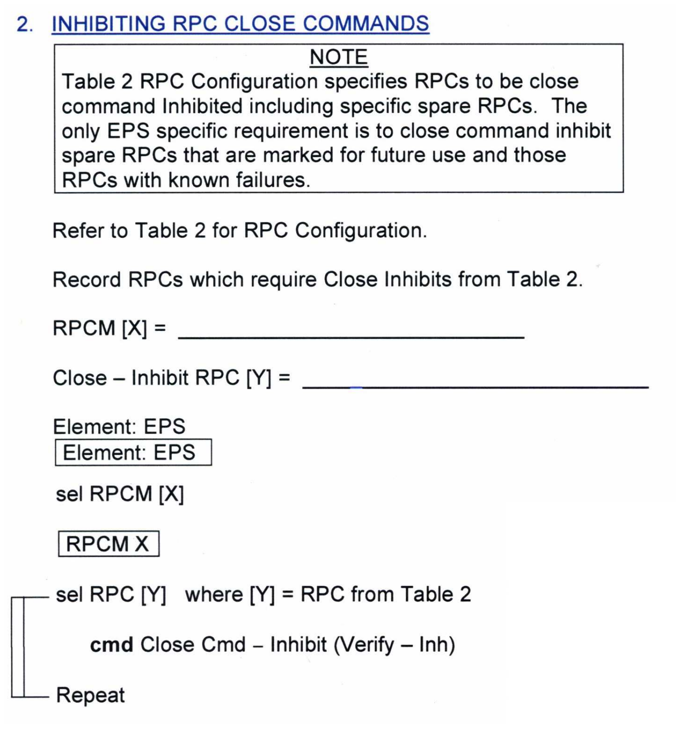
II.B. Adjustably autonomous procedure representation

We are taking procedures such as those described in the previous section and converting them into executable procedures that can be performed by human operators or autonomous agents (or both) by representing them in PRL. In this way we encode the procedure content. For example, procedure content includes the context (pre-conditions) that must be true for the procedure (or its steps) to be valid and to achieve its end condition (goal) and whether these pre-conditions need to hold only at the start of the procedure or during its entire execution. As another example, procedure content includes what the procedure accomplishes (success conditions or other post-conditions). Content can also include the resources required by the procedure, the time necessary to complete it or the skills the crew member needs to perform it.

II.B.1. Structure of a procedure

The basic structure of a procedure as represented in PRL is:

- Meta data: Includes a unique identifier, the procedure name, the procedure author, date, etc.
- Automation data: Includes the following:
  - Start conditions: a boolean expression that when evaluated to false means that the procedure should wait until the boolean expression is true before starting
  - Pre-conditions: evaluated after the start condition and if false then exit the procedure immediately with failure
  - Post-conditions: evaluated after a procedure is done and if it evaluates to false then the procedure has failed



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

sel RPCM [X]

RPCM X

sel RPC [Y] where [Y] = RPC from Table 2

cmd Close Cmd – Inhibit (Verify – Inh)

Repeat

Figure 2. Second step of a power on reset procedure.

  - End conditions: evaluated continuously and when it is true execution of the procedure is finished
  - Invariant conditions: must remain true during the entire execution of the procedure; otherwise the procedure fails
  - Resources: any resources (time, fuel, crew members, power, tools, etc.) required for execution of this procedure

- Parameters: Declarations of any data that is passed to the procedure from whatever is calling the procedure
- Local Variables: Declarations of any variables used internal to the procedure
- ExitModes: Definition of explicit procedure exit modes, specifying procedure success or failure, and giving optional description of the reason for exiting
- ProcTitle: Procedure number and title
- InfoStatement: Specifies explanatory information (e.g. notes, cautions, warnings) that might be needed or desired by a human executor
- Step: A step is the basic organizing structure of a procedure. A procedure can contain one or more steps and each step consists of the following parts:
  - Automation data: As above except replace the word "procedure" with "step"
  - Step title and unique identifier
  - Information to be displayed to the user before this step is executed in manual operations
  - A block that can be ordered (i.e., executed in sequence) or unordered (i.e., executed in any order). A block can also consist of an If-Then statement, a For-Each statement or a While statement. Inside of a block are:
    - ∗ Automation data, as above except for blocks

    * Another block allowing for arbitrary nesting of blocks in a step
    * Instructions also have automation date and include the following:
        - Command instruction, which sends an electronic command to the system being controlled
        - Verify instruction, which checks the value of a telemetry against a constant and fails the procedure if they do not match. Typically commands and verifies are paired such that the verify instruction determines if the command was successful.
        - Ensure instruction, which checks to value of a telemetry variable against a target and, if the value is not correct then issues a command that should make it correct
        - Input instruction, which assigns external data (from a crew member, telemetry, etc.) to a local variable
        - Manual instruction, which asks a human to issue a command
        - Physical device instruction, which asks a crew member to physically manipulate a device
        - Wait instruction, which waits for either a set period of time or until a boolean expression evaluates to true, whichever comes first
        - Procedure call, which can be blocking (i.e., the current procedure pauses until the called procedure finishes) or non-blocking (i.e., the two procedures continue to run in parallel)
        - Stop, pause and resume procedure, which effects the execution of the current procedure
  - Conditional branch, which contains a set of boolean expressions paired with a goto step or exit procedure command that is executed if the boolean expression is true. This defaults to go to the next step if no conditional branch is given.

They key components of a procedure are the steps, blocks, instructions and branches. Each step represents a set of logically connected blocks and instructions. There is no set minimum or maximum number of blocks or instructions per step. It is assumed that the author will create steps based on the needs and necessary structure for the procedure. Branching can only occur between steps not within steps. That is, you cannot branch from an instruction in one step to an instruction in another step. Also, there is no way to denote parallel steps – parallelism is only allowed within blocks. Blocks group sets of instructions stating whether they are done in strict order or can be done in any order. Blocks can also allow for repeating instructions or conditionally executing instructions. Branches tell the procedure executive where to go after finishing a step. Conditional branches can go many different places depending upon the evaluation of boolean expressions.

III. Representing systems

Procedures describe the processes by which a device or system is operated or debugged. They are oriented towards achieving some task or goal. They do not describe the device or system. However, a representation of the system is necessary for procedure execution. That is, a representation of all of the possible commands and telemetry, and taxonomy of the device or system is required to support procedure authoring and execution. This representation is different from the procedure representation described in the previous section.

The representation of commands and telemetry is necessary so that the procedure author knows what atomic elements are available to construct a procedure. Furthermore, the executive (manual or automated) must know how to get information from, or send action to, the controlled system and in what format. Ideally this representation of commands and telemetry should come from the hardware designer or vendor. We have chosen an industry standard representation called XML Telemetric and Command Exchange (XTCE) (http://space.omg.org/xtce/index.htm) for representing commands and telemetry.

We have modified XTCE slightly to allow for the description of generic systems, which can then be instantiated into specific systems. For example, we want to represent a generic RPCM that has generic commands and telemetry common to all RPCMs. Then we can instantiate a specific RPCM for each instance in the space station. As shown in the next section, this allows us to write procedures for any RPCM and have the specific RPCM upon which we want to execute the procedure be specified at procedure execution time. The procedure shown in Figure 1 is just such a procedure.

IV. A procedure example

In this section we look at the procedure shown in Figures 1 and 2 and see how different parts of it are represented in PRL. The first thing to note about this procedure is that it applies to any RPCM. So whatever is calling the procedure will need to pass in the specific RPCM that is being configured. This is done using a parameter. In PRL this would look like the following:

```
<Parameter Id=‘‘affectedRPCM’’, parameterType=‘‘In’’,
externalType=‘‘xtce:RPCM’’/>
```

where the Id is how we reference this parameter, the parameterType is either an Input parameter, an Output parameter or both and the externalType says that the parameter will be an RPCM, which is defined in the XTCE file for this system. Then, the procedure author can call commands on this local variable, similar to calling methods on an object. For example, the first command in step one of the example procedure is a common clear command:

```
affectedRPCM.CommonClear
```

This will issue the common clear command on whatever RPCM is passed to this procedure.

The for each instruction in step two is a little more complicated because you only want to close inhibit certain pre-defined RPCs of the RPCM. In these cases you would create a *list* if type RPC then call a function on the RPCM that returns a list of RPCs that need to be close inhibited as defined in the system representation. For example:

```
list_of_rpc = affectedRPCM.GetSubcomponentList.RPC(close_inhibit)
```

Then you can loop through the list using an iterator of type RPC and perform the close inhibit command. For example (using Y as the iterator of type RPC):

```
Y.close_inhibit
```

Here is an example of the first few instructions in the first step of procedure 5.420 (see Figure 1) in pseudo-PRL (that is, PRL without the XML tags that make it so verbose):

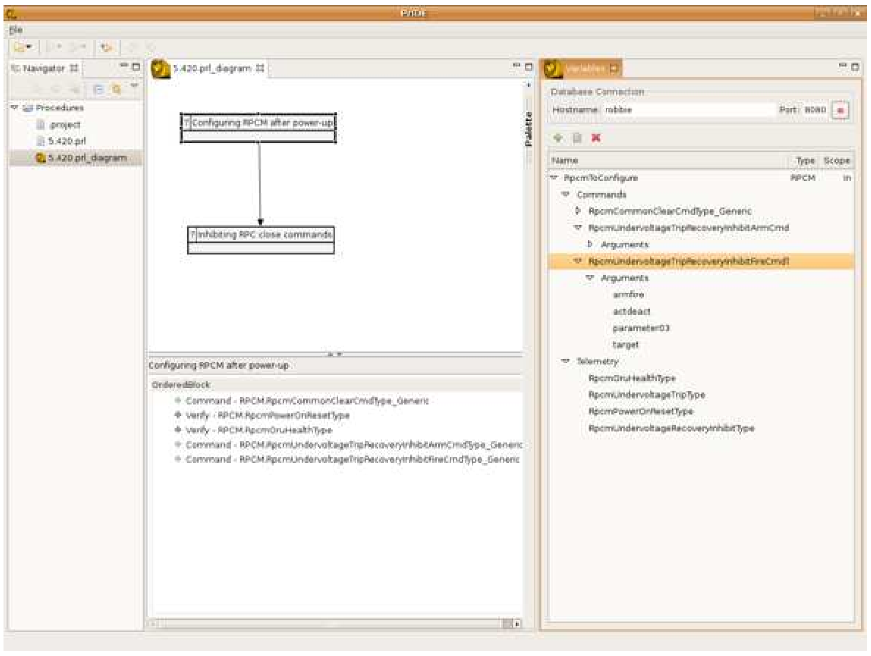
```
Procedure

  Parameters
    id=‘‘affectedRPCM’’ externalType=‘‘RPCM’’ parameterType=‘‘In’’
  LocalVaribles
    id=‘‘Y’’ externalType=‘‘RPC’’
  ExitModes
    id=‘‘exit_success’’ Message=‘‘Procedure exited successfully’’
    id=‘‘exit_verify_fail’’ Message=’’Procedure failed on verify’’
  Procedure Title
    name=‘‘5.420 RPCM Power On Reset’’ id=‘‘proc_5420’’

  Step step_id=‘‘step_1’’ title=‘‘Configuring RPCM after powerup’’
    OrderedBlock  block_id=‘‘block_1’’
        CommandInstruction instruction_id=‘‘instr_1’’
            Description ‘‘Common Clear’’
            CommandIdentifier affectedRPCM.CommonClear
        VerifyInstruction instruction_id=‘‘instr_2’’
            ExitModeReference=‘‘exit_verify_fail’’
            Description ‘‘Power On Reset -- blank’’
            Value affectedRPCM.PowerOnReset
            Operator ‘‘equal’’
            TargetValue ‘‘blank’’
    GotoStep stepRef=‘‘step_2’’
```

Note that only the first two instructions are given above as the rest are similarly formed. The ExitModes are returned by the procedure execution when the procedure ends. There can be as many different exit modes as the authors wants. In this case the `exit_success` mode is returned when the procedure completes. The `exit_verify_fail` is explicitly referenced in the verify instruction above and is returned if the verify fails (which causes the entire procedure to end). This provides knowledge to another system (e.g., a planner) or to a human as to what happened with the procedure execution. Command and verify instructions have human-readable descriptions as well as giving a command identifier or a verify value and target. The final line says that after executing this step go to the step with identifier `step_2`.

While procedures of this type are clearly harder to author than the simple English text of today's procedures, they offer the opportunity to assist humans in procedure execution. To help the procedure author we are developing tools that are described in the next section.



PRIDE
File
Navigator
Procedures
.project
5.420.prl
5.420.prl_diagram
5.420.prl_diagram
Configuring RPCM after power-up
Inhibiting RPC close commands
Palette
Configuring RPCM after power-up
OrderedBlock
Command - RPCM.RpcmCommonClearCmdType_Generic
Verify - RPCM.RpcmPowerOnResetType
Verify - RPCM.RpcmOruHealthType
Command - RPCM.RpcmUndervoltageTripRecoveryInhibitArmCmdType_Generic
Command - RPCM.RpcmUndervoltageTripRecoveryInhibitFireCmdType_Generic
Variables
Database Connection
Hostname robbie
Port 8080
Name
Type
Scope
RpcmToConfigure
RPCM
in
Commands
RpcmCommonClearCmdType_Generic
RpcmUndervoltageTripRecoveryInhibitArmCmd
Arguments
RpcmUndervoltageTripRecoveryInhibitFireCmd1
Arguments
armfire
actdeact
parameter03
target
Telemetry
RpcmOruHealthType
RpcmUndervoltageTripType
RpcmPowerOnResetType
RpcmUndervoltageRecoveryInhibitType

Figure 3. A procedure authoring tool implemented in Eclipse.

V. Developing procedures

Goal-based procedures will need to be authored, viewed, verified, validated and managed by a variety of people, many of whom will not understand XML or other representations and who will not be familiar with the requirements of a goal-based system. We are developing a Procedure Integrated Development Environment (PRIDE) that will provide an integrated set of tools for developing and verifying procedures. PRIDE offers both graphical and textual editing tools with syntax checking and enforcing of syntax constraints. Verification and validation tools check the procedure against flight rules and system constraints. Verification and validation tools also allow for running the procedure against low-fidelity simulations at development time. We are using the open source Eclipse (www.eclipse.org) platform to implement PRIDE. Figure 3 shows a prototype of PRIDE. Authors can drag from a palette representing PRL constructs onto a graphical view of the procedure. Additional palettes allow for dragging commands and telemetry from the system representation described in Section III. Once a procedure is authored, PRIDE can output the instantiated PRL XML file for use by other tools.

We are integrating verification and validation tools into PRIDE. Some will be static checkers that will do things like type-checking, finding steps that are never reached by branches, etc. Others will connect to system simulations, execute the procedure and provide feedback to the author. These might be very low-fidelity simulations (e.g., state machines) that can catch simple errors. The current validation process uses the full-up training simulations for shuttle and station (i.e., the SES and SSTF) to test new procedures at considerable expense.

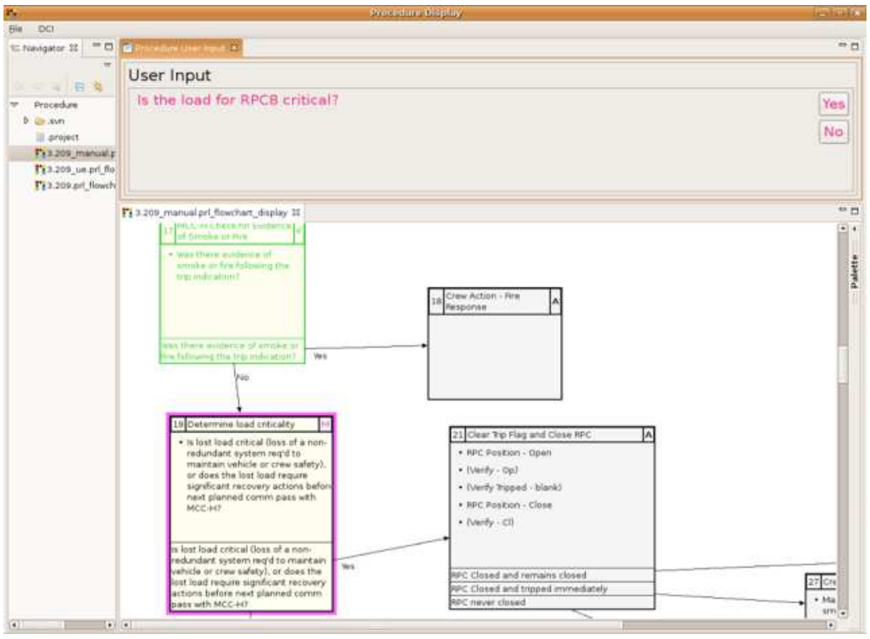

Procedure Display
File
DCI
Navigator
Procedure
.svn
.project
3.209_manual.p
3.209_ue.prl_flo
3.209.prl_flowch
Procedure User Input
User Input
Is the load for RPCB critical?
Yes
No
3.209_manual.prl_flowchart_display
Palette
17
MCC-H Check for Evidence of Smoke or Fire
• Was there evidence of smoke or fire following the trip indication?
Was there evidence of smoke or fire following the trip indication?
Yes
No
18
Crew Action - Fire Response
A
19
Determine load criticality
• Is lost load critical (loss of a non-redundant system req'd to maintain vehicle or crew safety), or does the lost load require significant recovery actions before next planned comm pass with MCC-H?
Is lost load critical (loss of a non-redundant system req'd to maintain vehicle or crew safety), or does the lost load require significant recovery actions before next planned comm pass with MCC-H?
Yes
21
Clear Trip Flag and Close RPC
A
• RPC Position - Open
• (Verify - Op)
• (Verify Tripped - blank)
• RPC Position - Close
• (Verify - Cl)
RPC Closed and remains closed
RPC Closed and tripped immediately
RPC never closed
27

Figure 4. The end user interface for display procedures. It is updated using status data from the execution engine.

VI. Executing procedures

Adjustable autonomy is the ability of autonomous systems to operate with dynamically varying levels of independence, intelligence and control.[2] The goal is to minimize the *necessity* for human interaction, but maximize the *capability* to interact. In the case of procedures that means that the human can choose autonomous execution, the traditional manual execution or a mixture of the two. We execute autonomously using an *execution engine*, which interprets the procedure representation and issues commands to the underlying system. We are using both the RAPS execution engine[3] and the PLEXIL-based Universal Executive in our experiments. There is a long history in artificial intelligence research of procedure execution systems including.[4–7] There are also several space-oriented executives including Timeliner from Draper Laboratories (current used in ISS) and the Spacecraft Command Language (SCL) from Interface and Control Systems Inc., which is used on several unmanned missions and is baselined for use on CEV/Orion. We have developed both PRL to PLEXIL and PRL to RAPS translators that allow for direct execution of PRL using those execution engines. PRL could also be automatically translated into Timeliner or SCL routines. We are using off-the-shelf execution engines and claim no research advances in this area.

The user sets the level of autonomy (LOA) through the procedure display. This is transmitted to an LOA server that the execution engine can query for which parts of the procedure it should execute autonomously and which parts it will prompt the human to execute. There are three levels of autonomy. The first level is purely autonomous and the execution engine will send commands and check telemetry with no human involvement. The second level is for the execution engine to ask consent of the human before autonomously executing a procedure, step or instruction. The third level is for the human to actually execute the step or instruction themselves without using the execution engine. Manual execution is performed via an end-user procedure display. This display reads in the PRL and shows the procedure to the user. The user can manipulate the LOA of the procedure and they can view the current procedure status and relevant telemetry. The end user display is written in Java and built on the same Eclipse open source platform as PRIDE (see Figure 4).

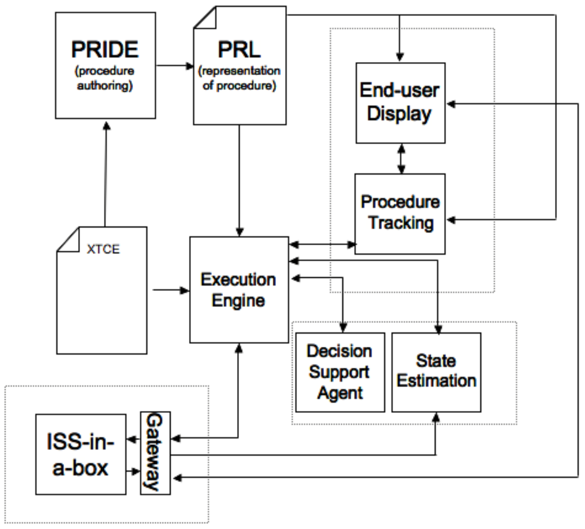


Figure 5. An adjustably autonomous procedure execution architecture.

VII. Integrated scenario

We are testing our adjustably autonomous approach using a high-fidelity simulation of the International Space Station (ISS). We have encoded several commonly executed ISS procedures in PRL. As Figure 5 shows, the PRL is automatically translated into the language used by the execution engine and also used to generate the end-user display. The execution engine sends commands to the system (the ISS simulation in our case) and receives telemetry. A procedure tracker maps executive engine status messages to completion of procedure steps and displays these to the end user. Through the display the end user can choose the level of automation of any step in the procedure. This is passed to the execution engine, which acts appropriately.

There are two additional boxes in the architecture: state estimation and the decision support agent. The former takes in raw telemetry from the system and produces more abstract state information. This more abstracted state information can be used in the PRL as part of the automation conditions or as part of the conditional branches. Of course, the procedure author would need to know what abstracted state variables are available. We are exploring the use of State Chart XML (SCXML) (see http://www.w3.org/TR/scxml/) to document state variables similar to how we use XTCE to document telemetry and commands. We are also exploring different algorithms for state determination, many of them model-based and similar to systems like Livingston[8] and HyDE. The decision support agent is a planner that produces a set of crew and system activities on a timeline. It is integrated with procedure execution so that the appropriate procedure is either executed autonomously or manually at the appropriate time. Resource information can be obtained by the planner directly from the PRL. Current shuttle and station operations involve significant manual planning,[9] but we are demonstrating the effectiveness of automated planners such as Europa.[10]

VIII. Conclusion

Procedures will continue to be a vital component of space missions – crewed and uncrewed. Allowing for goal-oriented, adjustably autonomous procedure execution will increase efficiency and safety of space missions. Over the past year we have developed software for developing and executing goal-based, adjustably autonomous procedures. We have also developed several goal-based representations for capturing procedure and system knowledge. Experiments with a high-fidelity space station simulation have shown the relevance and impact of adjustably autonomous procedure execution.

IX. Acknowledgements

The authors wish to thank colleagues at NASA Ames Research Center who participated in discussions underpinning many of the ideas presented in this paper, including Ari Jónsson, Vandi Verma, Michael Dalal and Michael Freed. Colleagues at NASA Johnson Space Center contributed to this paper including Lui Wang and Bebe Ly. The authors also wish to thank S&K Aerospace employees Scott Bell, Kevin Kusy, Tod Milam, Carroll Thronesbery and Mary Beth Hudson who helped implement many of the software processes described in this paper. Bob Phillips and David Cefaratti of L3 Com participated in definition of the Procedure Representation Language. Wes White and Michel Izygon of Tietronix Inc. provided assistance in understanding space station procedures. This work is supported by NASA's Exploration Technology Development Program (ETDP) under the Spacecraft Autonomy for Vehicles and Habitats (SAVH) project.

References

[1]Verma, V., Jonsson, A., Pasareanu, C., Simmons, R., and Tso, K., "Plan Execution Interchange Language (PLEXIL) for Executable Plans and Command Sequences," *Proceedings of the International Symposium on Artificial Intelligence, Robotics and Automation in Space (i-SAIRAS)*, 2005.

[2]Dorais, G., Bonasso, R. P., Kortenkamp, D., Pell, B., and Schreckenghost, D., "Adjustable Autonomy for Human-Centered Autonomous Systems on Mars," *Proceedings of the Mars Society Conference*, 1998.

[3]Firby, R. J., *Adaptive Execution in Complex Dynamic Worlds*, Ph.D. thesis, Yale University, 1989.

[4]Georgeff, M. P. and Ingrand, F. F., "Decision-Making in an Embedded Reasoning System," *International Joint Conference on Artificial Intelligence*, Aug. 1989, pp. 972–978.

[5]Agre, P. E. and Chapman, D., "What are Plans For?" *Robotics and Autonomous Systems*, Vol. 6, 1990.

[6]Gat, E., "ESL: A Language for Supporting Robust Plan Execution in Embedded Autonomous Agents," *Proceedings of the 1997 IEEE Aerospace Conference*, 1997.

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

[8]Williams, B. C. and Nayak, P. P., "A model-based approach to reactive self-configuring systems," *Proceedings of the National Conference on Artificial Intelligence (AAAI-96)*, 1996.

[9]Korth, D. and LeBlanc, T., "International Space Station Alpha Operations Planning," *Proceedings of the 3rd International NASA Workshop on Planning and Scheduling for Space (available from The Institute for Advanced Interdisciplinary Research Houston Texas)*, 2002.

[10]Jonsson, A., Morris, P., Muscettola, N., and Rajan, K., "Planning in interplanetary space: theory and practice," *Proceedings of the Fifth International Conference on AI Planning and Scheduling (AIPS)*, 2000, pp. 177–186.