Procedures as a Gateway to Spacecraft Autonomy

David Kortenkamp and **R. Peter Bonasso** and **Debra Schreckenghost**
TRACLabs Inc. at NASA Johnson Space Center ER2
Houston Texas 77058
kortenkamp@jsc.nasa.gov

Abstract

This paper examines the role of procedures in operating crewed spacecraft and how procedures can serve as an entry point for spacecraft autonomy. Procedures represent the knowledge necessary to operate a system and are a critical component of crewed spaceflight. Historically, procedures have been executed by humans reading from a piece of paper or a static display. In this paper we describe a new procedure paradigm in which procedures are computer-understandable and can be executed by either humans or computers. This requires new procedure and system representations, development tools, autonomy software and displays. A prototype procedure development and execution system is described and applied to a space station procedure.

Introduction

Spacecraft autonomy involves using software systems to minimize the need for human (crew members or ground control personnel) intervention in routine spacecraft operations. Implementation of spacecraft autonomy for a crewed mission requires a careful examination of how crewed spacecraft are currently operated and decisions about where to insert autonomy software. A potential insertion point for on-board autonomy is procedure execution. For current crewed missions (i.e., Space Shuttle and Space Station), procedures are primarily executed manually by crew members or ground controllers and are not designed for autonomous execution. There is a very crisp line between on-board autonomy software, which mainly deals with real-time control and anomaly response, and manual procedures, which deal with system moding and reconfiguration, mission goals, and complex diagnosis and recovery. This paper examines research that blurs the crisp line between creating spacecraft procedures that are designed for either automated or manual execution. This concept, called *adjustable autonomy*, will allow for incremental automation of spacecraft operations as crew and ground controllers become more and more comfortable with automated technologies executing formerly manual procedures while still allowing for manual execution when desired.

Procedures

Procedures encode the operational knowledge of a system as derived from system experts, training and experience. They are essential to the safe and efficient operation of any complex system. NASA has literally tens of thousands of procedures for Space Shuttle and Space Station, both for the flight controllers and for the crews. They are the accepted means by which any user commands a system.

Procedures have a complicated and involved lifecycle. The lifecycle of a procedure starts with knowledge acquisition about the system, its operating requirements and the mission goals it supports. The system can be an on-going vehicle such as the Space Station or a one-time payload such as a scientific experiment. Knowledge acquisition leads to procedure editing during which the first instance of a procedure is created. The procedure is then validated and verified manually using simulators – feedback to the procedure editor about any changes to the procedure is given at this time. A validated procedure is handed off to training personnel, which develop a training plan for the end user. Again, there may be feedback to the procedure editor. Finally, the procedure is ready for execution in a flight situation. Supporting this procedure lifecycle, especially as we move to flexible procedures, creates many software engineering challenges and requires an integrated suite of procedure development tools.

Procedures today

Space shuttle procedures are written using Microsoft Word and then printed and bound into books that fly with every shuttle mission. These books, along with additional ground procedures and detailed reference material, are also developed for flight controllers. This system, while inflexible, works well for a stable, known vehicle that changes little between missions. Space Station procedures are also currently authored using Microsoft Word. However, rather than being printed, they are instead converted to an eXtensible Markup Language (XML) representation and viewed using the International Procedure Viewer (IPV). This has the advantage of reducing paper (and thus weight) and making updates much easier. This is appropriate for an

evolving piece of hardware. However, for all of this progress, the XML representation is still presentation based because it is only used to denote where and how to display a procedure (e.g., what font, what indentation, whether bold or plain). The current XML representation is not capable of denoting the content of the procedure such as what the procedure is supposed to accomplish and in what context. In essence, the procedures are machine readable for purposes of display to a human but they are not machine understandable for purposes of autonomous execution of those procedures.

Procedure example

Figure 1 shows a snippet of a Space Station electrical power system (EPS) malfunction procedure. Procedures are typically divided into two types: 1) malfunction procedures, which help users to troubleshoot and diagnose a problem; and 2) checklist procedures, which guide users through nominal spacecraft operations. The EPS malfunction procedure triggers when event code 5520 is issued by the Space Station caution and warning system. The triggering of this malfunction procedure is manual, i.e., a ground controller or crew member notices the 5520 event code, pulls up the appropriate procedure and begins executing it. Execution consists of following the steps of the procedure. Step 1 checks to see if the event code has cleared on its own within three minutes. If so, execution proceeds to Step 2. If not, execution proceeds to Step 4. Let's assume the advisory clears. Step 2 is simply a warning to the user that the next step will have the consequence that data will be lost. Execution then proceeds to Step 3, which actually clears the data. The details in Step 3 are peculiar to Space Station operations in that the user executing the procedure navigates through Space Station displays in order to issue commands. The "sel" operations are selections of display windows. The final **cmd** is represented as a button on the displays that the user needs to press. Another interesting aspect of this procedure is Step 5. This is a call out to execute another procedure (a checklist procedure in this case) and then return and continue. We will use the first three steps of this malfunction procedure throughout this paper to illustrate our ideas.

The bigger picture

Procedures, while important, are only a small part of the process of operating a crewed spacecraft. Procedures encode the means by which a spacecraft can be operated and are relatively static. Procedures do not specify when (in time) they should be executed or for what purpose. This is the role of a mission *plan*. A mission plan is developed well before a mission and takes into account the various mission goals, personnel, time, resources and constraints. For example, mission planning on Space Station can begin up to a year before a new crew goes to station in involved consultation with international partners, scientists, public affairs, etc. (Korth & LeBlanc 2002; Balaban *et al.* 2006). As a mission approaches, the plan becomes more and more detailed and fixed. As the mission unfolds, small parts of the plan are uploaded to the crew for execution. For Space Station, a weeklong plan, called an Onboard Short Term Plan (OSTP), is sent up daily for the next week of the mission. That plan consists of a partially ordered set of *tasks* on a timeline for each crew member. Tasks can be either procedures or activities. Procedures we have discussed in length and the plan states when a procedure should be run and by whom. Activities are things the crew or automation do that do not have procedures, such as sleeping, eating, automatic data downloads, etc. Mission planners use procedures as their building blocks to create a successful plan that meets mission goals and is safe.

Building and executing procedures

A variety of tools, representations and software processes are necessary to build and execute procedures in an adjustably autonomous fashion. Ideally these tools, representations and processes are integrated such that they easily share information. In this section we present the tools, representations and software processes that we have developed for building and executing procedures.

Procedure representation

Procedures will need to be machine understandable in order to support adjustable autonomy concepts. This means that procedure content must be encoded. 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 success 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. The procedure execution assistance tools will need additional content information including paraphrasing knowledge for presentation, speech generation and links to embedded images or videos for just-in-time training.

The first three steps of the EPS malfunction procedure shown in Figure 1 are represented in a presentation-based representation similar to PDF. While easily read and interpreted by a trained human, this plain text format is not understandable by a computer, and there is no information about what this procedure accomplishes or what is necessary in order for the procedure to be used. It also requires the end user to navigate through complicated displays to issue a simple command.

We have been developing a new procedure representation called the Procedure Representation Language

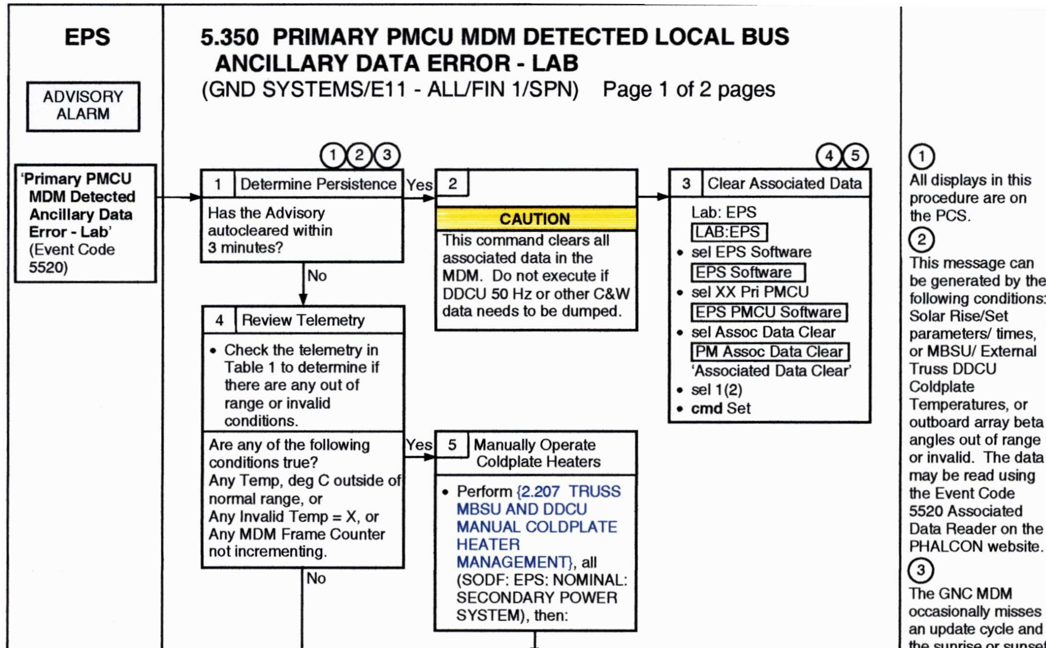

Figure 1: First five steps of an EPS malfunction procedure.

```
<Step stepId="s_1">
  <StepTitle>
    <StepNumber>1</StepNumber>
        <Text>Determine Persistence</Text>
  </StepTitle>
    <StepContent>
     <ConditionalBranch instructionID="i1_0" waitForBranch="true">
          <BranchTo>
             <EQ>
                <DataReference source="ontology"><ID>CW_5520</ID></DataReference>
                <Number>0</Number>
             </EQ>
             <GotoStep startStep="s_2" instructionID="i1_00">
                <DepartureDescription><Text>Go to step 2</Text></DepartureDescription>
             </GotoStep>
          </BranchTo>

          <BranchTo>
             <GT>
                <ElapsedTime><TimeReference><ID>s_1</ID></TimeReference></ElapsedTime>
                <Time min="3"/>
             </GT>
             <GotoStep startStep="s_end" instructionID="i1_01">
                <DepartureDescription><Text>End</Text></DepartureDescription>
             </GotoStep>

          </BranchTo>
    </ConditionalBranch>
 </StepContent>
 </Step>
```

Figure 2: First step of the EPS malfunction procedure in PRL.

(PRL). This language keeps the user friendly display format of current procedures but augments this with content-based information required for autonomous execution or tracking. We accomplished this by integrating features of the PLEXIL automated execution language developed at NASA (Verma *et al.* 2005) with the existing XML schema for procedures. The PLEXIL representation is similar to (and inspired by) many existing procedural representations that have been effective in autonomous vehicle control over the years (e.g., (Georgeff & Lansky 1986; Simmons & Apfelbaum 1998; Firby 1987). Other domains have also looked at procedural representations with great success (Ram *et al.* 1998).

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 true means that the procedure can start
  - Repeat-until conditions: keep performing the procedure until this boolean expression becomes true
  - Pre-conditions: evaluated after the start condition and if true then begin executing the procedure
  - Post-conditions: evaluated after a procedure is done and if it evaluates to false then the procedure has failed
  - 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
  - Resources: any resources (time, fuel, crew members, power, tools, etc.) required for execution of this procedure
- Computations: Mathematical formulas that can be used throughout the procedure
- Local Variables: Declarations of any variables used internal to the procedure
- Contexts: Defines the optional sets of context-sensitive values that can be used to instantiate this procedure for execution
- 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
  - Step content, which contains an instruction consisting of the following:
    * Automation data, as above except for instructions
    * Commands: Either executed through software or manually through a crew member
    * Input: Information entered into a local variable from an external source such as a crew member
    * Verify: A check on telemetry or by a crew member of a specific system state
    * Procedure call: A call to another procedure
    * Wait: A wait until a boolean expression becomes true or until a particular amount of time has passed
    * Control statements such as if-then-else, while or for-each
  - 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.

Example Figure 2 shows the first step of the EPS malfunction procedure represented in PRL. First, note that PRL is an XML schema. Second, that branching and the conditions for branching are explicitly represented as boolean and goto statements. Third, the `waitForBranch` attribute determines if this conditional branch should wait for one of the conditions to be true (the "true" value) or fall through to the next step if none are true. Fourth, the `DataReference` tag means that this is an external signal that needs to be read in at runtime. The "ontology" attribute states that at execution time the executor should look into the system representation for where to find this value (see next section). Finally, notice that there are still human readable elements in this representation such as the step title and departure description. With all of the content of this step made explicit, it is now possible to automate its execution. In fact, Section will describe just such an automated execution.

System representation

Procedures describe the processes by which a device or system[1] 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, telemetry, states, state transitions, connections

[1]A device or system in this context should be interpreted broadly and can mean software, a robot or an entire vehicle or habitat

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.

Commands and telemetry 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.

States The representation of states and state transitions is necessary so that the procedure author can reference them in preconditions and postconditions (e.g., don't do this procedure when the device is in this state) and so that the executive can check these states when executing the procedure. The state representation of a device could come early in its design before the hardware implementation and before specification of commands and telemetry. This would allow for early development and testing of procedures against a state model of the device. While we could extend XTCE to add state information, we felt that a separate representation would be more powerful. We have chosen State Chart XML (SCXML) (http://www.w3.org/TR/scxml/) for representing states and state transitions.

Taxonomy A system representation also needs to include the components of the system and the relationship between components – this is usually called a taxonomy. There are two kinds of relationships we expect to capture. First there is a hierarchical relationship between spacecraft components. For example, a spacecraft consists of many systems – power, life support, navigation, propulsion, etc. Each system has many subsystems and subsystems have components (valves, tanks, switches, etc.). Hierarchy and containment are important to represent in order to allow for efficient display and reasoning. The second kind of relationship is connectivity between spacecraft components. For example, the output of an oxygen generation system may be connected to an oxygen storage tank or a battery may be connected to a power distribution component. Connectivity is important to represent so that situation awareness displays can be built for humans. We are still in the process of determining appropriate taxonomy representations – existing standards such as XML Metadata Interchange (XMI) may be appropriate.

System-level constraints Any complicated system will have overarching constraints and flight rules. An example is the flight rule that the Space Station shouldn't thrust when there is an Extravehicular Activity (EVA). This type of information is not captured in PRL or in the various files discussed in the previous sections. Currently flight rules and other global constraints are captured in documents written in natural language. This makes it impossible for software tools to reason over these constraints and check for consistency and applicability. This also makes it possible to check constraints during run-time to enhance safety. We are in the early stages of exploring how to capture these flight rules and global and are considering the Object Constraint Language (OCL) of UML as a possible representation of system constraints.

Dependencies Each of these system representations and the procedure representation depend upon each other. The procedure representation (PRL) needs to reference commands, telemetry and states. The system state representation needs to map telemetry to states and commands to state transitions. The system-level constraints will be between system states. A core set of components will be stored in the XTCE representation. The taxonomy will use that to create hierarchies and connections. An additional representation is needed to store graphical information such as the location of information on a screen or its attributes (color, font, icons, etc.). A rendering program can use this display representation in conjunction with the taxonomy, the XTCE and the system state representations to provide situational awareness.

Procedure development environment

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. We are developing a Procedure Integrated Development Environment (PRIDE) that will provide an integrated set of tools for dealing with procedures. We are using the open source Eclipse (www.eclipse.org) platform to implement PRIDE.

Functions We have identified the following functions of a procedure integrated development environment:

- Authoring
  - Graphical
  - Textual
  - Syntax checking (e.g, is procedure properly formed?)
  - Syntax constraints (e.g., is a variable assignment within range?)
- Viewing
  - Static views as seen by end user
  - Dynamic views that can change based on real-time inputs
- Interaction
  - Ability to send information outside of PRIDE
- Verification and Validation

  - Check procedures against flight rules
  - Check procedures against system constraints
  - Assist in evaluation of simulation results
- Configuration and workflow management
  - Verifying links to other procedures
  - Interaction with a procedure repository
  - Interaction with workflow processes (e.g., procedure sign-off)

In the rest of this section we will focus on how we are providing the graphical authoring function in PRIDE.

Authoring Procedures will initially be authored by system experts. The experts will not be programmers and will not have familiarity with XML or content oriented representations. We need to develop authoring tools that allow experts to express their intents without needing to understand the details of the underlying language. We also expect procedures to be edited by multiple experts, each with different goals. For example, the system expert will outline the initial procedure. A human factors expert may then annotate that initial procedure with information about how to display it to the end user. An automation expert may further annotate the procedures with information pertinent to automated procedure execution.

To implement the procedure authoring function in PRIDE we are taking advantage of the Eclipse Graphical Modeling Framework (GMF) to import the PRL XML schema and convert it to an Eclipse Modeling Framework (EMF) representation. From that we can create a palette that represents PRL structure and commands. Authors can drag from that palette onto either a graphical view of the procedure (see Figure 3). Additional palettes allow for dragging commands and telemetry (from the XTCE file described in Section), states (from the SCXML file described in Section) or callouts to other procedures. Once a procedure is authored, PRIDE can output the instantiated PRL XML file for use by other tools.

Automated procedure execution

Procedures can be executed autonomously using an *execution engine*, which interprets the procedure representation and issues commands to the underlying system. There has been a great deal of research in the last decade on procedural execution systems, with some of the more prominent being PRS (Georgeff & Ingrand 1989), RAPS (Firby 1989), Teleoreactive Programs (Nilsson 1994), Universal Plans (Schoppers 1987), APEX (Freed 1998) and TDL (Simmons & Apfelbaum 1998). The Space Station program also has a procedural execution system called Timeliner from Draper Laboratories. While underlying implementation details may change, all procedural executives have similar functions: 1) they have a library of applicable procedures; 2) they choose procedures that are eligible to run by matching start and pre-conditions with system states and telemetry in realtime; 3) they decompose hierarchical procedures into subprocedures; 4) they dispatch commands to lower level control processes; and 5) they monitor for relevant states in the system. We will expand on these functions in the next section.

Execution functions Most procedure execution systems have the following core functions:

- **Monitoring and State Management:** Maps the low-level signals and states to appropriate state conditions in the procedure. Checks for conditions to initiate step transitions or change resource allocations.
- **Procedure Selection**: Instantiates procedures and refines those procedures by decomposing procedures into steps, instructions, or calls to other procedures. This function also manages the acceptance, prioritization and selection of multiple applicable procedures.
- **Procedure Execution:** Dispatches commands to low-level control processes to bring about change in the operational environment and enables monitoring through sensors and states to monitor the effects of these commands. This function prioritizes the execution of concurrent activities and is responsible for transitioning procedure components (steps, instructions, etc.) through their operational states, e.g., inactive, completed, failed, etc..
- **Resource Allocation:** Assigns resources to eligible tasks and maintains these assignments based on task priority and resource availability.

Execution interfaces The procedure execution system needs information from a variety of sources and produces information that is needed by other processes. Specifically, the executive needs the following information to function correctly:

- The procedure tasks themselves in a procedure representation. This can be a static file or sent to the executive from a planner.
- Task execution information such as what tasks should be executed when or what tasks should be withdrawn from execution.
- Telemetry from the underlying system that is being controlled

The executive produces the following information that may be used by other processes:

- Commands, including parameters, that are dispatched to lower level control processes
- Task execution data such as status of the procedures, steps, and instructions (e.g., completed, failed, pending, etc.) and the status of resources
- State query responses in which external processes can query the executive's state and monitoring information

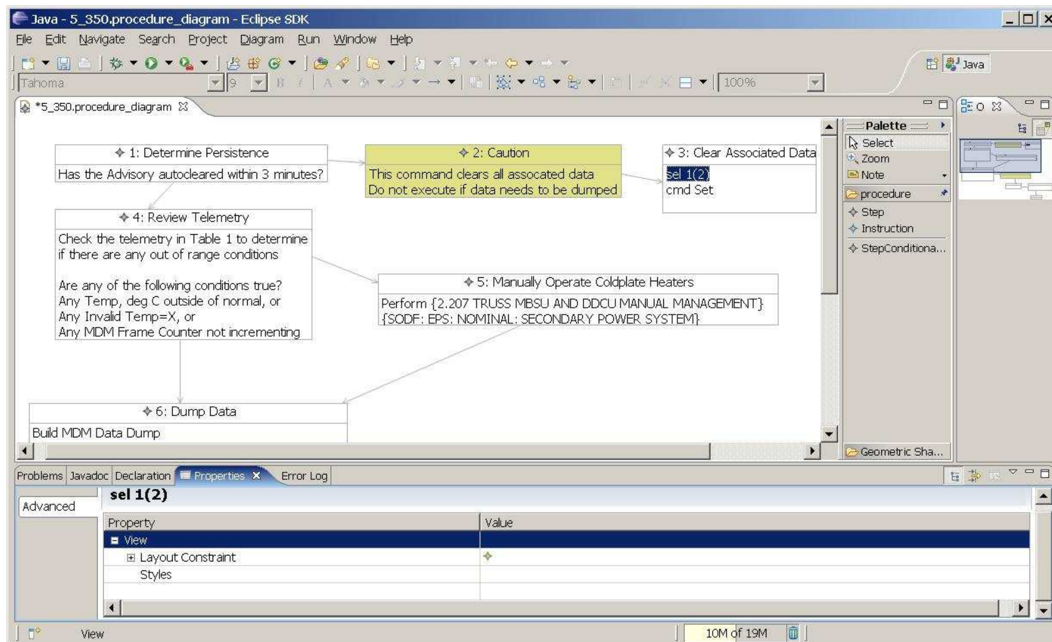

Figure 3: A procedure authoring tool implemented in Eclipse.

End user procedure display

Humans are an important part of the procedure process. The end user is the person who will be either executing the procedure or monitoring the autonomous execution of a procedure. The end user may also switch back and forth between manual and autonomous execution (e.g., execute the first two steps manually, then have the autonomous executive execute the next three steps, then execute the next step manually, etc.). Thus, the end user will need a display interface that supports this adjustable autonomy mode of operation. We hope to reuse much of the software that is built for authoring procedures (see Section) as the end user interface without the authoring capabilities. Ideally, the end user interface will look similar to the "paper" procedures of today (e.g., Figure 1), but allow for direct execution of commands and direct display of telemetry by the end user. The end user interface will also have to display the results of procedure tracking (next section) by highlighting or otherwise noting steps that have been completed, steps that are currently executing, and steps that are pending. Execution status, especially failures, will need to be made explicit. Support for adjustable autonomy requires an easy way to note steps that should be done manually and steps that should be done automatically.

For a single procedure being done by a single end user the display requirements are not demanding. However, if we begin to address multiple procedures being executed in parallel by one or more end users with interaction between them, then a wider variety of issues arise. These include notification of the status of other end users or executives, interruption reasoning, and multi-modal notification. For example, a procedure may have a step that is a wait for some lengthy period of time (say one hour) at which point an end user could begin another parallel procedure. At the end of the hour the end user would need to be notified appropriately, find a break-point in their new procedure, return to the old procedure, get situated, and begin executing. In another example, two end users that are separated by distance may be performing a single procedure (say an EVA astronaut and a ground controller) and need to coordinate their activities. We have begun addressing these issues and will integrate our results into our expanding tool suite (Schreckenghost 1999; Schreckenghost *et al.* 2002).

Procedure tracking

Procedures will continue to be executed manually in upcoming space missions. There will be actions that can only be done by a person for physical or operational reasons. This poses problems for an adjustably autonomous approach to procedure execution. In a purely automated approach the executive knows exactly what is being done and what the status is. However, if some parts of the procedure are manual, then the current execution status will need to be inferred from telemetry or by direct query to the end user. The procedure tracking process does this inference. It uses all available data to determine which step of the procedure is being executed and what the execution status is. It then makes this available to other processes such as the executive and the end user display.

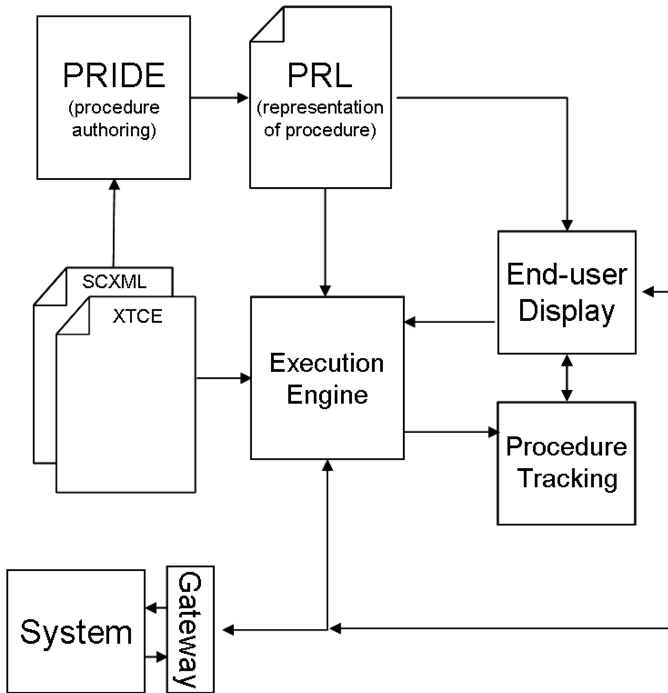

Figure 4: A procedure building and execution architecture.

Case study

We have assembled the components described in the previous section into a procedure building and execution architecture (see Figure 4 and used this architecture to execute the first three steps of the EPS malfunction procedure (Figure 1). Our current PRIDE implementation is capable of handling only a subset of PRL, so we encoded the EPS malfunction procedure directly into a PRL XML file. This was then translated by hand into a representation used by RAPS (Firby 1987). The execution engine used in our case study is RAPS, which is implemented in LISP. We represented the commands and telemetry required for this procedure in XTCE and then hand-translated that into the RAPS representation. RAPS was connected via a CORBA gateway to a low-fidelity simulation of the International Space Station. We are in the process of replacing the low-fidelity simulation with a high-fidelity simulation and will report on that in the final version of this paper. The execution engine sends execution status via CORBA to a procedure tracker implemented using a temporal pattern recognition system called CERA (Fitzgerald *et al.* 2006). The procedure tracker, in turn, determines when steps of the procedure have been started, finished, failed, etc. and passes this information via CORBA (and some intermediary processes) to an end user interface written in Eclipse that displays the procedure steps and changes their colors as their execution status changes.

Future work: Dynamically reconfigurable procedures

The process described in this paper focuses on procedures that are authored well before a mission and that are relatively static throughout the lifetime of a device or vehicle. In the future we envision a paradigm shift where procedures are treated as evolving, reconfigurable and reusable entities instead of the static, inflexible entities used currently. We envision a database of procedure fragments, each of which is intended to perform a single, well-defined operation. Procedures to accomplish larger multi-step operations are assembled and verified on-the-fly from these smaller procedure fragments, similar to what humans do currently by hand. In early missions, procedures could be assembled on the ground with significant human oversight. However, we envision moving all procedure fragments on-board and having them automatically assembled for crew members in response to their daily tasks and current mission context. Because future exploration missions will likely use modular, reconfigurable systems, it will be impossible to predefine all of the possible procedures for all possible system configurations. By carefully defining our procedure representation we can ask system vendors to deliver procedure fragments for operating their system that will be added to the database. Thus, when systems are integrated, procedures could also be integrated with ease. The procedure representation, authoring tools, and execution engines currently under development are key components of the move to dynamically reconfigurable procedures.

Conclusion

Procedures will continue to be a primary component of crewed spaceflight operations. By developing new tools, representations and software we can enable adjustably autonomous procedures in which humans and automation work together to handle basic spacecraft operations. This can result in more efficient and safe spacecraft operations.

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 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 Technologies 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) un-

der the Spacecraft Autonomy for Vehicles and Habitats (SAVH) project.

References

Balaban, E.; Orosz, M.; Kichkaylo, T.; Goforth, A.; Sweet, A.; and Neches, R. 2006. Planning to explore: Using a coordinated multisource infrastructure to overcome present and future space flight planning challenges. In *Proceedings of the AAAI 2006 Spring Symposium on Distributed Plan and Schedule Management (available from AAAI Press at www.aaai.org).*

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. 1989. *Adaptive Execution in Complex Dynamic Worlds*. Ph.D. Dissertation, Yale University.

Fitzgerald, W.; Firby, R. J.; Phillips, A.; and Kairys, J. 2006. Complex event pattern recognition for long-term system monitoring. In *Proceedings of the AAAI 2003 Spring Symposium on Human Interaction with Autonomous Systems in Complex Environments (available from AAAI Press at www.aaai.org*.

Freed, M. 1998. Managing multiple tasks in complex, dynamic environments. In *Proceedings of the 1998 National Conference on Artificial Intelligence*.

Georgeff, M. P., and Ingrand, F. F. 1989. Decision-making in an embedded reasoning system. In *International Joint Conference on Artificial Intelligence*, 972–978.

Georgeff, M., and Lansky, A. 1986. Procedural knowledge. *IEEE Special Issue on Knowledge Representation* 74(1):1383–1398.

Korth, D., and LeBlanc, T. 2002. International space station alpha operations planning. In *Proceedings of the 3rd International NASA Workshop on Planning and Scheduling for Space (available from The Institute for Advanced Interdisciplinary Research Houston Texas).*

Nilsson, N. J. 1994. Teleo-reactive programs for agent control. *Journal of Artificial Intelligence Research* 1(1).

Ram, A.; Catrambone, R.; Guzdial, M. J.; Kehoe, C. M.; McCrickard, D. S.; and Stasko, J. T. 1998. Pml: Representing procedural domains for multimedia presentations. Technical Report GIT-GVU-98-20, College of Computing, Georgia Institute of Technology.

Schoppers, M. 1987. Universal plans for reactive robots in unpredictable environments. In *Proceedings of the International Joint Conferences on Artificial Intelligence (IJCAI).*

Schreckenghost, D.; Thronesbery, C.; Bonasso, P.; Kortenkamp, D.; and Martin, C. 2002. Intelligent control of life support for space missions. *IEEE Intelligent Systems* 17(5).

Schreckenghost, D. 1999. Checklists for human-robot collaboration during space operations. In *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*.

Simmons, R., and Apfelbaum, D. 1998. A task description language for robot control. In *Proceedings Conference on Intelligent Robotics and Systems*.

Verma, V.; Jonsson, A.; Pasareanu, C.; Simmons, R.; and Tso, K. 2005. Plan execution interchange language (PLEXIL) for executable plans and command sequences. In *Proceedings of the International Symposium on Artificial Intelligence, Robotics and Automation in Space (i-SAIRAS).*