Model-based Verification and Validation for Procedure Authoring

Guillaume Brat and **Dimitra Giannakopoulou**
Carnegie Mellon University - Silicon Valley
guillaume.p.brat@nasa.gov
dimitra.giannakopoulou@nasa.gov

Michel Izygon and **Emmy Alex**
Tietronix
michel.izygon@tietronix.com
emmy.alex@tietronix.com

Lui Wang
NASA Johnson Space Center
lui.wang-1@nasa.gov

Jeremy Frank
NASA Ames Research Center
jeremy.d.frank@nasa.gov

Arthur Molin
S&K Aerospace
amolin@ska-corp.com

1. Introduction

The "Apollo 13" movie was a great account of how dangerous human space flight is. It clearly showed that there is a fine line between mission success and catastrophic failures. Besides being a great thriller (we all wanted the astronauts to make it back safely, even though things look really bad for a while), it also offered a great look at how things are ran at NASA. The movie clearly showed that all activities are planned in the most minute details and described in procedures. It was true then, and, it is still true now for the Space Shuttle and the International Space Station (ISS).

Procedures are plans for crew (i.e. astronauts) and flight controllers (which provides guidance from the ground). There are literally thousands of them for the ISS and the Shuttle; they will also be used on the new vehicle, called Orion, being developed for NASA. Procedures are written to be very general; hence they may have to be adapted for different situations. The movie actually showed a dramatic example in which a power-up procedure had to be adapted so that the power load stayed under a certain amperage. That procedure was tried and (sort of) validated in a flight simulator before being given to the crew. The interactions between sub-systems were so subtle that the procedure could not get worked out on-board by the crew. It needed to be carefully adapted and validated on the ground. While technology has evolved since the Apollo area, system complexity has certainly not decreased. Therefore, procedure authoring, validation and verification are still highly critical activities.

This paper describes our effort in improving procedure authoring, and more specifically, what can be done to speed-up and improve their verification and validation (V&V). We start by justifying the need for better procedure V&V. We describe our illustrative example, the power system for the ISS. We then present the A4O (Autonomy for Operations) project, under which these technologies have been developed. Then we describe our analysis and the challenges we encountered during the analysis. Finally, we conclude with some lessons learned and present our future work.

2. Procedures for human space flight

Procedures for human space flight are different from software. Many procedures describe inherently manual activities performed by people. The Space Shuttle was an almost completely mechanical vehicle. However, the ISS is commandable from computers. In fact about 100K of commands are sent from ground to the ISS every year. The effect of executing each command is shown by telemetry that is sent back to ground. Controllers are responsible for checking that the telemetry matches the intent of the command. Procedures have thus grown more like software, and this trend is expected to continue. One of our objectives is to see if procedures can benefit from Software Engineering processes, and in particular, from a formal automated V&V process.

2.1 Procedure authoring process

Procedures have traditionally been written as plain english (somewhat abrupt as in a list of actions). They are now authored using MS Word; so, in that sense, they are in some electronic format. They also follow some strict guidelines on their format and content, but they are not formal per se. They are meant to be read by humans and interpreted. Some of the semantics is captured in the writing styles (such as tabulations), but they do not follow any common formal language. Moreover, there is a disconnect between the procedure semantics and the command and telemetry semantics; for instance, you can tell if an identifier (for a command or telemetry data) has changed, but not if the meaning (of the command or telemetry) has changed. Therefore, a procedure is understandable only in the context of a given dictionary of commands and telemetry.

Unlike software programs, procedures are meant to be fairly general. The intent is to capture as many cases as possible and specialize them at execution time given the current context. For example, operation of the same class of hardware, e.g. a power distribution unit, may differ depending on what is connected downstream. If there are critical loads, these must be switched over to some other power source prior to powering off the distribution unit. So, procedures are authored as generic procedures, but they must be verified as specific instances of some specific execution context.

Figure 1 shows the current procedure authoring process for the ISS or the Space Shuttle. Once the procedure has been changed, it goes through several review steps, all of

them performed by humans, or groups of humans. It is quite clear that it is a lengthy process. The multiple review steps should in principle prevent human error. However, most of the validation relies on the experience of the reviewers. Note that the final products are human readable, but most of them are formatted for the International Procedure Viewer.

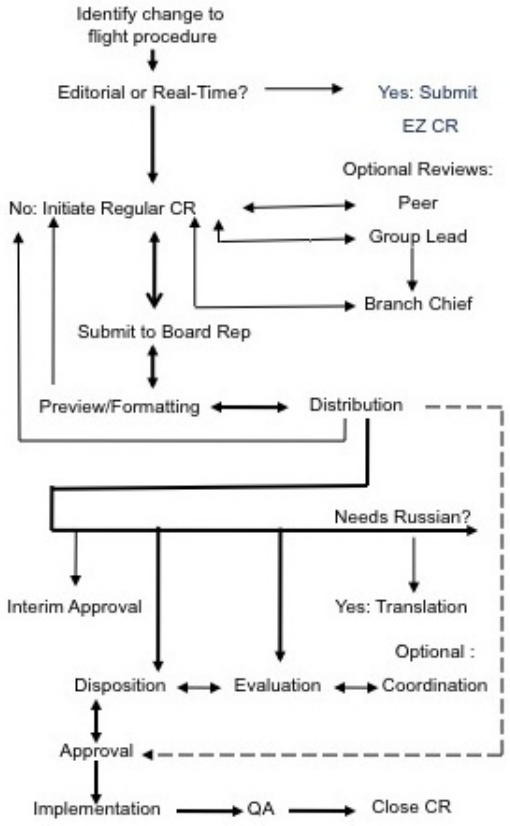


Figure 1: Authoring process for procedures.

2.2 State of the practice of Procedure V&V

Similarly to software, procedures need to be verified and validated. Procedures are tested not only at their inception, but also for each revision, and due to their generality, for each instantiation. Procedures are revised frequently because commands and telemetry change frequently. Moreover for the ISS, as buildup continues, procedures change and new ones. In-flight experience and hardware degradation also result in new or changed procedures.

As shown in Figure 1, procedure V&V is currently done through human reviews; high-fidelity testbeds are also used. Human reviews have the usual disadvantages of taking a long time and being error prone. Similarly, the problem with testbeds is that they are expensive (to create and maintain), which makes them a scarce resource. They usually are forward simulations, which are hard to reverse; initial conditions are also difficult to set up. Finally, generating test cases can be a problem.

2.3 Our Vision of Procedure V&V

Our vision is to improve on this current state of the practice by taking the best Software Engineering has to offer and bringing it to the world of procedure verification. Clearly, our V&V drivers are as follows.

1. Procedures are general, but they need to be verified for many possible executions
2. Procedures are somewhat informal; they need to be gradually formalized.
3. There can be mismatches between procedures and the dictionary of commands and telemetry.
4. Procedures change; they need to be tested not only at their inception, but also for each revision due to error correction or new hardware specification.

In previous work, Connors *et al.* have shown how how mismatches between procedures and dictionary of commands and telemetry can be addressed through static checking (Connors et al. 2009) if procedures are authored in a formal language, in this case, the PRL language described in 4.1 Brat *et al.* have also shown how to verify plans and procedures using various model checkers (Brat et al. 2008). However, the mismatch between the procedure language and the modeling language used by the model checkers was a big limitation of their approach. In this work, we are demonstrating the use of the Java Path Finder (JPF) model checker and its integration in the PRIDE environment, described in 4.3

Now, to enable the type of formal V&V we are envisioning (i.e. model checking), there needs to be formal models of the procedure, the system being commanded, and the possible execution contexts. Unfortunately, formal models are rarely available at NASA, or at least, they are rarely created as part of the software development process during, say, the design phase. This is particularly true for the ISS code base, which is an international effort and has been growing significantly over the years. The only available models are the ones used for the high-fidelity testbeds. They are derived from the real ISS flight software, but they are very complex and difficult to use stand-alone. Until recently, these models have been only available in highly contended testing facility, and, they were unsuitable for automated verification.

3. A Specific Example

3.1 ISS EPS Powerdown

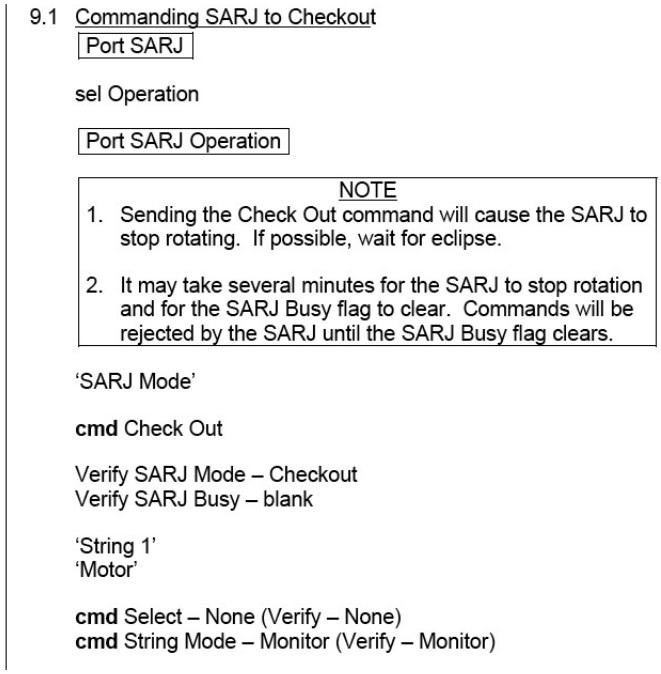
As most modern spacecrafts, the electrical system (EPS) on the ISS relies on solar power. For the ISS, it is produced by (currently eight) large solar arrays. Each solar array wing consists of two retractable "blankets" of (approximately 33000) solar cells with a mast between them. The solar arrays track the Sun using gimbals to follow the Sun as the space station moves around the Earth and to adjust for the angle of the space station's orbit to the ecliptic. The rotation to track the Sun is provided by a mechanism called the solar alpha rotary joint (SARJ).

In our work, we focused on EPS powerdown procedures, and more specifically, the ones for powering down all loads

powered by the DC-to-DC converters (DDCU). For example, the Powerdown (1.252) procedure, which is used to powerdown S0 DDCU S01A converters, is a 12-page document, which calls 17 other procedures that are nested as many as three procedure calls deep. There are 47 commands to execute and 63 telemetry items need to be verified (i.e., controllers need to check that the returned telemetry values are the expected ones) during execution by the controllers. There are also seven conditional execution steps. If you put all the documents together, it might be as many as 50 pages of text. Given the size, it is preferable to automate the V&V process rather than relying on humans. To illustrate V&V issues, we focused on the commands involved in configuring the SARJs and chose to demonstrate a subtle timing bug.

3.2 The SARJ commanding bug

In this section, we describe the commands involved in commanding the Port SARJ and under which conditions these commands might fail. For that, we focus our attention to the step that commands the SARJ to checkout (i.e., to stop rotating). The commands are listed under Step 9.1 of the 1.252 procedure as shown in Figure 2. These steps are interesting


9.1 Commanding SARJ to Checkout

Port SARJ

sel Operation

Port SARJ Operation

NOTE

1. Sending the Check Out command will cause the SARJ to stop rotating. If possible, wait for eclipse.
2. It may take several minutes for the SARJ to stop rotation and for the SARJ Busy flag to clear. Commands will be rejected by the SARJ until the SARJ Busy flag clears.

'SARJ Mode'

cmd Check Out

Verify SARJ Mode – Checkout
Verify SARJ Busy – blank

'String 1'
'Motor'

cmd Select – None (Verify – None)
cmd String Mode – Monitor (Verify – Monitor)

Figure 2: Steps for commanding the Port SRAJ to checkout.

(from a V&V point of view) for several reasons. First, "cmd Check Out" is followed by two "Verify" instructions, which are there to ensure that what is described in the note section is true; i.e., that the SARJ has stopped rotating (hence the Busy flag is set to blank) and the telemetry indicates that the SARJ is now in Checkout mode. If the second Verify command (i.e., "Verify SARJ busy") was not present, controllers could be tempted to send other commands, which would fail silently (see next paragraph for more details) if the SARJ was still rotating. So, the second Verify command tells the controller to verify that the SARJ has indeed stopped completely before they proceed with the rest of the procedure. Note that "Verify" instructions are usually performed by the controller, whose only feedback is provided by the telemetry coming back from the ISS. The controller does not see the SARJ, and therefore, the SARJ busy flag is the only indicator that the SARJ has indeed stopped.

Now, Procedure 1.252 has already been verified and validated. Therefore, it does not contain any error. To illustrate our V&V technique we decided to modify Procedure 1.252 and introduce an error by omitting the "Verify SARJ Busy - blank" instruction. This could make the controller to send the next command (i.e., "cmd Select - None") without being aware that the SARJ is still rotating since Note 2 in Figure 2 indicates that it may take several minutes for the SARJ to stop rotating. The note also says that any command sent while the SARJ is rotating would be rejected by the SARJ, which means that it would be received but ignored and then dropped by the SARJ. Therefore, by omitting the "Verify SARJ Busy - blank" instruction, we have introduced the possibility of issuing and dropping (silently) commands.

It is fairly obvious that this bug may not be caught during testing. In order to test for this problem, the SARJ must be correctly simulated, and possible mismatches between the procedure and actual behavior must be covered during testing. If the delay used to simulate the time needed for the SARJ to stop rotating is too short, no command will be dropped. However, if the delay is long, then commands may get dropped. Furthermore, testing such a problem as this is cumbersome with forward simulation. This points to the fact that procedure V&V should not rely exclusively on simulation or testing, but also include techniques such as model checking as we will describe in a subsequent section.

3.3 The ISIS system

International Space Station Systems Integrated Simulation (ISIS) is a desktop high fidelity training simulator for the International Space Station program. ISIS runs the actual unmodified ISS flight software, system models and the astronaut command and control interface in an open system design architecture that can be programmed to integrate with other software models.

The United Space Alliance (USA) Advanced Technology group develops ISIS. Since it was originally envisioned, the ISIS usage has evolved to include ground operator system and display training, flight software modification testing and a realistic test bed for exploration automation technology research and development.

The configuration of the ISIS that we use for our project consists of two high-end HP workstation class machines, one ISS Portable Computer System (PCS) laptop and one A4O development laptop. All four computers are connected on a separate network. Figure 3 shows the configuration layout of ISIS system for the project. The two high-end HP workstations are the basic workhorses to simulate the ISS vehicle and ISS flight software. The first HP workstation's primary function of this workstation is to execute the Ada Space Station Training Facility (SSTF) Host Partitions soft-

ware, which models the ISS environment and systems. The other function is to run the ISIS Information Sharing Protocol (ISP) server for distributing telemetry data to the ISIS network via ISP client software. The second HP workstation executes the ISS MDM Development Environment (MADE) application, which in term runs the unmodified version of the MDM Flight Software Configuration Software Configuration Items (CSCI). The MDM flight software is integrated with SSTF Host via Space Station Training Facility (SSTF) Portal application environment and system models. Also, a Virtual Command Server (VCS) runs on the server to allow external applications to connect and command the ISS MDM flight software. Next, the PCS laptop, which is the actual command and control workstation for the onboard crew, is used to monitor telemetry and issue commands to the ISS vehicle. Finally, the A4O development laptop is used to run our dynamic verification software to verify the validity of the procedures.

ISIS provides a key functionality to verify and validate software changes and procedures. A key design decision was made to reuse the existing software such as the MADE and SSTF Models from the ISS development program and re-package the software for operational use. As a consequence of this decision, ISIS is very difficult to install, configure and run. Furthermore, it is intended to replicate the ISS for training a person, not as a fully reversible model of the software suitable for model checking.

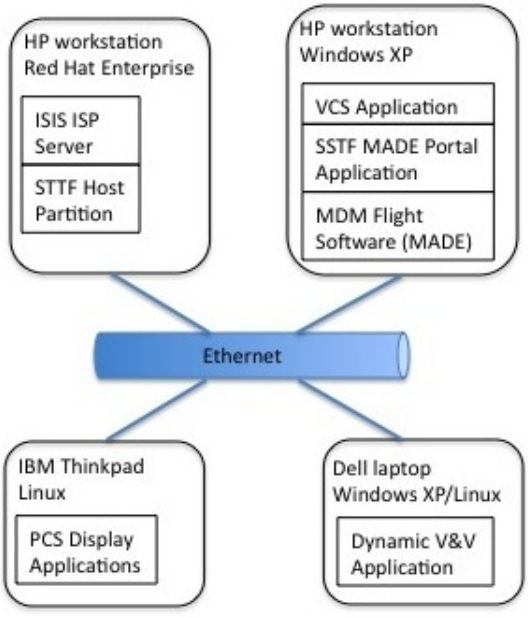


Figure 3: The ISIS system.

4. A4O Technology Development

The Automation for Operations Project (A4O) develops advanced technology to enable the operation of Constellation Program mission elements such as the Orion spacecraft, the International Space Station (ISS), and robotic assets that will support human exploration of the Moon (Frank 2008a; 2008b). The projects technology developments include:

- Languages for specifying standard operating procedures in electronic form, which allows fine grained and gradual procedure automation (PRL and PLEXIL, see below).
- A software infrastructure that enables seamless tool interoperability among all mission operations software components

4.1 PRL

PRL (Kortenkamp, Bonasso, and Schreckenghost 2008) is an electronic procedure representation language, which enables the authoring of human-understandable procedures, e.g., procedures involving astronauts on the ISS. The goal of PRL is to build upon a user-friendly display format and extend it with the following elements needed for autonomous systems.

- Meta data provides names, context, version, etc. for procedure
- Control data provides logical control and safety conditions
- Steps and nodes structure procedure for human readability
- Instructions specify instructions, commands, etc

PRL is a modular language which allows for a compositional approach to procedure authoring. Thus small procedures, specific to some sub-activities, can be composed into a larger procedure which describes a full activity. PRL is based on an XML-schema. PRL procedures can be executed by translating them into PLEXIL plans and using any executive accepting PLEXIL. Figure 4 shows an example of the XML representation of a PRL if-then statement.

An innovation of PRL is to tightly integrate commands and telemetry into the procedure description (which is not done today). Note that this is important for automated V&V since commands and telemetry will correspond to events in the models used for model checking. Having this information integrated with the procedure simplifies the creation of these models.

4.2 PLEXIL

Space mission operations require flexible, efficient and reliable plan execution. In typical operations command sequences (which are a simple subset of general executable plans) are generated on the ground, either manually or with assistance from automated planning, and sent to the spacecraft. For more advanced operations more expressive executable plans may be used; the plans might also be generated automatically on board the spacecraft. In all these cases, the executable plans are received by a software system that executes the plan. This software system, often called an executive, must ensure that the execution of the commands and response of the fault protection system conforms to preplanned behavior. PLEXIL is designed specifically for flexible and reliable command execution. It is designed to be portable, lightweight, predictable, and verifiable, and at the

```
<IfThen>
 <EQ>
  <DataReference source="cached">
   <ID>CW_5520</ID>
  </DataReference>
  <Constant>
    <IntegerValue>0</IntegerValue>
  </Constant>
 </EQ>
 <InstructionBlock>
  <Instruction instructionID="inst3">
   <AutomationData/>
   <ManualInstruction>
     <Text> This command clears all
associated data in the MDM. Do not
execute if DDCU 50 Hz or other C&W
data needs to be dumped.
      </Text>
   </ManualInstruction>
```

Figure 4: PRL example: a simple if-them statement.

same time, the language does not sacrifice expressiveness (Verma et al. 2005; 2007).

In order for the PLEXIL language to be useable, an execution system is required to interpret it. The Universal Executive is an execution system designed to facilitate reuse and inter-operability of execution and planning frameworks. As part of this project, we have developed an automatic translation from PRL to PLEXIL. PRL is used for authoring and PLEXIL for executing the procedure. Figure 5 shows a PLEXIL example for an ISS procedure.

4.3 PRIDE

The Procedure Integrated Development Environment (PRIDE) is a procedure authoring tool being developed in the A4O project to make developing PRL easier for flight controllers. PRIDE has a graphical interface (developed with Eclipse) that offers the following services:

- A connection to a database containing information about available commands and telemetry for a given spacecraft
- Automatic expansion into PRL instructions
- Static checks of PRL files for assessing PRL's syntactical and semantical validity, ensuring that procedures are invoked correctly, consulting the system-representation database to validate usage constraints, and, verifying the consistency of logical conditions using a constraint solver.

PRIDE also offers a sub-menu that triggers the use of the Java PathFinder model checker and to a finite-state machine simulator.

```
...
NodeId: Step_1
    Boolean this_step_failed=0
    InvariantCondition: this_step_failed=0
    PostCondition: LookupNow(LAB_UOP6,
state)==INACTIVE
    NodeList: {
     Sequence: {
      Node: { Command: power_down("UOP6",
"equipment") }
      Node: {
       NodeId: Step_1_press_check_and_retry
       Integer cnt=0;
                    PostCondition:
LookupNow(switch,
power_out)=="NOT_DEPRESSED"
      RepeatUntilCondition: AND(OR(cnt>=2,
LookupNow(ENABLE)=="illuminated"),
                    OR(cnt==3,
LookupNow(OK)=="illuminated"))
      Sequence: {
       Node: { Command: press(switch,
power_out, LAB_UOP6)
       Node: { Assignment: cnt=cnt+1} } }
     if LookupNow(LAB_UOP_6,
disconnect_necessary)==1
                    Node: {Command:
disconnect(equipment, LAB_UOP6) }
...
```

Figure 5: PLEXIL example: extract of an ISS powerdown procedure.

4.4 FSM

FSM is a simulation (of the controlled system, e.g., the ISS flight software) tool based on finite-state machines (hence, the name FSM). The goal is to enable a manual walkthrough when editing a procedure. The FSM runs a finite-state machine (following Harel's state chart formalism and semantics (Harel 1987)) description of the ISS flight software and interacts with a human posing as a potential controller. In essence, FSM can be seen as a low-fidelity simulator that can be used during procedure authoring (as opposed to highly-contended high-fidelity testbeds). The first and most basic V&V intended use of FSM is to check that all necessary started conditions are specified in a procedure. If some of these conditions are missing, the simulation will block or follow an obviously wrong path.

Note that FSM is a simulation environment. Each simulation is the equivalent of a test run. It follows one particular path amongst the system transitions and do not attempt to explore all possible paths. A full and exhaustive exploration can only be done using a technique such as model checking.

FSMs are created manually using the Unimod tool (Unimod). We used some of the engineering documents used for training the flight controllers; they describe the behavior of the EPS devices. We also validated our models by care-

ful discussion with the flight controllers. The models do not represent the dynamics of the system, but the perceived (by controllers) states and transitions that can be taken by the system. So, if a procedure is also modeled as a finite-state machine, it can be coupled with the FSM and fed to a model checker such as Java PathFinder (JPF) to verify properties.

4.5 JPF

The original JPF is a model checker for Java software (Brat et al. 2000a; 2000b). Since its creation it has been extended a number of ways, one of which allows the model checking of finite-state machines (such as UML State Charts) (Mehlitz 2008). JPF state chart framework is a combination of (state chart) libraries that are based on a unique translation scheme for UML state charts, together with generic applications (test drivers) to execute these state charts. The framework supports both testing and model checking, using the same scripting language for environment specification.

Our goal in using model checking is as follows:

- overcome the limitations of testing (simulation) by enabling exploration of the full state space,
- enable the verification of complex properties (which could come from formal requirements or from flight rules),
- automate the V&V process.

It is clear that we can achieve this goal only by having models of all the interacting components, i.e., the controlled system, the procedure, and the execution context.

5. Running the Analysis

This section describes how we integrated the different A4O element to implement model checking of PRL procedure and made it available in the PRIDE authoring environment. Figure 6 shows how the PRL procedure is first translated into a PLEXIL plan, which is then translated into a (Java-represented) state chart model that can be understood by JPF. Moreover, the figure shows how Unimod models (which are UML state charts models) are used to build the FSM simulator and are also translated into JPF-understandable models. These two models are then fed to JPF with some safety property for analysis. When it finds a property violation, JPF spits out an error trace, which (after some massaging) can be re-played in the FSM simulator.

The next sub-sections describe some of the results we have obtained and challenges we have faced in the course of our experiment.

5.1 Building the FSM

Our first big challenge is to create the FSM. Some NASA projects, like Orion, mandate a model-based development process, which can possibly produce the sort of models we need for model checking. However, it is not the case for the ISS. Given that access to the ISS flight software is impossible and reverse-engineering the system is very difficult and costly, we did the next best thing. Our FSM was created by inferring system behavior from the training manuals used by the controllers and focusing on the parts that are relevant

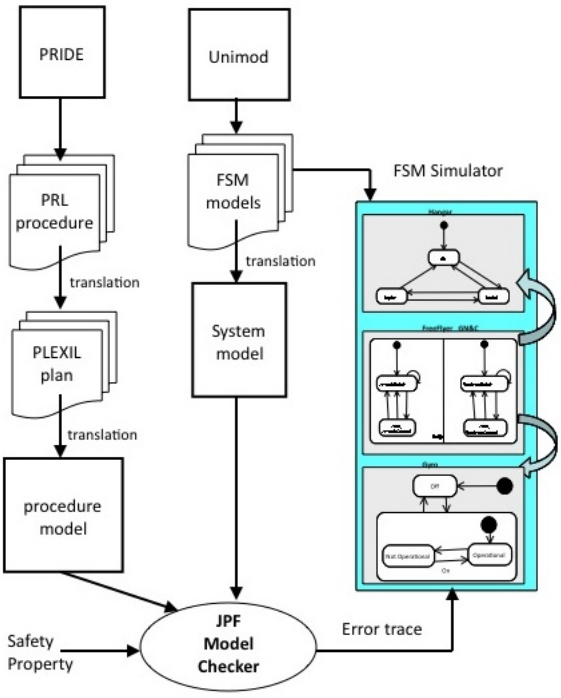


Figure 6: The set-up for model checking in A4O.

to procedure 1.252: we validated the models by sitting with ISIS and executing commands in the procedure as specified to determine what system state was. Note that this process was only able to give us nominal and a subset of the offnominal behavior. For example, as we can see in Figure 2, the note in step 9.1 of Procedure 1.252 indicates the behavior of the SARJ when commands are sent while the SARJ is busy, but not what other outcomes of the Check Out command might be. We also validated our FSM by asking flight controllers for help.

Obviously, this is problematic from a V&V point of view, but we deemed it acceptable since our goal was to create a proof of concept. Moreover, if our technique is to be deployed, it will be on one of the new systems (e.g., Orion) and we would insist on the necessity to develop the FSMs in parallel with the software (and update the FSMs each time the software is updated). An alternative way is to develop the FSM at the time high-fidelity simulators are developed.

5.2 Verifying the FSM and the procedure

As described above, the procedure and the FSM are not in the right format for JPF. They first have to be translated into the State Chart extension of JPF (i.e. Java programs corresponding to the finite-state machines).

PLEXIL translation to JPF statecharts In A4O, we assume that the starting point is not a procedure written in an MS Word document, but a PRL procedure authored with PRIDE. This procedure goes through two translation passes before being amenable for model checking.

1. The first translation transforms a PRL procedure into a

PLEXIL plan. The translator was designed by the developer of PLEXIL to represent the PRL procedure faithfully. Care is taken to keep track of the correspondence between the PRL instructions and the PLEXIL instructions, which means that if can produce a counter-example (or trace) in PLEXIL we can translate it back into a PRL trace.

2. The second translation transforms a PLEXIL plan into a finite-state machine expressed as a Java program. This two-pass translation scheme allows to re-use our framework for PLEXIL programs, even if they have not been created from PRL procedures.

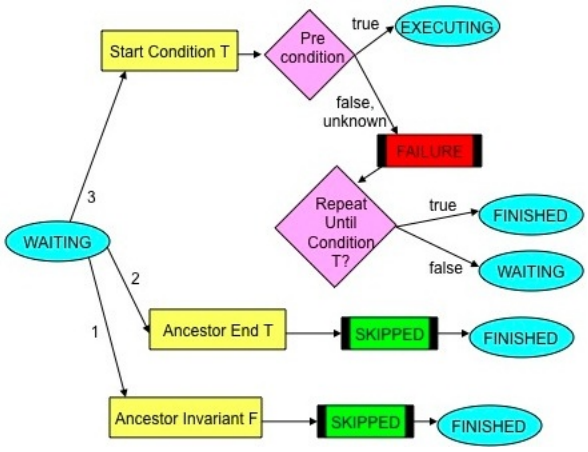


Figure 7: An example of a Plexil semantics model.

Now, PLEXIL was designed with verification in mind. So, formal behavioral semantics were developed for every PLEXIL instructions (or more specifically node) (Dowek, Muñoz, and Păsăreanu 2007; 2008). These semantics are represented in the form of finite-state machines (see Figure 7). This allowed us to develop faithful models of PLEXIL plans by instantiating the behavioral models of every PLEXIL node according to its use in the PLEXIL plan to be translated. Figure 7 describes the behavioral model for a node is status waiting. The finite-state models for the generic PLEXIL nodes are hardcoded in a Java library. This allows the act of translating to be reduced to generating the transitions between the nodes and describing the characterization of each node used in the plan.

This technique has several advantages. First, we produce easily readable models and keep the PLEXIL models fairly close to their original PLEXIL plans. Second, if the semantics of PLEXIL are updated, we do not need to change our translator; we just need to update the Java representation of the semantics in our Java library. We therefore achieve a separation of concerns between the language used to represent the plan and the plan itself.

FSM translation to JPF statecharts The FSM is created using Unimod, which follows the semantics of UML statecharts. They are therefore directly amenable to a translation into JPF statecharts. The translation is pretty simple, and, we therefore describe only the challenging parts.

States and transitions are preserved in the translation, but we do not preserve any specialization code added to the finite-state machine. Thus, we are not able to translate parts related to timing (since they are not strictly speaking part of standard statecharts). Unfortunately, models for the SARJ need to include some timing information to represent for example the time needed by the SARJ to stop rotating once a ”stop SARJ” command has been issued. This timing information had to be added by hand in our JPF model.

Scalability As with all model checking exercises, scalability is an issue. On one hand, procedures are relatively small and sequential. They may have a few branching conditions, and in some cases, loops; but, by far and large, they are relatively simple. Therefore the procedure model is essentially one state machine with lots of states. Its size corresponds to the product of the number of PLEXIL nodes (somewhat equivalent to the number of procedure instructions) and the average number of states in the semantic models for each PLEXIL node. On the other hand, the FSM (the model for the EPS) generates a fairly large state space. The FSM consists of many (more than eighty in our case) parallel state machines which synchronize on internal and external events. The composition of the procedure model and the FSM result in a large state space.

Our JPF-based verifier is coded as an Eclipse plugin. Our first intention was to keep the translations passes and the model checking activity on one process (the plugin process) by calling JPF through a direct method call. Unfortunately, we were not able to associate enough memory (we need at least 800 MB) with the plugin to allow model checking to complete. Therefore, JPF is called on a separate process, which allows us to control exactly how much memory we make available to the model checker.

For our experiments, we chose a MacBook Pro laptop with a 2.4 GHz Intel Core 2 Duo processor and 4 GB of memory. It took JPF one or two minutes to find the bug we planted in the procedure model. Full exploration of the state space of models without any bug took about 45 minutes.

Bugs in the JPF code We also encountered problems defining the FSM. In our first attempt, the FSM was not able to update the ”previous cmd” variable before the next command was sent. This bug in the FSM led to a hitmiss situation in showing the bug targeted by JPF. When the checkout command was received by the model, it should have immediately set the ”SARJ to busy” flag indicating that other commands should be ignored until the flag becomes ”not busy” again. On receiving a checkout command the FSM would schedule the SARJ to get busy on a separate thread, which resulted in a window of opportunity for the next command to sneak in before the SARJ got to the busy mode.

The transitions showing the bug are as follows:

- Checkout command: schedule busy command and returns
- The next command comes in. But the SARJ may not be busy (since it was scheduled) so the transition gets done.
- Now busy to SARJ mode kicks in, and after some time, transitions to not busy

The fix consisted of, on receiving a checkout command, returning back to the caller only after the SARJ is set to busy so that there is no chance for the next command to sneak in. It resulted in the following transitions:

- Check out command returns after SARJ set to busy
- The next command comes in, but cannot transition because SARJ mode is busy.

The details of this problem and its fix may seem opaque to the reader, but it shows that building the FSM is not a trivial exercise. It is at least as difficult as designing the synchronization of the real system, which points to the usefulness of these models in a design phase. They allow to catch tricky synchronization bugs when designing complex systems consisting of interacting components.

6. Lessons learned and Future Work

We have presented a verification technique for a complete authoring framework for procedures at NASA. Our framework allows for the authoring of PRL procedures, their translation into PLEXIL plans for execution, and their verification using simulation (FSM) and model checking (via Java Path Finder) in the PRIDE Eclipse plugin. In this paper we insist on the verification part, especially the one using model checking; PRL procedures are translated into finite-state machine amenable to model checking when coupled with a finite-state machine for the controlled system. We have described how we demonstrated this technology to flight controllers for the International Space Station.

We have been able to learn several lessons from this experience. The model checking approach to V&V of procedures imposes significant costs in the development of the models/simulations as well as the development of the model checking software. The expected benefits are in reduction of time to verify procedures, reduction in costs due to contention of big expensive facilities, and reduction in errors in flight (with their attendant costs). Finally, our process includes several translation passes. In practice, before any deployment, each translator should be validated.

Our example shows that model checking can find subtle synchronization bugs which can be easily missed by testing or simulations. When we show our work to flight controllers, they all concur that it is a useful technology which deserves to be explored further. They also pointed out the fact that it needs to be hardened before it can be deployed.

Our future work will focus on the scalability aspects. We plan to investigate how we can use compositional verification to fight scalability. It is fairly easy to identify the different components in the models of the controlled systems. Moreover, not all of these components are involved in the verification of a property. Therefore, we should be able to take advantage of the assumption learning algorithm developed in to reduce the size of the model checking problem (Barringer, Giannakopoulou, and Pasareanu 2004).

References

Barringer, H.; Giannakopoulou, D.; and Pasareanu, C. 2004. Component verification with automatically generated assumptions. *Journal of Automated Software Engineering* 11:93–98.

Brat, G.; Havelund, K.; Park, S.; and Visser, W. 2000a. Model checking programs. In *Proceedings of ASE'00: 15th IEEE Int'l Conf. on Automated Software Engineering*, 3–11.

Brat, G.; Park, S.; Havelund, K.; and Visser, W. 2000b. Java pathfinder-second generation of a java model checker. In *Post-CAV Workshop on Advances in Verification*.

Brat, G.; Gheorghiu, M.; Giannakopoulou, D.; and Pasareanu, C. 2008. Verification of plans and procedures. In *Proceedings of IEEE Aerospace 2008*.

Connors, E.; Muñoz, C.; Schnur, C.; and Siminiceanu, R. 2009. Static verification of spacecraft procedures. In *Proceedings of the AIAA Infotech@Aerospace Conference 2009, AIAA 2009-2033*.

Dowek, G.; Muñoz, C.; and Păsăreanu. 2007. A formal analysis framework for PLEXIL. In *Proceedings of 3rd Workshop on Planning and Plan Execution for Real-World Systems*.

Dowek, G.; Muñoz, C.; and Păsăreanu, C. 2008. A small-step semantics of PLEXIL. Technical Report 2008-11, National Institute of Aerospace, Hampton, VA.

Frank, J. 2008a. Automation for operations. In *Proceedings of the AIAA Space 2008 Conference and Exposition*.

Frank, J. 2008b. Costs and benefits of automation for lunar surface operations: Preliminary results. In *Proceedings of the AIAA Space 2008 Conference and Exposition*.

Harel, D. 1987. Statecharts: A visual formalism for complex systems. *Science of Computer Programming* 8(3):231274.

Kortenkamp, D.; Bonasso, R. P.; and Schreckenghost, D. 2008. A procedure representation language for human spaceflight operations. In *Proceedings of the 9th International Symposium on Artificial Intelligence, Robotics and Automation in Space (i-SAIRAS-08)*.

Mehlitz, P. C. 2008. Trust your model - verifying aerospace system models with javał pathfinder. In *Proceedings of IEEE Aerospace Conf. '08*.

Unimod. http://unimod.sourceforge.net/.

Verma, V.; Estlin, T.; Jnsson, A.; Pasareanu, C.; Simmons, R.; and Tso, K. 2005. Plan execution interchange language (plexil) for executable plans and command sequences. In *International Symposium on Artificial Intelligence, Robotics and Automation in Space (iSAIRAS)*.

Verma, V.; Baskaran, V.; Utz, H.; and Fry, C. 2007. Demonstration of robust execution on a nasa lunar rover testbed. In *International Symposium on Artificial Intelligence, Robotics and Automation in Space (iSAIRAS)*.