Verifying Equivalence of Procedures in Different Languages: Preliminary Results

David J. Musliner, Michael J. S. Pelican, Peter J. Schlette
Smart Information Flow Technologies (SIFT)
{dmusliner, mpelican, pschlette}@sift.info

Abstract

NASA operates manned spacecraft according to rigorously-defined procedures, some of which can be executed both automatically and manually. Unfortunately, the operating procedures for different pieces of equipment and different execution modes (onboard/offboard, manual/automatic) may be written in entirely different languages. In this paper, we describe an approach to verifying that the underlying semantics of these diverse procedure representations are the same; that is, we want to verify that the two different procedures "do the same thing." We accomplish this verification by translating each procedural language into a common verification language, Promela, and using the SPIN verification tool to confirm that the procedures behave identically when given identical inputs. This paper describes the patterns of translation into Promela and explores the scalability of this approach. More generally, this work represents first steps towards defining a rigorous semantics for both PRL and SCL, which can be used for future verification and validation efforts supporting high-confidence software for manned spaceflight.

Introduction

Like any organization with complex, hazardous equipment, NASA operates manned spacecraft according to rigorously-defined standard operating procedures (SOPs). These procedures carefully define what steps should be taken to accomplish a wide variety of goals, including changing the operating modes of equipment, starting up and shutting down components or subsystems, and conducting various joint machine/crew activities such as spacewalks. The procedures may include sequences of primitive commands that change the state of onboard equipment, tests that verify certain state changes via telemetry, or branching logic that varies the procedural behavior depending on the state of the spacecraft or other environmental factors. Some procedures can be executed both automatically and manually. Spacecraft operating procedures are very carefully reviewed to ensure correctness (Frank 2008).

As the space program has evolved over time, the operating procedures for different pieces of equipment and different execution modes (onboard/offboard, manual/automatic) have been written in entirely different languages. For example, some components of the new Orion crew vehicle will be supplied with automatic procedures written in SCL, the Spacecraft Command Language (Buckley & Vangaasbeck 1994), while backup manual procedures may be developed in PRL, the Procedure Representation Language (Kortenkamp, Bonasso, & Schreckenghost 2008; 2007). PRL is an evolving language being designed to capture both manual and automatically-executable procedures. In contrast, SCL is an off-the-shelf product designed for fully-automated procedures or, more generally, fully-autonomous control systems involving both procedural and event-triggered, rule-based elements.

PRL and SCL features differ significantly, to the extent that an expert would have difficulty comparing even small procedures. For example, an SCL developer could capture a command's safety conditions in one or more "constraints," rules that are checked *after* a command has been selected by a script, but *before* the command has been sent to the spacecraft system. In PRL, it would be more natural to represent such conditions as "VerifyInstruction" steps before every use of the command. To compare the procedure implementations statically, a tool must identify all relevant constraints that would be active in the execution context and determine whether they are checking the same preconditions that the PRL VerifyInstruction checks.

Our research focuses on proving that specific PRL and SCL procedures have the same underlying execution semantics, so that NASA can be assured that if an automatic SCL program cannot be executed, a backup manual procedure in PRL will be equivalent and safe. Our approach generalizes to comparisons between other procedure representation languages, other NASA programs, and other industries.

In the following sections, we first overview PRL and SCL, showing how a simple example procedure fragment might be encoded in each language. We then describe our approach to verifying that these different encodings produce the same behavior. Our approach involves translating each procedural language into a common verification language, Promela, and confirming that the procedures behave identically when given identical inputs (Heimdahl, Choi, & Whalen 2002), using the SPIN verification tool (Holzmann 1997; 2005). We illustrate the respective SCL and PRL translations, present a preliminary evaluation of scalability,

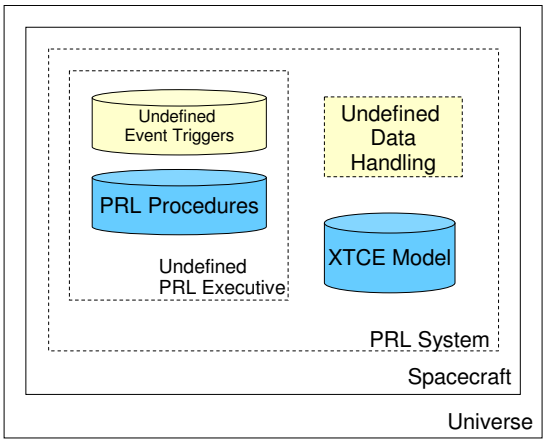

Figure 1: PRL semantics depend in part on the specification of future executives.

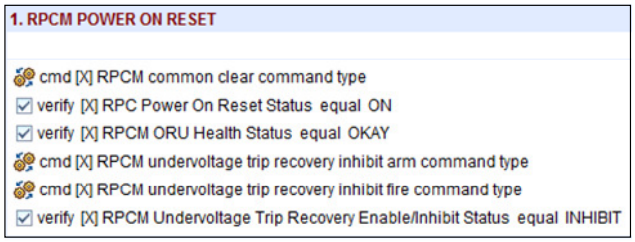

Figure 2: The PRIDE integrated development environment supports relatively painless PRL editing and visualization.

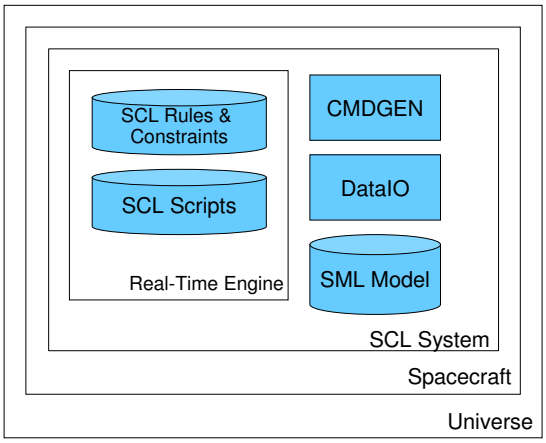

Figure 3: SCL's flight-proven implementation defines its execution semantics.

and discuss limitations of the approach.

While our translation patterns are not yet fully automated, our preliminary results indicate that this language-translation-and-model-checking approach is both feasible and scalable to the desired level of fidelity. This work represents first steps towards defining a rigorous semantics for both PRL and SCL, which can be used for future verification and validation efforts supporting high-confidence software for manned spaceflight. We conclude with observations on the remaining challenges in increasing the fidelity of the verification process and addressing the resulting scalability problems.

The Procedure Representation Language (PRL)

PRL is a still-evolving language designed to capture procedures that may be executed either by automation or by humans (Kortenkamp, Bonasso, & Schreckenghost 2008). Defined by an XML schema, PRL allows a programmer to construct top-level *procedures* that are decomposed into *steps*, each of which may execute blocks of primitive *instructions* and control statements. Instructions can include spacecraft commands, tests of telemetry values, calls to other procedures, and `wait` instructions that block for some time or until a boolean expression becomes true. Instructions may be specified as only executable by manual means, or may include *automation data* to help describe how an (as yet undefined) automatic PRL executive should run the procedures. Automation data can include the expected `StartConditions` that must be true to enable the procedure, `InvariantConditions` that must remain true during execution (or the procedure fails), and `EndConditions` that wait until they are true to allow the procedure to end. PRL commands refer to spacecraft components by their unique identifiers as defined in an XML Telemetric & Command Exchange (XTCE) model (Object Management Group 2004).

PRL is being developed to support a gradual transition from fully-manual, textual procedures towards automation. As a result, some elements of a fully-automatic PRL system are not yet defined, including a complete formal semantics for the language and an automatic PRL executive. However, initial steps towards both have been taken in an experimental translation of PRL into the Universal Executive's PLEXIL language (Kortenkamp, Bonasso, & Schreckenghost 2008).

An Eclipse-based development environment, PRIDE, has been developed to simplify procedure authoring (Izygon, Kortenkamp, & Molin 2008). The PRL fragment in Figure 2 is drawn from a real ISS procedure that configures an electrical component by issuing a series of commands and verifying assorted telemetry values.

The Spacecraft Command Language (SCL)

As illustrated in Figure 3, Spacecraft Command Language (SCL) is a suite of tools for building and deploying automated control systems for spacecraft. SCL has flown on many unmanned space missions, including FUSE, EO-1, and TacSat-2. SCL's Real Time Engine (RTE) executes programs defined in a language that combines procedural scripting with event-triggered rules and constraints. The RTE interfaces with the spacecraft sensors through the DataIO module, which implements data sampling rates, smoothing, and other filtering operations defined by a data model written in Spacecraft Markup Language (SML). Outgoing commands are sent via the CMDGEN component to spacecraft

```
rule EnsureHealth
 subsystem Health
 priority 28
 activation yes
 if X.RPCMORUHealthStatus != OKAY then
        exit
 end if
end EnsureHealth

// Main returns true if completes OK.
function main(X)
 IssueCommand("X.RPCMCommonClearCmdType")
 if (!VerifyEquality(X.RPCPowerOnResetStatus,
         ON)) then
        return false
 end if
 // More steps omitted...
 return true
endfunction main

function IssueCommand(subject)
 SendCommand(subject)
endfunction IssueCommand

function VerifyEquality(subject, targetValue)
 if (dereference(subject) == targetValue)
        then return true
 else
        return false
 end
endfunction VerifyEquality
```

Figure 4: SCL combines rules and hierarchical function calls, challenging the representational power of automatic verification systems.

actuators.

The SCL language allows behavior descriptions in three different program elements: rules, constraints, and scripts. Rules can be triggered by external events (including commands from crew and ground) and updates to internal data. Constraints check safety conditions or validity criteria before spacecraft commands are issued. Scripts are written in a fairly standard procedural language that includes function calls and both local and global variables. SCL's RTE schedules rule evaluation using a priority-based agenda, and also manages parallel execution of scripts at fixed priorities. Taken together, these elements of SCL provide an extremely flexible programming environment and the possibility of creating control flows that would be very difficult to verify. In practice, experienced SCL developers use a set of well-understood design patterns, such as using individual event-triggered rules to activate scripts (which provide sequencing logic), rather than Prolog style rule chaining.

As an implemented and deployed system, SCL has a concrete *de facto* execution semantics, although some details are not clearly described in the publicly available documents (*e.g.*, the precise algorithm of the RTE scheduler is unclear). To make the verification problem tractable, the use of parallel interacting scripts must be limited.

To illustrate some of the representational power of the SCL language, we have used both rules and scripts to encode a portion of the same example ISS procedure. Figure 4 shows our approximate SCL translation of the first three steps of the partial procedure shown in Figure 2. The `main` function in our SCL encoding includes two steps of the procedure: issuing a single command and verifying its result. For the third step, another state verification, we used a simple rule that monitors a property and takes action when it enters a given state. Here, we issue an `exit` command whenever X's ORU health enters any state other than `OKAY`. Such rules— along with constraints, which function in a similar manner— have a number of complexities that must be handled. For example, each rule is assigned to a subsystem, and every rule or constraint within a subsystem can be activated or deactivated using a single SCL command. The interactions between rules, constraints, and scripts are mediated by their priority levels and the aforementioned scheduler algorithm, which makes precise modeling even more difficult.

Approach

To verify the functional equivalence of spacecraft procedures implemented in SCL and PRL, we translate the native implementations into a third language, Promela. We then use the SPIN model checking tool (Holzmann 1997; 2005) to generate a complete set of possible input sequences and feed them to the Promela procedure models, collecting each output command stream. The procedure implementations are equivalent if, for every input sequence, the command streams match in every place. In other words, we check for observable execution trace equivalence, where the only observables are the commands sent to the spacecraft.

SPIN

We use the SPIN model checker to exhaustively verify Promela models of procedure implementations and inputs. SPIN takes Promela as input and generates a set of C language files unique to the input given. Then the user compiles these into an executable which performs exhaustive verification. This verification detects cycles, assertion violations embedded into Promela, blocks of unreachable code, and so on. If a branch of the verification violates an assertion or encounters some other error, SPIN creates a ".trail" file which contains information on the state of the model up until the time of the error. This feature provides a counter-example that shows under what circumstances the two encoded procedures behave differently.

In Promela, a statement is not necessarily executable; if a condition evaluates to false, the statement is ignored until the condition becomes true. Some commands, such as assignment, are always executable. Others, such as `false` are never executable. Many depend on the state of the system, such as boolean conditions or a request for a message from a channel which may or may not be empty. If an unexecutable statement is reached in a `if` or `do` block, a different statement which is executable will be run instead. If Promela becomes "stuck" at a point where no possible branch has a

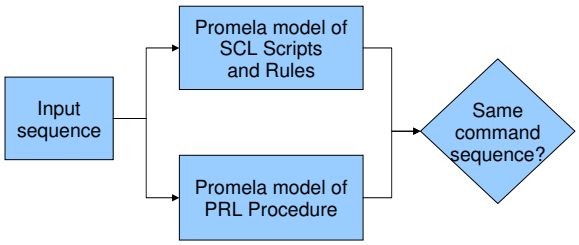

Figure 5: SPIN generates possible input sequences and feeds them to models of PRL procedures and SCL scripts and rules. The implementations are considered equivalent only if the sequences of command outputs are the same.

statement that is executable, then execution ends. The `->` operator is used to emphasize the first statement in a conditional's role as a guard. The language supports analogs to most basic imperative language features: `if`, `do` (a loop), labels and `goto`s, and calling functions with parameters passed. A key difference is that if more than one leading statement is executable simultaneously in an `if` or `do`, the statement block that will be executed first is selected nondeterministically. This feature enables one to easily select a value for an input from a domain of potential values, or represent other sources of nondeterminism. In an exhaustive verification, all possibilities will be explored.

Promela provides message channels that send and receive data between processes. The most basic and frequently used channel operators are `!` and `?`, which respectively send and receive messages. To share data between two processes, it is necessary to pass the same channel as a parameter to both processes. Channels can also be tested with `empty` and `full`, as well as `len`, which will return a positive number (implicitly `true`) if there are any messages in the queue, and false otherwise. These can be used to take alternate action if a queue is full or to receive messages until a channel is empty. In addition to the basic send and receive, the `!!` and `??` operators serve a more specialized purpose. `!!` is particularly relevant to our efforts and will be discussed later.

Modeling system state

To test procedure equivalence, we need a framework in which to provide identical sets of inputs to models of the procedures and then gather their output in a way that allows them to be compared. For this purpose, each procedure will take two channels as inputs. The "state" channel provides the initial state of the procedure and system configuration (including values selected for state variables) and will be filled with the end state at end of execution. The "command log" channel will be passed to the procedure with no messages in it. This channel will be used to create a log of all the spacecraft commands issued by the procedure.

As mentioned above, modeling nondeterminism in Promela is trivial. Fig. 6 demonstrates a small random initialization process, by which both state channels will be filled with the same initial state.

Just as critical as supplying an initial state is verifying

```
#define RPC_POWER_ON_RESET 0
#define RPCM_ORU_HEALTH_STATUS 1
#define ON 2
#define OFF 3
/* ... */
chan prlState = [NUM_INPUTS] of {byte,short};
chan sclState = [NUM_INPUTS] of {byte,short};

/* Set initial state for the pump status */
if
::  prlState!RPC_POWER_ON_RESET,ON ->
    sclState!RPC_POWER_ON_RESET,ON
::  prlState!RPC_POWER_ON_RESET,OFF ->
    sclState!RPC_POWER_ON_RESET,OFF
fi;
/* .... */
```

Figure 6: Promela's nondeterministic `if` generates sequences of input streams that cover the range of possible inputs (*i.e.*, system state) and copies them to send to both procedure models.

```
/* Model execution precedes this... */
int prlType, prlValue, sclType, sclValue;
bool success = false;
do
::      (len(prlLog) && len(sclLog)) ->
        prlLog?prlValue;
        sclLog?sclValue;
        if
        ::      (prlValue == sclValue) ->
                printf("Models agree\n");
        ::      else -> /* mismatch! */
                printf("Models disagree!\n");
                success = false;
                break /* Abort loop */
        fi;

/* When at least one queue is dry */
::      else ->
        /* Failure if one queue non-empty */
        if
        ::      len(prlLog) ||
                        len(sclLog)) ->
                success = false
        ::      else -> success = true
        fi;
        break
od;

/* Omitted similar check of state */
assert(success)
```

Figure 7: Channel contents must agree in both value and size in order to be considered equivalent.

the end state. Our current approach is quite simple: the top message in each procedure's command log is pulled off, and the two are compared. If the two are equal, this process is repeated. However, if the two stacks have different numbers of messages or if two messages don't contain the same data, then an appropriate error message is printed and an assertion is violated. The same process is then carried out on the state of the two models. Fig. 7 shows a summary of this process in Promela.

Having established a method for testing the equivalence of two models, we now discuss the process of constructing Promela models that emulate the behavior of PRL and SCL procedures. Because this project is still in its early stages, generation of models for these procedures has thus far been accomplished by hand. However, we have worked with the aim of demonstrating a systematic conversion, and have already taken small steps towards automatic translation. Below we discuss the challenges in describing PRL or SCL code with the limited faculties provided by Promela, and these challenges apply to both manual and automatic methods.

PRL Translation

Translating PRL into Promela is relatively simple, for a variety of reasons. The scope of every block and command is well established, so the control flow is very easy to understand. PRL's lack of support for global variables and pass by reference is convenient, since Promela doesn't support these features natively either. The presence of strings and enumerations seems at first to be a source of difficulty, but they can simply be exchanged for constants via `#define` directives. PRL also has the capacity to jump between steps, which we can emulate using a Promela statement label and `goto`. Automatic generation can easily handle these chores.

SCL Translation

Generating Promela to represent SCL is much less straightforward. In order to accurately emulate SCL, details such as rule and script scheduling must be taken into account even before the task-specific code can be examined. However, we defer discussion of these complexities and start with the core approach to procedure representation.

In order to support SCL's notions of passing by reference, return values, and global variables, Promela channels are used to shuttle values between parent and child functions, or, in the case of globals, between any two functions. Channels acting as return values also create a convenient way for parents to wait for their children to finish before beginning the next task, as the parent can simply try to read from the return channel and SPIN will block that execution path until the value is available. The state overhead for spawning a child process is reasonable, but not so small as to be ignored altogether. This is especially so in the case of nesting functions, where greater quantities of memory are required to track every function in the stack.

The next items to consider are rules and constraints. Rules are mechanisms which go into action when a variable or variables meet certain conditions. Although the SCL RTE's behavior is more complex, in our current approximation, when the rule's conditions are met, a subprocedure is executed. The most natural way to enforce this would be a separate, concurrent Promela process that constantly monitors the state of the simulation and fires these rules; unfortunately, we do not have a scalable way to accomplish this in Promela. A different, less-elegant solution has been devised: macros. As alluded to earlier when discussing PRL, Promela uses the C preprocessor. A macro is defined to call every rule (usually expressed as macros themselves) and is inserted between every spacecraft command, so that any state change can be observed immediately and acted upon, as it would be in SCL. Due to their similarity, constraints are handled in a similar, though slightly more complex manner: the check on variable state occurs *before* a command is issued, and if the constraint is violated, a flag is set indicating that the command should be jumped over using `goto`.

Every subprocedure within SCL has a priority which must be taken into account by the SCL scheduler, which dictates when each subprocedure may run. Though the exact internal workings are not known, we have built a simplified model wherein subprocedures are sorted first by priority, and second by order in which they've joined the queue. If not for Promela channels' `!!` operator, this requirement could be very difficult and expensive to enforce; a priority queue would have to be implemented within the very tight bounds of the language, probably with the effect of expanding the state space immensely. Fortunately, the `!!` operator essentially turns a message queue into a priority queue: messages entering a queue can be placed according to a priority expressed in the first message field. This allows a scheduler process to send an activation message to the highest priority process that arrived first in the queue.

Model Examples

To test our approach, we took an existing ISS procedure to reconfigure a power controller (for which we have a PRL encoding created by NASA's Automation for Operations program (Frank 2008)) and created a possible SCL implementation. We then systematically hand-translated the PRL and SCL representations into Promela models. The Promela translation of the PRL encoding follows the source language quite closely (compare Figure 8 to Figure 2).

The same procedure, this time written in SCL and translated to Promela, is shown in Figure 9. The most notable difference is that this procedure is now accomplished by a main process and two helper processes, rather than a single process. In order to emulate this behavior, our Promela model uses a message channel that holds a single `bool`. This channel is supplied to each child of the parent process. If the child has no meaningful value to return, an arbitrary value is supplied via this channel at the end of execution to indicate to the parent that the child has finished; otherwise, the appropriate value is sent to the parent where it can be handled. Note that in general the return value could be of any type; for example, a message with multiple fields

```
/* This macro collects the data in the
initial state channel into an array. */
POP_SENSORS;

/* Command instruction */
xmit!X_RPCMCommonClearCmdType;

/* Verify instruction: */
if
::      localSensors[RPC_POWER_ON_RESET]
         == ON -> skip
::      else ->
        goto exitFailure
fi;

/* Verify instruction: */
if
::      localSensors[RPCM_ORU_HEALTH_STATUS]
         == OKAY -> skip
::      else ->
        goto exitFailure
fi;

goto exitSuccess;

exitFailure:
xmit!Failure;
goto finish;

exitSuccess:
xmit!Success;
goto finish;

finish:
/* This macro puts the sensor data back
into the initial channel to be examined. */
PUSH_SENSORS;
}
```

Figure 8: The structure of PRL lends itself well to representation in Promela.

```
#define ENSURE_HEALTH_RULE\
  if \
  ::    localSensors[RPCM_ORU_HEALTH_STATUS]\
        != OKAY ->\
        goto returnFalse;\
  ::    else -> skip\
  fi
#define CHECK_RULES ENSURE_HEALTH_RULE

proctype issueCommand(mtype command;
                      chan xmit, ret)
{
  xmit!command;
  ret!true
}
proctype verifyEquality(mtype propertyValue,
               desiredValue; chan xmit, ret)
{
  if
  ::    propertyValue == desiredValue ->
        ret!true
  ::    else ->
        ret!false
  fi
}
proctype sclModel(chan xmit, initial)
{
  chan retChan = [1] of { bool }
  bool returnVal;

  POP_SENSORS;    /* Gather initial state */

  CHECK_RULES;
  /* Issue Common Clear command. */
  run issueCommand(X_RPCMCommonClearCmdType,
                   xmit, retChan);
  retChan?returnVal;
  CHECK_RULES;
  /* Verify Power On Reset status */
  run verifyEquality
   (localSensors[RPC_POWER_ON_RESET_STATUS],
   ON, xmit, retChan);
  retChan?returnVal;
  if
    ::  returnVal == false ->
        goto returnFalse;
    ::  else -> skip
  fi;
  CHECK_RULES;
  goto returnTrue;

  returnFalse:
  xmit!Failure;
  goto finish;

  returnTrue:
  xmit!Success;
  goto finish;

  finish:
  PUSH_SENSORS;
}
```

Figure 9: Emulating SCL in Promela is harder.

```
chan init1 = [I] of {short,short};
chan init2 = [I] of {short,short};
chan xmit1 = [M] of {short};
chan xmit2 = [M] of {short};

/* Repeat I times, one initial state
   sent for each input. */
if
 ::     init1!0,0; init2!0,0
 ::     init1!0,1; init2,0,1
        /* ... */
 ::     init1!0,D-1; init2,0,D-1
fi;
  /* ... more inits */
if
 ::     init1!I-1,0; init2!I-1,0
 ::     init1!I-1,1; init2,I-1,1
        /* ... */
 ::     init1!I-1,D-1; init2,I-1,D-1
fi;

/* Both models take in these input values,
then send M messages containing a random
input back to the xmit channels */
run Model1(init1, xmit1);
run Model2(init2, xmit2);

/* Iterate through each message from the
models */
do
  ::    nempty(xmit1) && nempty(xmit2) ->
        /* get front value of each channel */
  ::    empty(xmit1) || empty(xmit2) ->
        break;
od;
```

Figure 10: The abstract model is designed to mirror the equivalence model discussed previously, while allowing for simple quantitative scaling.

could be used to send multiple values. Through the use of the ENSURE_HEALTH_RULE macro, the verification of the health of the craft that was explicit in the PRL translation is now, to an extent, implicit. If need be, many more rules could be added to the system and added to the list of rules enforced by the CHECK_RULES macro.

For the sake of clarity in the demonstration, the SCL scheduler mechanism is not present in this example; however, the basis of its implementation should be clear and only a few modifications would be necessary to incorporate it.

Evaluation

Naturally, a scheme using exhaustive verification must be concerned with scalability, as larger and more complex procedures cause exponentially more possible states to be created. To evaluate the scalability of our general approach to procedure encoding and comparison, we created a scalable abstract Promela model that follows similar patterns. Figure 10 shows the structure of the abstract model and its similarities to the model used for actual comparison of procedures. We generated a spectrum of test samples by varying three dimensions: number of inputs (I), domain size for inputs (D), and message volume of each submodel (M). The abstract model creates I initial state variables, and each one is nondeterministically assigned an integer in the range $[0, D-1]$. Two sub-models abstractly represent the procedure encodings; each is passed an input channel just large enough to hold the I inputs, as well as a second "trace" channel with a size of M short integers. Through this channel, the sub-models send M messages. The content of the messages sent is randomly selected from the inputs. This was done to avoid sending the same value repeatedly, which SPIN might somehow optimize. Once both models have executed, the messages are drawn out of their channels one by one into local variables. No comparison is done, however, since the abstract procedure models do not have real semantics but choose their outputs randomly. We expect that omitting the comparison operators will have little effect on performance relative to other factors, since in the real procedure comparison process the outcome of these operators depends on the initial state, and does not cause nondeterministic branching in the state space.

Our evaluation results indicate that the primary concerns are the number and range of inputs, which cause an explosion in the size of the state space. We expect that the factor of state space increase should be $\in O(D^I)$.

We ran numerous tests on this abstract model, and present

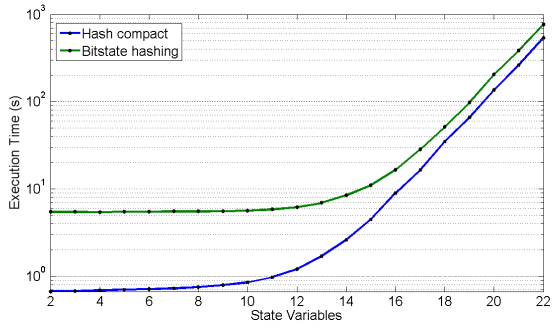

Figure 11: For smaller problems, bitstate hashing's overhead and potential loss of accuracy makes it undesirable.

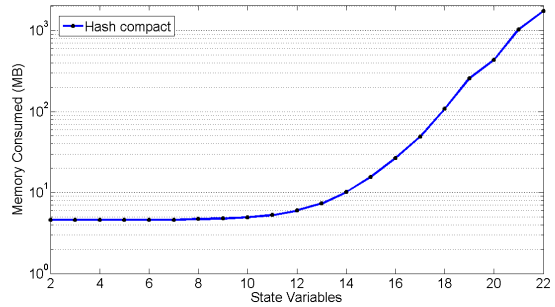

Figure 12: Even when states are highly compressed, small increases in the number of state variables can cost enormous amounts of memory during exhaustive verification.

results for tests having from two to twenty-two inputs, each with $D = 2$ and $M = 100$. Two SPIN modes were tested: one used `-DHC0`, which uses "hash table compaction" with the highest compression. This method is lossless and will always search exhaustively. The other SPIN mode used "bitstate hashing" (compile time flag `-DBITSTATE`), which does not guarantee completeness but, with well-chosen parameters, can explore the vast majority of the state space using a fixed amount of memory. For bitstate hashing, the default 3 bits per state were used, and the hashtable was allotted 2^{35} entries. Both modes were limited to 5000 megabytes of memory.

Figure 11 shows the relationship between the number of state variables used when modeling and the time taken to explore the model. The time expressed in the figure is the sum of the time taken to generate the C code, compile it using `gcc`, and run it. Clearly, hash compaction is more time efficient for tractable problems. However, these two methods operate very differently: hash compaction is complete and faster, but the memory it consumes increases directly with the size of the state space (and by extension, exponentially in the number of state variables). Figure 12 demonstrates this fact. The bitstate hashing method is not shown in this graph because it consumes a constant amount of memory. As state space increases, it can be seen that the hash compaction method will no longer be available when the size of the problem exceeds that of today's machines. Bitstate hashing is still possible on these problems because it keeps a hash table whose size is independent of the problem. The cost that bitstate hashing pays is instead in computation time and coverage. Although it can be used on problems of any size, if the task is large enough then the lack of coverage will reduce confidence in the result (Holzmann 1995), even with multiple verifications. It should be noted, however, that for the models verified in the figures, bitstate hashing found as many unique states as the hash compaction method – that is, all of them. Further investigation is needed to find at what point the accuracy of bitstate hashing drops below the point of being trustworthy.

Based on the results as visualized in the graph and in the fundamental mathematics, it seems crucial to avoid an increase in the number of inputs and the size of the input domains. Of the two, an increase in domain size seems most easily controlled, by intelligently deciding which initial values may be redundant. For example, if a value must be at least 50 to pass a check then the values 1 through 49 will act identically, so only one of these values need be tested; this would avert a large increase in state space. The details of implementing such a scheme are so far undetermined.

Related Work

This work is related to a wide variety of prior and ongoing research in verification of high-reliability systems including work on performed for NASA's ongoing Automation for Operations (A4O) project (Frank 2008).

NASA Procedure Verification

Previous work on verification of procedures for NASA missions has largely focused on verifying the *internal* consistency, safety, and semantics of individual procedures and scripts, rather than comparison between two implementations of a procedure. For example, recent NASA research on verification of procedures written in PRL has addressed static verification to ensure well-formed Program Universal Identifier references, as well as dynamic verification of assertions such as "after the state 'abort plan' is set to true, no node in the plan repeats (loops)" (Brat *et al.* 2008). Similarly, verification methods have been used to ensure static and limited dynamic properties of executable scripts coded in PLEXIL by translating the PLEXIL into Java and using the Java Pathfinder model checking tool.

Formal Verification Methods: Program Equivalence

The program equivalence problem for general programming languages attracts much good research from the larger verification research community. Program equivalence is formalized as several forms of *bisimulation*. In general, bisimulation refers to the idea that two programs have the same state transition structure. CADP (Construction and Analysis of Distributed Processes, `http://www.inrialpes.fr/vasy/cadp/`) is a popular suite of tools that can analyze abstract programs (formulated as Labelled Transition Systems (LTSs)) and provide numerous analyses including checking for common faults (*e.g.*, deadlock) as well as verifying more complex properties expressed in specifications such as temporal logic or mu-calculus. Given two programs formulated as LTSs (in our case, two procedures from different sources), the CADP bisimulation tool can check to see if the procedures are equivalent, modulo one of several *equivalence relations*. These relations, including strong equivalence, observational equivalence, and safety equivalence, provide different levels of guarantees about how the procedures correspond. We are interested in the application of CADP to the procedure comparison problem, but its high licensing fees prevented us from using it.

Verification of Procedures

The verification of procedures has been explored in other contexts, such as nuclear power plant operation. For example, Zhang (1999) has used SPIN model-checking to verify properties of operator procedures (*e.g.*, liveness), and developed an incremental approach for the construction of system models with increasing complexity in order to reduce the cost of finding mistakes. These techniques may be useful for spacecraft procedures when spacecraft models become more available, depending upon the complexity of the models and procedures and the performance of the verification tools.

Verification of Rule-Based Systems

Verifying the use of SCL rules is a specific case of the more general problem of verifying rule-based systems. Previous

work in this area is largely focused on improving the software development process, not on verifying the correctness of the outputs (Preece 2001). There is also interest in verifying the consistency and completeness of rule sets used for applications such as the semantic web (Paschke 2006). These techniques may be useful for testing properties of SCL rule bases.

Conclusion and Future Directions

We have shown how operational procedures captured in two different languages, SCL and PRL, can be translated into Promela and thence shown to be observationally equivalent. Our preliminary evaluations of scalability indicate that the approach is viable for reasonably compact procedures of the sort we expect to occur in practice.

However, we have not yet modeled significant aspects of the SCL and PRL languages, such as the full SCL scheduler and PRL's unordered statement blocks. Once we have developed a more complete mapping between the languages, automating the process of mapping the procedural languages to Promela should be fairly straightforward.

Adding more expressive correctness criteria would also increase the utility of our approach for procedure verification. In some cases, requiring exact execution trace matching is too strict. For example, some commands may be irrelevant or order may not matter for some set of commands. For those groups, it may be possible to accumulate the commands as an aggregate for comparison's sake. We must also consider where the meta-information about command order comes from. In PRL this can be expressed by using an unordered block. In SCL, a source code convention would have to be established to mark unordered groups of commands.

More importantly, we have not yet experimented with dynamic aspects of the spacecraft's response to commands, and its environment. Our experiments only tested procedure equivalence for different static initial conditions, because we lack a model of the spacecraft's response to the procedural commands. A significant advantage of the Promela/SPIN approach is that we should be able to interface the verifier to an existing spacecraft simulator, and have the simulator emulate the expected dynamics. This would allow SPIN to explore and verify procedure behaviors that rely on telemetry values changing in response to procedure commands. (Dealing with nondeterministic simulators is another interesting research challenge.) There are also finite state machine models of spacecraft which could be directly encoded in Promela.

We also hope to explore the use of non-enumerative formal methods such as proof systems to examine the procedure structures directly and prove that they are equivalent.

Acknowledgments

The authors would like to thank Mats Heimdahl for valuable conversations that led to our equivalence-checking approach. In addition, we would like to recognize Guillaume Brat, Jeremy Frank, and David Kortenkamp, whose feedback has helped to shape this project. This work was supported by NASA's Automation for Operations program under SBIR Contract NNX09CC43P. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NASA or the U.S. Government.

References

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

Buckley, B., and Vangaasbeck, J. 1994. SCL: An off-the-shelf system for spacecraft control. In *Third Int'l Symp. on Space Mission Operations and Ground Data Systems*, 559–568.

Frank, J. 2008. Automation for Operations. In *Proceedings of the AIAA SPACE 2008 Conference*.

Heimdahl, M. P. E.; Choi, Y.; and Whalen, M. 2002. Deviation analysis through model checking. In *Proc. IEEE In'tl Automated Software Engineering Conference*, 37–46. IEEE Society Press.

Holzmann, G. J. 1995. An analysis of bitstate hashing. In *Formal Methods in System Design*, 301–314.

Holzmann, G. J. 1997. The model checker SPIN. *IEEE Transactions on Software Engineering* 23(5).

Holzmann, G. J. 2005. Software model checking with SPIN. *Advances in Computers* 65:78–109.

Izygon, M.; Kortenkamp, D.; and Molin, A. 2008. A procedure integrated development environment for future spacecraft and habitats. In *Proceedings of the Space Technology and Applications International Forum (STAIF 2008), available as American Institute of Physics Conference Proceedings Volume 969*.

Kortenkamp, D.; Bonasso, R. P.; and Schreckenghost, D. 2007. Developing and executing goal-based, adjustably autonomous procedures. In *Proceedings AIAA InfoSysAerospace Conference*.

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)*.

Object Management Group. 2004. XML telemetric & command exchange (XTCE) – informative specification.

Paschke, A. 2006. Verification, validation and integrity of distributed and interchanged rule based policies and contracts in the semantic web. In *Second International Semantic Web Policy Workshop (SWPW'06)*, 2–16.

Preece, A. 2001. Evaluating verification and validation methods in knowledge engineering. In Roy, R., ed., *Micro-Level Knowledge Management*. Morgan-Kaufman. 123–145.

Zhang, W. 1999. Model checking operator procedures. In *Proc. Int'l SPIN Workshop*, 200–215. London, UK: Springer-Verlag.