Formal Methods for Comparing Behavior of Procedures in Different Languages

David J. Musliner, Michael W. Boldt, Michael J. S. Pelican, Daniel J. Geschwender
Smart Information Flow Technologies (SIFT)
Minneapolis, USA
email: {dmusliner,mboldt,mpelican}@sift.info

***Abstract*—We are developing software tools to compare the behavior of two executable procedures, written in the same or different languages. These tools are useful in situations such as verifying that an automated procedure does the same thing as its manual backup, ensuring that a change to a procedure impacts only the intended behavior, and verifying that a procedure converted to a new language maintains its behavior. Our newest tool translates each procedure into a common language, links the initial conditions of the procedures together, and uses symbolic execution to explore the behavior of every execution path, comparing the behavior of the two procedures on each path. In this paper, we describe our approach in detail and present a case study of applying our tool to NASA procedures that had previously been automatically translated from the Procedure Representation Language (PRL) to the Timeliner scripting language. Our tool easily scaled to these real-world procedures, and identified several bugs in the prototype translator.**

Keywords*— *procedure equivalence, symbolic execution

I. 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 [1]. The procedures may include sequences of primitive commands that change the state of onboard equipment, tests that verify certain state changes via input 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.

Automatically comparing the behavior two such procedures proves useful in many situations, such as verifying that an automated procedure does the same thing as its manual backup, ensuring that a change to a procedure impacts only the intended behavior, and verifying that a procedure converted to a new language or platform maintains its behavior. Comparing two procedures can be difficult because they may be written in different languages, and they may use different symbols and control structures to represent the same desired behavior.

In this paper, we describe our tool for automatically comparing the behavior of two procedures. We also present a case study applying this tool to verifying the equivalence of NASA procedures that had previously been automatically translated from PRL to the Timeliner scripting language. Our tool easily scaled to these real-world procedures and identified several bugs in the translator.

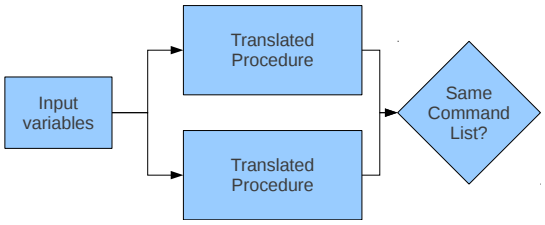

Fig. 1. Procedures are considered equivalent if their command outputs are the same.

II. Approach

A procedure has *inputs*, in the form of telemetry values that the procedure's *control logic* examines to determine what output *commands* should be issued. Our tool must determine whether, given a range of possible inputs, two procedures will issue the same output commands, in the same order. If not, the tool should identify the differences in a compact way.

To verify the functional equivalence of procedures, our approach includes the following steps:

1) Translate each procedure to C to enable the use of symbolic execution tools [2], [3].
2) Combine the translated procedures with driver code that sends them the same inputs and compares their output.
3) Using symbolic execution, repeatedly execute each procedure until all reachable combinations of paths through each procedure are covered, collecting and comparing the sequence of commands from each execution.

A. Translation to C

Procedure languages contain variables, operators, statements, and blocks, just like programming languages. *Steps* or *sequences* in procedures map nicely to functions in programming languages, as they represent parameterized, callable units of execution.

However, some procedure languages contain event-based statements, which trigger when certain conditions are met. For example, a procedure may take an action every 10 seconds, or when a sensor reading exceeds some threshold. Time-based

 ISBN: 978-1-61208-319-3

triggers can be easily translated to multi-threaded timer-based operations in standard programming languages. One way of dealing with other event triggers is to check for their trigger conditions between each statement of the translated procedure, though this seems inefficient and may explode the verification state space. Our case studies have not included event-based statements, so we have not needed to solve this problem yet.

To identify and compare the commands issued by the procedures, we map each command to a unique integer. We also use this mapping technique to handle command parameters. When a procedure issues a command, we append the corresponding integers for the command and its parameters to a list. This allows us to compare the commands issued by the two procedures by simply comparing the list of integers.

B. Composition and Execution

Once translated, the procedures are inserted into a framework that provides identical inputs to each procedure and compares their output, as shown in Figure 1. Then a symbolic execution tool repeatedly runs each procedure with different inputs, ensuring coverage of all combinations of reachable branches between the two procedures.

In symbolic execution, a user identifies a set of variables that he is concerned with and the tool rewrites the target program so that those "symbolic" variables can be controlled and tracked. The tool iteratively executes the instrumented program and updates its internal state with the results of the run. Each time the instrumented program executes, it initializes the symbolic variables with the values provided by the tool. As it executes, the target program records its execution, including the loads, stores, assignments, branches, function calls, and returns that relate to the symbolic (traced) variables. The tool iterates until it executes every code path in the target program or meets other stopping criteria.

In our case, we define a set of symbolic global input variables which can be accessed by both procedures. The symbolic execution engine will run the procedures numerous times, each time following a different execution path and generating output command lists. Comparing these lists effectively compares the behavior of the two procedures. Over the course of the runs, every reachable combination of paths through the procedures is executed, so all potential differences in behavior are considered.

III. Case Study: NASA Procedures

NASA is developing a translator to convert a set of PRL procedures to Timeliner. We created a software tool to verify the translation, using an implementation our approach in Section II. Our tool verifies that the original PRL procedure and the translated Timeliner procedure behave identically, or it identifies how their behavior can differ. Here we describe the implementation details and present the results of this case study, including translation errors identified by our tool.

A. 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 [4]. 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 manually-executable, or may include *automation data* to help describe how an 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.

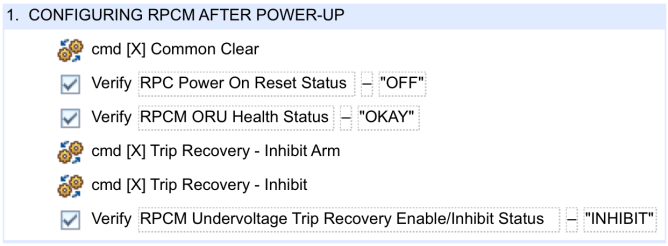

Fig. 2. The PRIDE integrated development environment supports relatively painless PRL editing and visualization.

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 PLEXIL language [4].

An Eclipse-based development environment, PRIDE, has been developed to simplify procedure authoring [5]. Figure 2 shows a fragment of PRL as it appears in the PRIDE environment. This example is one part of a procedure to configure an electrical component by issuing a series of commands and verifying assorted telemetry values.

As illustrated by the example in Figure 3, translating PRL into C is relatively simple. The scope of every block and command is well established, so the control flow is very easy to understand. PRL steps are analogous to functions. Most of the PRL blocks have straightforward conversions to C loops and conditionals. Only the "unordered" block has no direct equivalent. The PRL language contains many types of instructions, though we only encountered `CommandInstruction` and `VerifyInstruction` in this procedure set. A `CommandInstruction` issues a command, which is collected in a list for comparison in our translation. A `VerifyInstruction` is intended to test whether input telemetry shows that the state of the controlled system has reached some expected value. In our translation, we map this to a test of an input value; if the test fails, the procedure will halt.

 ISBN: 978-1-61208-319-3

```
int a_step1(node **commands) {
  //procedures/example.prl

  push(commands,x_rpcmcommonclearcmdtype,
      "instr11864393570120");

  if (! (x_rpcmpoweronresettype == off))
      { //=instr11864393688691
    return -1;
  }

  if (! (x_rpcmoruhealthtype == okay))
      { //=instr11864393820572
    return -1;
  }

  push(commands,x_rpcmundervoltagetrip
      recoveryinhibitarmcmdtype,
      "instr11864393957873");

  push(commands,x_rpcmundervoltagetrip
      recoveryinhibitfirecmdtype,
      "instr11864393982804");

  if (! (x_rpcmundervoltagerecovery
      inhibittype == inhibit)) {
      //=instr11864394053505
    return -1;
  }

  return 0;
}

node * a() { //=5.420
  //procedures/example.prl
  node *commands = NULL;
  if(a_step1(&commands) < 0) return
      commands;
  return commands;
}
```

Fig. 3. The PRL procedure from Figure 2 translated into C.

B. Timeliner

Timeliner is a suite of tools for building and deploying automated and semi-automated control systems. Timeliner has been utilized in several components of the International Space Satation, including the Command and Control Multiplexer-DeMultiplexer (MDM) and the Payload MDM. Timeliner's Logic Engine executes Timeliner Bundles, which are sets of procedures written in a language that combines fully autonomous and human interactive activities.

Timeliner has English-like code and many time-oriented keywords and control structures. This makes the language more accessible to system specialists without a background in computer programming. A Timeliner procedure is specified within a source file known a *Bundle*. A single Bundle may contain numerous *Sequences* and *Subsequences*, which are both groupings of Timeliner statements to be executed together. Sequences will execute in parallel by default, but can be made to execute serially. Subsequences may be called upon by any sequence to perform some common task. Sequences and even entire Bundles may be started and stopped at any time with the proper commands.

To illustrate some of the constructs of the Timeliner language, Figure 4 shows an approximate Timeliner translation of the partial PRL procedure shown in Figure 2. The `Master` sequence in our Timeliner encoding is the first sequence to execute and is responsible for beginning other sequences. In this example the `Master` sequence executes the sequence `step1`. Within `step1` are the four instructions present in the original PRL procedure. PRL's `CommandInstruction` is easily represented by a Timeliner `COMMAND` statement. PRL's `VerifyInstruction` is represented by a test of an input value; if the test fails, the translation sets an error flag and halts execution.

```
BUNDLE b5_420

  DECLARE bErrFlag BOOLEAN

  SEQUENCE Master ACTIVE

    -- STARTing step1
    START step1
    WHEN step1.SEQSTAT = SEQ_FINISHED THEN
      MESSAGE "Exiting Sequence Master
          successfully."
    END WHEN

  CLOSE SEQUENCE Master

  SEQUENCE step1

    COMMAND X.RpcmCommonClearCmdType

    SET bErrFlag = TRUE
    IF X.RpcmPowerOnResetType = "OFF" THEN
      SET bErrFlag = FALSE
    END IF
    IF bErrFlag = TRUE THEN
      WARNING "ERROR:X.RpcmPowerOnResetType"
      HALT b5_420
    END IF

    SET bErrFlag = TRUE
    IF X.RpcmOruHealthType = "OKAY" THEN
      SET bErrFlag = FALSE
    END IF
    IF bErrFlag = TRUE THEN
      WARNING "ERROR:X.RpcmOruHealthType"
      HALT b5_420
    END IF

    COMMAND X.RpcmUndervoltageTripRecovery
        InhibitArmCmdType

    COMMAND X.RpcmUndervoltageTripRecovery
        InhibitFireCmdType

    SET bErrFlag = TRUE
    IF X.RpcmUndervoltageRecoveryInhibit
        Type = "INHIBIT" THEN
      SET bErrFlag = FALSE
    END IF
    IF bErrFlag = TRUE THEN
      WARNING "ERROR:X.RpcmUndervoltage
          RecoveryInhibitType"
      HALT b5_420
    END IF

  CLOSE SEQUENCE step1

CLOSE BUNDLE b5_420
```

Fig. 4. The PRL procedure from Figure 2 translated into a Timeliner bundle.

Generating C to represent Timeliner is generally straightforward. For example, Figure 5 shows the translation of the Timeliner procedure in Figure 4. Expressions are similar to C expressions, so they require little processing. Most control

```
int b_step1(node **commands) {
  //procedures/example.tls

  push(commands,x_rpcmcommonclearcmdtype,
      "23");

  berrflag = 1;
  if (x_rpcmpoweronresettype == off) { //=29
    berrflag = 0;
  }
  if (berrflag == 1) { //=32
    fprintf(stderr, "warning: error:x.rpcm
        poweronresettype\n");
    return -1;
  }

  berrflag = 1;
  if (x_rpcmoruhealthtype == okay) { //=41
    berrflag = 0;
  }
  if (berrflag == 1) { //=44
    fprintf(stderr, "warning: error:x.rpcm
        oruhealthtype\n");
    return -1;
  }

  push(commands,x_rpcmundervoltagetrip
      recoveryinhibitarmcmdtype,"52");

  push(commands,x_rpcmundervoltagetrip
      recoveryinhibitfirecmdtype,"57");

  berrflag = 1;
  if (x_rpcmundervoltagerecoveryinhibit
      type == inhibit) { //=64
    berrflag = 0;
  }
  if (berrflag == 1) { //=67
    fprintf(stderr, "warning: error:x.
        rpcmundervoltagerecoveryinhibit
        type\n");
    return -1;
  }
  return 0;
}

int b_master(node **commands) {
  //procedures/example.tls
  if (b_step1(commands) < 0) return -1;
  fprintf(stderr, "message: exiting sequence
      master successfully.\n");
  return 0;
}

node * b() { //=b5_420
  //procedures/example.tls
  node *commands = NULL;
  if(b_master(&commands) < 0) return
      commands;
  return commands;
}
```

Fig. 5. The Timeliner procedure from Figure 4 translated into C.

statements and instructions present no translation problems. However, there are a handful of language features that could be difficult to translate if they were encountered.

One such feature is the ability of Timeliner sequences to run in parallel with each other. A Timeliner bundle will generally have one master sequence that can execute other sequences. In addition to the possibility of issuing interleaved commands, parallel sequence execution also allows for "contingency" sequences that constantly monitor a condition in order to respond with a series of commands. These constructs could be problematic to translate but they were not encountered in our case study. The case study procedures run all of their sequences serially.

Timeliner also includes several time-based control statements. Our current procedure comparison only compares the *sequence* of commands, maintaining no temporal information. If this timing is considered an important part of the procedure execution, then the comparison may be inaccurate, as this timing will be lost.

C. CREST

We use the CREST concolic execution tool to verify all execution paths without having to exhaustively check every input value. CREST is referred to as a "concolic" execution tool because it executes code both concretely and symbolically. CREST uses the C Intermediate Language (CIL) [6] to instrument a target program to simultaneously perform symbolic and concrete execution. To use CREST, a user identifies a set of variables that he is concerned with and CREST rewrites the target program so that those "symbolic" variables can be controlled and tracked. During each test iteration, CREST writes an inputs file, executes the instrumented program and updates its internal state with the results of the run. Each time the instrumented program executes, it initializes the symbolic variables with the values from the inputs file. If there are not enough inputs or if no inputs are specified, the target program initializes each symbolic variable with a random value. As it executes, the target program records its execution, including the assignments and branches that relate to the symbolic variables. User-specified search modes, such as random or depth first, use this execution information to choose inputs for subsequent test iterations. CREST runs test iterations until it executes every code path in the target program or meets other stopping criteria.

D. Testing Framework

To test procedure equivalence, our tool provides identical inputs to each procedure and compares the sequences of commands they issue. Figure 6 shows the structure of our code surrounding the two procedures. We use globally-defined symbolic variables for to represent the procedure inputs. Since they are symbolic variables, CREST will follow all branches involving them.

CREST executes this code numerous times, each time following a different execution path. Each translated procedure will generate its own command list. The two lists should match exactly along every execution path. If they do not, there is some difference between the two procedures, which the tool reports.

E. Results

We tested our tool on 61 different real-world procedures used by one of NASA's experimental manned space program platforms. The procedures totalled almost 18000 lines of PRL, containing 172 steps, 347 commands, and 394 verify statements. The largest procedure consisted of 28 steps with 102 commands and 136 verification statements The PRL

 ISBN: 978-1-61208-319-3

```
//Command list functions
struct node {...}
void push(node **head, int invalue,
    const char *location) {...}
int pop(node **head) {...}

//Command enumeration
enum mtype{
  Command1,
  Command2,
  ...}

//Input variable declaration
int Input1;
int Input2;
...

node * a() {
  //Translated code from Procedure A
}

node * b() {
  //Translated code from Procedure B
}

void main(void){
  //specify inputs to be handled
  //symbolically by CREST
  CREST_int(Input1);
  CREST_int(Input2);
  ...

  //run each procedure
  fprintf(stderr, ``Procedure A:\n'');
  node *a_commands = a();
  fprintf(stderr, ``Procedure B:\n'');
  node *b_commands = b();
  if (a_commands == NULL) {
    fprintf(stderr, ``A null!\n'');
  }
  if (b_commands == NULL) {
    fprintf(stderr, ``B null!\n'');
  }

  //loop through the commands until both
  //lists are empty
  while(a_commands!=NULL ||
      b_commands!=NULL) {
    //if the two commands are not the
    //same, the procedures differ and
    //this run is stopped
    int a_cmd = pop(&a_commands);
    int b_cmd = pop(&b_commands);
    fprintf(stderr, ``A: %d\nB: %d\n\n'',
        a_cmd, b_cmd);
    if(a_cmd != b_cmd) {
      fprintf(stderr, ``Failure - '');
      exit(1);
    }
  }
  fprintf(stderr, ``Success - '');
}
```

Fig. 6. The main program runs both procedures and compares the output commands to ensure their equivalence.

was converted automatically by a NASA-sponsored prototype translator into almost 15000 lines of Timeliner script.

Our tool easily scaled to these real-world procedures; testing the entire set takes less than 30 seconds on a modern laptop. Through this analysis, we identified four bugs in the prototype translator.

Two of the bugs were syntactic in nature. First, the translated Timeliner compares operator input to string values, but Timeliner does not accept string input. This was carried over from PRL, which *does* accept string input from the operator. Second, the Timeliner translation failed to declare the variable used to hold the results of some operations.

The remaining two bugs caused subtle behavioral differences between the procedures. The first occurs in procedures where the PRL contains an `IfThen` tag with no `Condition` tag. This is legal PRL syntax; it requires manual execution. The `title` attribute of the `IfThen` tag serves as the human-readable condition query, and an operator must input its value. The prototype PRL-Timeliner translator tool did not correctly take this into account. The Timeliner translation of such a construct has no `if` statement or condition. It does retain the instructions in the PRL `IfThen`, however, executing them even when the condition is false. The structure of this bug is shown in Figure 7.

The other behavioral difference also involves the `IfThen` tag. In PRL, an `IfThenBlock` may contain one or more `IfThen` tags, optionally followed by an `Else` tag. PRL semantics dictate that each `IfThen` and its `Else` are mutually exclusive, like an `if...elseif...else` construct. So, for the translation into Timeliner, the first `IfThen` should become an `IF` statement, and any following `IfThen` tags in the same `IfThenBlock` should become `ELSIF` statements. The prototype PRL-Timeliner translator instead translated all PRL `IfThen`s to Timeliner `IF` statements, allowing for more than one to execute in a single run. Additionally, the translator left out the `ELSE` clause, so those statements will always execute. The structure of this bug is shown in Figure 8.

While these translator bugs were easily repaired, they illustrate the value of our approach to automatic comparison of procedure behaviors.

IV. 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 [1]. 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)" [7]. Similarly, verification methods have been used to ensure static and limited dynamic properties of executable scripts coded in PLEXIL.

The verification of procedures has been explored in other contexts, such as nuclear power plant operation. For example, Zhang [8] 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,

 ISBN: 978-1-61208-319-3

```
<prl:IfThenBlock>
  <prl:IfThen title="Condition">
  <!    Action -->
  </prl:IfThen>
</prl:IfThenBlock>
```

(a)

```
Action
ENDIF
```

(b)

```
q = QUERY "Condition"
IF q THEN
  Action
ENDIF
```

(c)

Fig. 7. The conditionless `IfThen` semantics bug found by our tool. (a) Source PRL. (b) Incorrect Timeliner from translator. (c) Correct Timeliner translation.

```
<prl:IfThenBlock>
  <prl:IfThen>
    <prl:Condition>Cond1</prl:Condition>
    <!    Action1 -->
  </prl:IfThen>
  <prl:IfThen>
    <prl:Condition>Cond2</prl:Condition>
    <!    Action2 -->
  </prl:IfThen>
<prl:Else>
    <!    Action3 -->
  </prl:Else>
</prl:IfThenBlock>
```

(a)

```
IF Cond1 THEN
  Action1
ENDIF

IF Cond2 THEN
  Action2
ENDIF

Action3
```

(b)

```
IF Cond1 THEN
  Action1
ELSIF Cond2 THEN
  Action2
ELSE
  Action3
ENDIF
```

(c)

Fig. 8. The multiple-`IfThen` semantics bug found by our tool. (a) Source PRL. (b) Incorrect Timeliner from translator. (c) Correct Timeliner translation.

depending upon the complexity of the models and procedures and the performance of the verification tools.

The program equivalence problem for general programming languages attracts extensive research from the verification 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 is a popular suite of tools that can analyze abstract programs (formulated as Labelled Transition Systems (LTSs)) and verify complex properties expressed in specifications such as temporal logic or mu-calculus [9]. 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. High licensing fees prevented us from investigating CADP for our case study.

V. Conclusion and Future Directions

We have presented an approach to comparing the behavior of two procedures by translating them into a common language, linking their inputs, and using symbolic execution to explore all execution paths. We have illustrated the feasibility and usefulness of this analysis through a case study, in which we successfully identified bugs in a prototype language translator operating on real NASA procedures.

However, we have not yet modeled significant aspects of the Timeliner and PRL languages, such as parallel execution of Timeliner sequences and PRL's unordered statement blocks.

Adding more expressive correctness criteria would also increase the utility of our approach for procedure verification. In some cases, requiring an identical sequence of commands is too strict. For example, some commands may be irrelevant or order may not matter.

We are currently completing an integration of our procedure comparison tool into the PRIDE procedure editing environment, so that procedure authors can compare procedures and interact graphically with any identified behavioral differences.

Acknowledgments

The authors would like to thank Mats Heimdahl for inspiring our equivalence-checking approach. 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

[1] J. Frank, "Automation for Operations," in *Proceedings of the AIAA SPACE 2008 Conference*, 2008.

[2] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, July 1976.

[3] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu, "Differential symbolic execution," in *Proc. 16th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering*. ACM, 2008, pp. 226–237.

[4] D. Kortenkamp, R. P. Bonasso, and D. Schreckenghost, "A procedure representation language for human spaceflight operations," in *Proc. Int'l Symp. on Artificial Intelligence, Robotics and Automation in Space*, 2008.

[5] M. Izygon, D. Kortenkamp, and A. Molin, "A procedure integrated development environment for future spacecraft and habitats," in *Proc. Space Technology and Applications International Forum, available as American Institute of Physics Conf. Proc. Volume 969*, 2008.

[6] G. Necula, S. McPeak, S. Rahul, and W. Weimer, "CIL: Intermediate language and tools for analysis and transformation of C programs," in *Compiler Construction*, ser. Lecture Notes in Computer Science, R. Horspool, Ed. Springer, 2002, vol. 2304, pp. 209–265.

[7] G. Brat, M. Gheorghiu, D. Giannakopoulou, and C. S. Păsăreanu, "Verification of plans and procedures," in *Proc. IEEE Aerospace Conf.*, 2008.

[8] W. Zhang, "Model checking operator procedures," in *Proc. Int'l SPIN Workshop*. London, UK: Springer-Verlag, 1999, pp. 200–215.

[9] H. Garavel, F. Lang, R. Mateescu, and W. Serwe, "CADP 2010: A toolbox for the construction and analysis of distributed processes," in *Proc. Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, 2011.

 ISBN: 978-1-61208-319-3