AIAA Infotech@Aerospace Conference
and
AIAA Unmanned...Unlimited Conference
6 - 9 April 2009, Seattle, Washington

AIAA 2009-2033

Static Verification of Spacecraft Procedures

Erin M. Connors[*]

College of William and Mary, Williamsburg, VA, USA

Cesar A. Munoz[†]

National Institute of Aerospace, Hampton, VA, USA

Cameron Schnur[‡]

Stevens Institute of Technology, Hoboken, NJ, USA

Radu Siminiceanu[§]

National Institute of Aerospace, Hampton VA, USA

The Automation for Operations project (A4O), which is part of NASA's Exploration Technology Development Program, aims at advancing automation technology in support of spacecraft operations. Procedures that encode the means by which spacecraft systems are operated are a critical target of automation technology. In this paper, we present a static verification tool for NASA's Procedure Representation Language (PRL), a XML-based notation for specifying manual and autonomous operation procedures. The tool checks PRL files for valid syntactical structure and semantics, ensures that procedures are invoked correctly, consults a system-representation database to validate usage constraints, and, finally, verifies the consistency of logical conditions using a constraint solver. All these checks are implemented statically, i.e., at the moment when procedures are being created, before they are deployed and executed. The tool has been fully integrated into the Procedure Integrated Development Environment (PRIDE), a PRL authoring and editor environment developed by NASA.

Acronyms

A4O	Automation for Operations
DOM	Data Object Model
DPLL	Davis-Putnam-Logemann-Loveland Algorithm
PRIDE	Procedure Integrated Development Environment
PRL	Procedure Representation Language
RPC	Remote Power Control
RPCM	Remote Power Control Module
SMT	Satisfiability Modulo Theories
XTCE	XML Telemetry and Command Exchange

[*]LARSS Intern at NASA Langley from 2006 to 2008. Current Software Systems Engineer with the MITRE Corporation, Bedford, MA.

[†]`munoz@nianet.org`. Senior Research Scientist, National Institute of Aerospace, 100 Exploration Way, Hampton VA 23666, USA. Member AIAA.

[‡]LARSS Intern at NASA Langley in 2008.

[§]`radu@nianet.org`. Research Scientist, National Institute of Aerospace, 100 Exploration Way, Hampton VA 23666, USA. Member AIAA.

Copyright © 2009 by the American Institute of Aeronautics and Astronautics, Inc.
The U.S. Government has a royalty-free license to exercise all rights under the copyright claimed herein for Governmental purposes.
All other rights are reserved by the copyright owner.

I. Introduction

Producing reliable, correct code is a long-standing problem in the software development world. Software errors can have costly effects, particularly if they are found late during the software development process. The National Aeronautic and Space Administration (NASA) is no stranger to software errors and, unfortunately, in space missions such errors can have devastating effects. One of NASA's software-related incidents occurred in 1999 on the Mars Climate Orbiter. The orbiter was designed to function as an interplanetary weather satellite and a communications relay for the Mars Polar Lander, but it failed to achieve Mars orbit and was burned up in the planet's atmosphere due to a unit conversion error.[1] The same year the Mars Polar Lander was also lost. It crashed due to spurious signals that were generated when the landing gear was deployed. These signals gave the false indication that the Polar Lander had already landed. Together the Mars Climate Orbiter and Polar Lander cost $275 million to develop and launch.[2] The European Space Agency has also had its share of software problems. In 1996 the Ariane 5 veered off its flight path, broke apart, and exploded less than a minute after its launch. This failure was caused by incorrect attitude data due to a software exception.[3] Fortunately, in these instances no lives were lost. Software for crewed spacecraft operations, being safety-critical, must be held to a higher level of scrutiny. For such operations, *reliable* and *correct* code is essential. This paper examines the use of *static verification* techniques to improve the consistency and correctness of *procedures* for commanding spacecraft systems.

The Automation for Operations (A4O) project, which is part of NASA's Exploration Technology Development Program, involves the advancement of Artificial Intelligence technology for autonomous operations in support of crewed space missions. Spacecraft procedures that contain the information necessary to operate spacecraft systems are a critical target of this new technology. For instance, for the Space Station an Onboard Short Term Plan is sent up to the station daily. This mission plan consists of a set of tasks, which can be either manual procedures, such as command execution and telemetry checks, or automatic activities, such as data updates, or human activities. The mission plan gives necessary supporting information about procedures, including when and where a procedure should be run and by whom.[4]

Figure 1 shows part of an existing procedure of the International Space Station with the commands written out in English. This procedure deals with the Remote Power Control Module (RPCM) and is called RPCM POWER ON RESET. This procedure is run after the RPCM is powered up. It performs three commands on the RPCM, and verifies the telemetry from the RPCM to ensure that the commands have completed successfully. It then consults the configuration information to determine which of the Remote Power Controls (RPCs) associated with the RPCM need to be set to the Close Inhibit state, sets the state for each of the RPC's, and then checks the RPCM to confirm that it is in the correct state.

```
CONFIGURING RPCM AFTER POWER_UP
  OrderedBlock:
    cmd [X] Common Clear
    verify [X] RPC Power On Reset Status
    verify [X] RPCM ORU Health Status
    cmd [X] Trip Recovery: Inhibit Arm
    cmd [X] Trip Recovery: Inhibit
    verify [X] RPCM Undervoltage Trip Recovery Enable/Inhibit Status
```

Figure 1. A procedure to inhibit trip recovery.

The hierarchical structure of these procedures and the fact that it is written in plain English make it easy for human operators to follow them. However, the English syntax makes authoring of these kind of procedures error prone and automatic analysis very difficult. For those reasons, as part of A4O, NASA is developing the Procedure Representation Language (PRL),[5] an XML-based notation for specifying mixed operation procedures. The mixed operation nature refers to the fact that PRL supports *adjustable autonomy*, i.e., modes of execution where manual operations can be gradually automated and where, on the other hand, autonomous operations can seamlessly fall back to manual operations if required.

Static verification refers to mathematical techniques for the analysis of software that do not require the actual execution of the system. One of the better known and widely used forms of static verification is *type-checking*, a process used by compilers to catch errors such as adding a number to Boolean value. A more involved form of static verification is the generation of proof obligation conditions that guarantee the

correctness of a piece of code with respect to a formal specification.[6–9] These proof obligation conditions are usually too complex to be discharged automatically, so an interactive prover is used.

In this paper, we report the development of a static verification tool for PRL, called PRLChecker, that lies mid-way between a type-checker and a full verification condition generator. The tool checks PRL files for valid syntactical structure and semantics, ensures that procedures are invoked correctly, consults a system-representation database to validate usage constraints, and finally, checks the consistency of logical conditions using an automated constraint solver. PRLChecker is fully integrated into the Procedure Integrated Development Environment (PRIDE),[10] a PRL authoring and editor environment developed by NASA.

II. Procedure Representation Language

A PRL procedure is a set of actions structured in a hierarchical way. A procedure not only specifies the actions but also defines *when* the actions can be executed and *what* are the consequences of executing them. A procedure consists of parameters, local variables, and sub-tasks called steps. Steps are formed by blocks and end in one of three ways: 1) a conditional branch; 2) an unconditional branch to another step; or 3) a procedure exit point. Blocks are containers for instructions such as command execution, queries to humans, telemetry queries, and verification checks. The instructions are the basic command structures in PRL.

The procedure, step, block, and instruction components have meta data, such as textual comments, and automation data that provides execution control and monitoring capabilities. In particular the automation data includes *wait conditions* called `StartCondition` and `EndCondition`. These conditions specify when a particular component is ready to start, in the case of `StartCondition`, or end, in the case of `EndCondition`, its execution. It also includes *check conditions* called `Precondition`, `Invariant`, and `PostCondition` that must be satisfied at the beginning, during, and at the end of the execution, respectively. In contrast to wait conditions, check conditions yield execution failures if they are not satisfied.

The procedure, step, and instruction XML elements are displayed in Figures 2 and 3.

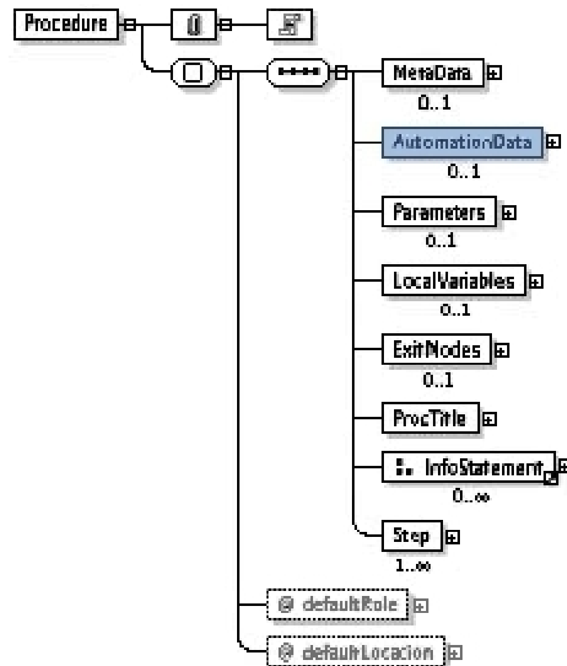

Figure 2. Schema for a procedure element.

NASA's Procedure Integrated Development Environment (PRIDE)[10] is a graphical environment that provides authoring and editing capabilities for dealing with PRL procedures. PRIDE presents different editing modes and different views of a procedure depending on tasks and user preferences, but uses a consistent data representation through all these views and modes. PRIDE aims to simplify the task of

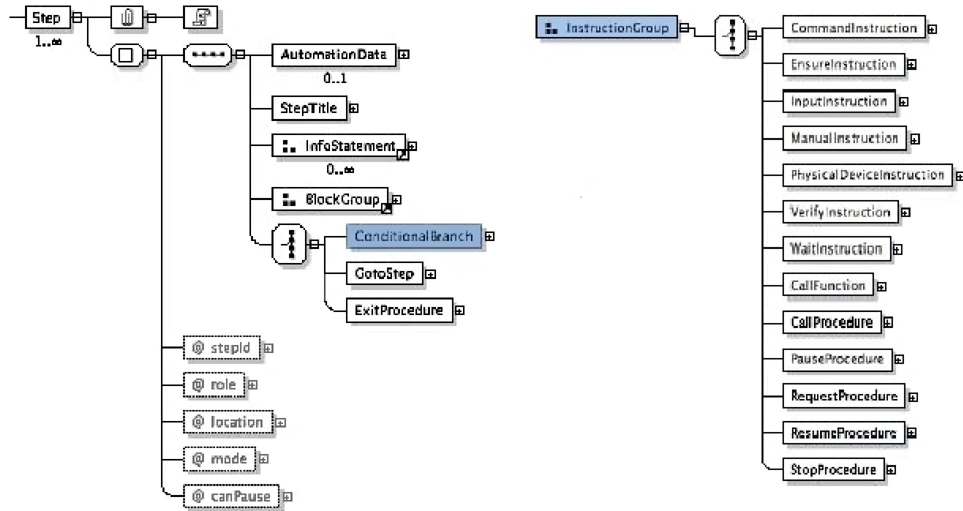

Figure 3. (a) Schema for a step element. (b) Schema for an instruction element.

authoring procedures and to improve their quality. It allows procedure authors to concentrate on the fields of their expertise, without worrying about details of editing and formatting. PRIDE is being implemented as a plug-in in the open-source development platform Eclipse. Figure 4 presents a screen-shoot of PRIDE.

To ensure that the PRL code generated by PRIDE is correct, we have implemented a tool, called PRLChecker, that verifies at compile time that the PRIDE-generated PRL code satisfies a pre-determined set of rules. Violations of these rules have been identified by experts as either actual errors or situations that potentially lead to actual errors.

III. PRL Static Checker

PRLChecker is a static checker for PRL that has been integrated into the PRIDE environment. Two main kinds of checks are implemented by the tool. The first set of checks enforce well-formedness properties that are not validated by PRL's XML-based schema. The following are examples of structural checks:

- Declarations, procedures, steps, and identifiers are unique.
- No missing or undeclared identifiers exist.
- No missing identifier attributes, identifier numbers, or names exist.
- Constant variables are not modified after initialization.
- Set of instructions is consistent with the level of autonomy.
- Conditions are Boolean expressions.
- Conditional branches and guards are complete and do not overlap.
- Conditional branches contain a default branch.
- Procedures are paused only by the appropriate callers of the procedure.
- Procedure calls are not circular.
- The step structure is a tree (no cycles, merging, or disconnected steps).

The second type of checks enforce basic semantic properties of PRL procedures and may depend on the domain description provided by a system representation database. The following are examples of semantic checks:

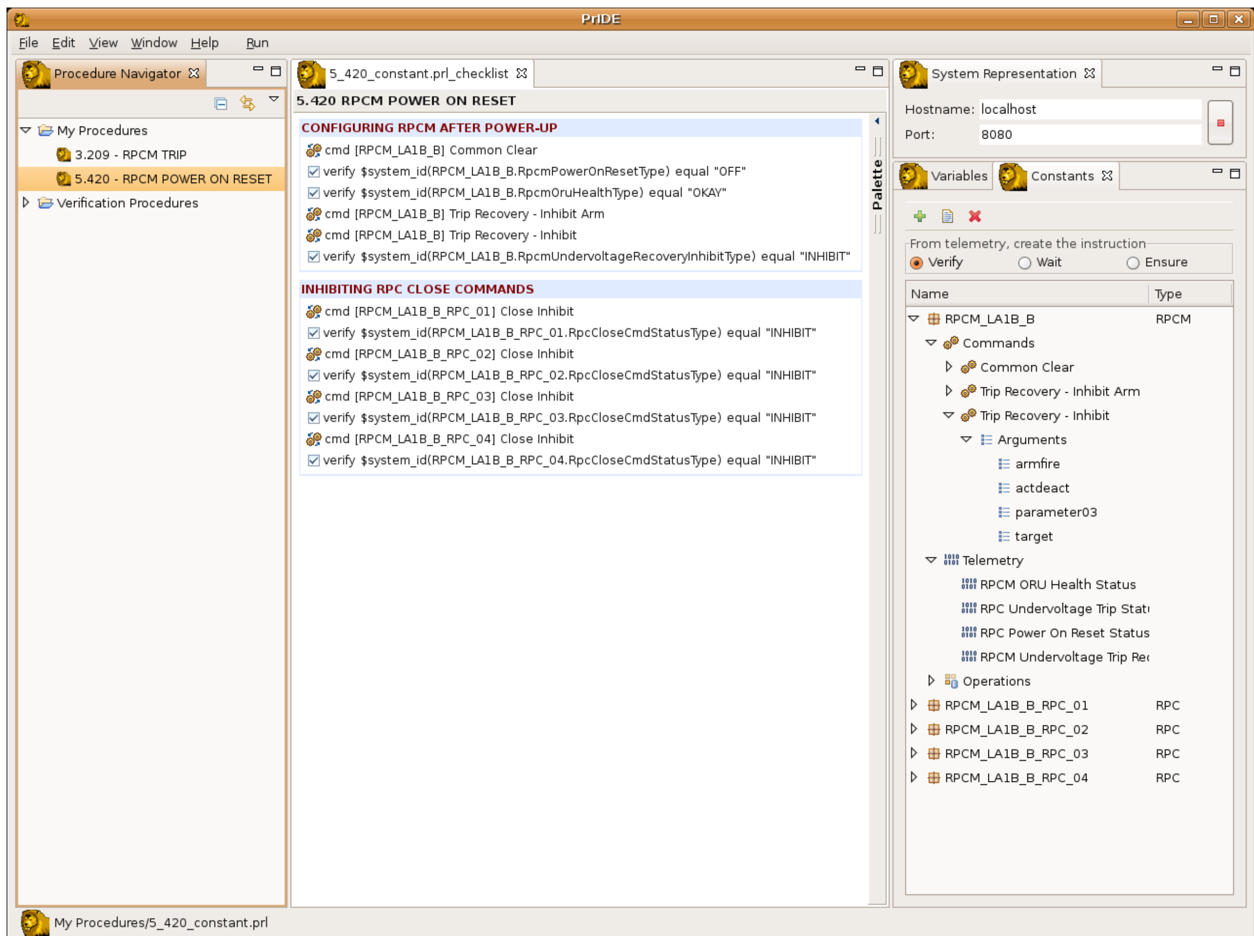

Figure 4. Procedure Integrated Development Environment tool.

- PRL expressions are correctly typed, e.g., variables are well-scoped and expression are well-typed.
- External types are defined in the system representation database.
- Commands and telemetry are defined in the system representation database.

As part of the semantic checks we include *consistency checking* of PRL conditions, i.e., wait conditions, check conditions, and guards in conditional branches. There are two main issues that arise with conditional branches. Firstly, each conditional branch of the condition statement must not overlap with any other branch. If two possible branches exist for a given condition statement, unintentional results may occur. Secondly, the conditional branches must be complete to ensure that in the absence of a default branch, the conditional branches cover all cases. Additionally, the wait and check conditions must be tested for consistency: 1) they should be eventually satisfiable and 2) they should not contradict each other.

III.A. XML Manipulation

PRL is a XML-based notation. The use of XML simplifies the task of manipulating the syntactical components of PRL since it enables the use of any off-the-shelf XML parser. In particular, we use the Data Object Model (DOM)[11] standard to create and manipulate a hierarchy of data objects to retrieve and modify the files' contents. A DOM is a tree structure, where each node contains one of the syntactical elements of an XML representation. The PRLChecker walks through all node elements and attributes, collecting information and performing the necessary checks. For some of these checks, the tool needs to query a system representation database to validate the access to external telemetry and commands. This database uses the XML Telemetric and Command Exchange (XTCE) standard format.[12]

III.B. Basic Checks

Some preliminary checks must be done to ensure that the XTCE database is available and that the PRLChecker is able to access it. If this is not the case, all checks that require the database are skipped and the user may receive some false type-checking errors for types that could not be validated without the database.

Additionally, the basic checks ensure that element identifiers are unique, that mandatory attributes are not missing, that local variables are declared, and that constant variables are not modified after initialization. In PRL, step, block, and instruction elements have identifiers, which are required to be unique within the scope of a procedure. The identifier of a procedure is assumed to be globally unique. To ensure uniqueness the checker creates a list of identifier names. When a new identifier is declared, the checker verifies that it has not been previously declared.

It is important that no required identifiers, attributes, identifier numbers, or names are missing. For example, if one procedure calls another but fails to provide the identifying information of the procedure to be called, an error must be issued. The checker verifies the presence of such required identifiers and produces an error if one is missing. When dealing with constant values, it is important to ensure that the value of the constant is not changed somewhere in the procedure. To check that constant variables are not modified after initialization the checker stores all constant variable names and values using a hash table. Whenever a constant is encountered, the checker verifies that its value is consistent throughout the procedure.

III.C. Level of Autonomy Checks

In PRL, both manual and automatic commands are allowed. Manual commands are to be completed by astronauts and automatic commands are to be completed by physical devices and computers. To enable this mixed mode of operation, PRL provides a level of autonomy mode for each computational element. Since certain instructions can only be completed when the procedure is in a particular mode, the checker ensures that the set of instructions is consistent with the level of autonomy. For example, the use of Request Procedure, Pause Procedure, and Resume Procedure is prohibited in manual mode.

III.D. Type Inference and Type-checking

PRL supports expressions of the following types: `BooleanValue`, `IntegerValue`, `RealValue`, `TimeValue`, `Text`, enumerations of integer values, and external types for commands and telemetry defined in the XTCE database. The PRLChecker implements both a type inference and a type checking algorithm. The type

inference algorithm takes a PRL expression and returns its PRL type. The type-checking algorithm verifies that expressions are well-typed with respect to the declared type.

We illustrate the type-checker with a few examples of ill-typed expressions and the corresponding output generated by the PRLChecker. Figure 5 shows an invalid PRL condition where a `RealValue` is compared to a `BooleanValue`. The error message contains the file name of the corresponding procedure and the location of the XML element, in a tree-based notation, where the error occurs.

```
<Condition>
  <Equal>
    <Constant>
      <RealValue>3.2</RealValue>
    </Constant>
    <Constant>
      <BooleanValue>false</BooleanValue>
    </Constant>
  </Equal>
</Condition>

** ERROR: RealValue cannot be compared to type BooleanValue
File: PRL/prl01.prl
At: /1/1/2/2/2
```

Figure 5. PRL code that illegally compares a real and a boolean with a relational operator.

The checker verifies that local variables and parameters are defined before they are used, and that when those variables are used they are of the expected type. For every command or telemetry reference, the type must be verified against the system representation database. Figure 6 shows the declaration of parameter `X` whose type is `RPCM`. When that parameter is used, the checker verifies that it has been declared either as a parameter or as a local variable. After that, it queries the XTCE database about the external type `RPCM`. Finally, it checks that `RpcmCommonClearCmdType` is a valid external command for a `RPCM` object. If one of these checks fails, an error is reported.

```
<Parameters>
  <Parameter externalType="RPCM" parameterIdentifier="X" parameterType="In"/>
</Parameters>

  ...

<Command>
  <CommandIdentifier>
    <DataReference source="system_id">
      <DataReference source="local">
        <Identifier>X</Identifier>
      </DataReference>
      <Identifier>RpcmCommonClearCmdType</Identifier>
    </DataReference>
  </CommandIdentifier>
</Command>
```

Figure 6. An example of data references in the PRL code.

III.E. Procedure and Step Structure

PRL specifies that the step structure is a directed graph with only one entry point. Circularities are to be avoided since they may create infinite loops. It is equally important to determine if a step is disconnected from the execution path and is therefore considered "dead code" since it will never be executed. In Figure 7,

Step_1, Step_2, and Step_3 are involved in an execution cycle. Step_4 is disconnected from the execution path. A merging call occurs when two different steps go to the same step. For instance, Step_5 is a merging step. This is permitted but is considered a warning since it requires synchronization, which is particularly difficult to implement in manual mode.

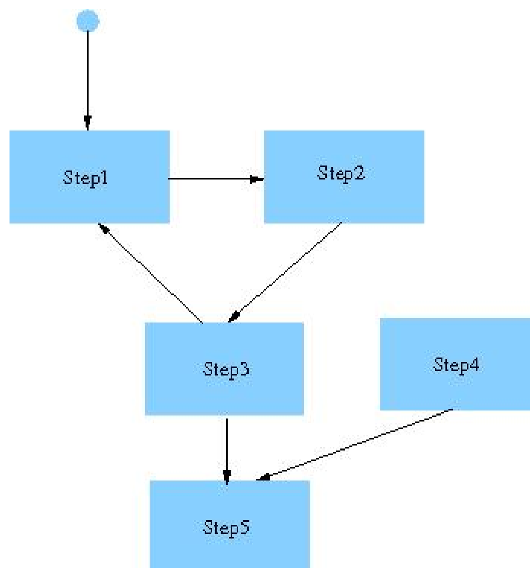

Figure 7. Circular, merging, and disconnected procedure calls.

The PRL specification allows a procedure to call or request another procedure. However, a procedure can pause another only if it is the parent or child of the procedure it tries to pause. To ensure that, the procedure hash table and graph are used to determine the current procedure's parent and child vertices. These vertex identifiers are then checked against the identifier of the procedure that is to be paused. Similarly, a procedure can only resume the execution of another procedure if it is the parent, child, or procedure that paused the procedure in question. The same steps are taken as before in order to check that a procedure is resumed correctly. Any procedure can stop the execution of another procedure.

III.F. Satisfiability Checking

PRL code allows for a series of conditions, including start, end, pre, post, invariant and branching conditions. There are two main issues that arise with conditional branch checking. Firstly, each conditional branch of the condition statement must not overlap with any other branch. If two possible branches exist for a given condition statement, unintentional results may occur. Secondly, the conditional branches must be complete to ensure that in the absence of a default branch, the conditional branches cover all cases. Additionally, the start, end, pre, post and invariant conditions must be tested for consistency.

To verify consistency, the checker generates instances of Satisfiability Modulo Theories (SMT) problems that are fed to the off-the-shelf SMT solver Yices,[13] an efficient SMT solver developed at SRI International. The Yices Boolean satisfiability solver is based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm,[14] a backtracking-based algorithm for deciding the satisfiability of propositional logic formulae in conjunctive normal form. Yices integrates its DPLL-based solver with specialized theory solvers that handle a variety of first-order theories: arithmetic, uninterpreted functions, bit vectors, arrays, recursive data-types, and more.

The input of Yices follows the SMT-LIB standard for satisfiability formulae, thus allowing for flexibility should a different SMT solver be used in the future. To check for overlapping branches for a condition with n branches, there are $\frac{n(n-1)}{2}$ instances generated. For example assume the following branching:

$$\texttt{if } x > 5 \texttt{ goto } Step_1 \tag{1}$$
$$\texttt{if } x < 0 \texttt{ goto } Step_2 \tag{2}$$

$$\texttt{if } x \leq 2 \texttt{ goto } Step_3 \quad (3)$$

In this case, there will be three satisfiability tests for branch overlapping:

- $x > 5 \ \wedge \ x < 0$ is unsatisfiable, which means that conditions 1 and 2 do not overlap.
- $x > 5 \ \wedge \ x \leq 2$ is unsatisfiable, which means that conditions 1 and 3 do not overlap.
- $x < 0 \ \wedge \ x \leq 2$ is satisfiable, for example take $x = -1$, which means that 2 and 3 do overlap.

To check for branch completeness, only one satisfiability check is required. In the previous example, the branch coverage is incomplete if and only if the negation of the disjunction of the conditions is satisfiable. In our case, we test wether $\neg(x > 5 \ \vee \ x < 0 \ \vee \ x \leq 2)$ is satisfiable or not. Since this is not the case, it means that there exists untreated cases such as $x = 3$.

The satisfiability portion of the checking tool must test all PRL allowed types that include: integers, strings, reals, booleans, and time types. Integer and boolean checking is fairly straightforward, but string, real, and time checking requires some additional computation. Yices does not allow for string satisfiability checking. The PRL schema only allows for equality condition checking for string types, which allows for the use of hash tables to accomplish the string checking. Each string value is used as a key value and the integer hash value is fed to the SMT solver for satisfiability checking. Real numbers are only allowed in fractional form in Yices and must be converted before they are checked. The time type values are defined by integer attribute values in the form of days, hours, minutes, seconds, and milliseconds.

For the start, end, pre, post, and invariant the checker first tests if the condition is satisfiable. Then, the checker tests that the pre-condition implies the invariant condition and that the invariant condition implies the post condition.

IV. Conclusion

Improving the correctness of a system and language code is a well-studied problem. Traditionally verification has been used to check that a system is correct by checking that its design specification is correct. However, even if the specification can be verified with formal methods, that does not ensure that the implementation of the design specification is correct.

Static checking is a program analysis technique that helps to find logical errors in the code without executing it. Static checking can greatly improve software productivity since it is cheaper and easier to correct an error earlier rather than later. Several modern static checkers already exist for various programming languages, such as ESC/Java,[15] FindBugs,[16] and JLint[17] for Java.

In this paper, we focused on PRL, a language for the specification of procedures used to automate spacecraft operations. Procedures contain the information necessary to operate a system with the capability of supporting human exploration missions. Mission planners use procedures as their building blocks to create a safe and successful plan that accomplishes mission goals. Hence, correct PRL procedure code is a vital part in maintaining the success and safety of these missions.

With a complex schema such as PRL, it is likely that errors could be introduced within a set of procedures. Software verification and validation ensures that software being developed or changed will satisfy functional requirements and that each step in the process of building the software produces the right results. The PRLChecker ensures that the procedures being created or modified will comply with a basic set of structural and constency rules.

The PRLChecker has been integrated into PRIDE, a PRL authoring tool. Figure 8 shows a screen-shot of the PRIDE environment with a few messages generated by the checker. In the center of the screen, a user can graphically create procedures using the available variables and constants, which are displayed on the left side of the screen. Along the bottom of the screen the errors produced by the PRL static checker are displayed. Currently, the checkers performs 40 different kinds of checks. As PRIDE becomes the standard way to write procedures for spacecraft missions, we will add more checks based on the feedback provided by procedure authors.

Finally, we recognize that the PRLChecker does not guarantee that the PRL code is absolutely free of errors. However, it increases the confidence in the correctness of the PRL authored code. Given the critical nature of crewed spacecraft operations, we strongly believe that mathematically-based tools such as PRLChecker are not only valuable, but required, for authoring of procedures for the next generation of aerospace

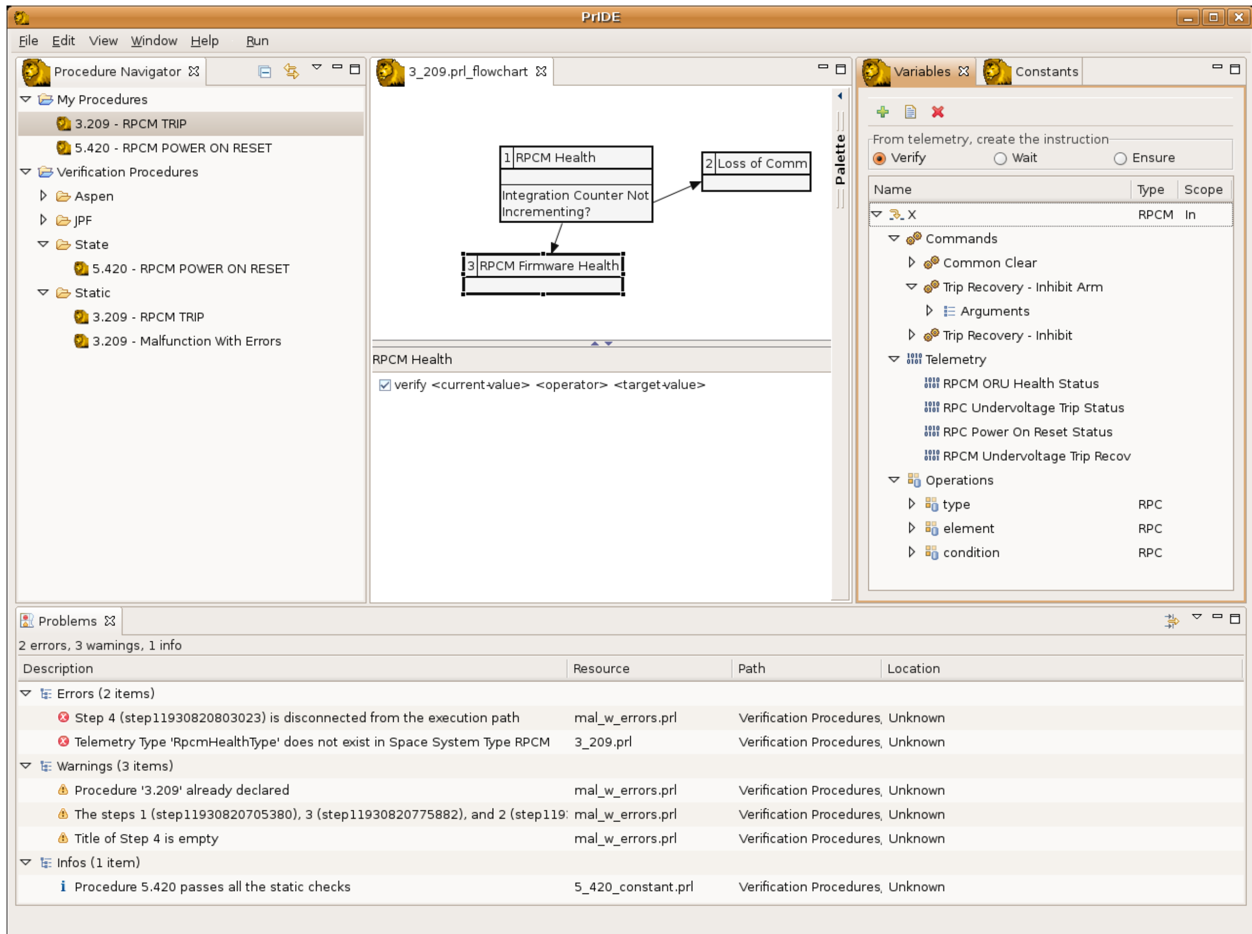

Figure 8. PRIDE authoring tool integrated with PRLChecker.

Acknowledgments

We would like to thank the members of the NASA Langley Research Center, Ames Research Center, Johnson Space Center, and the Jet Propulsion Lab for their contributions to this project.

References

[1]Oberg, J., "Why the Mars Probe Went Off Course, Spectrum Magazine," *Spectrum Magazine*, December 1999.

[2]MPIAT, "Mars Program Independent Assessment Team Summary Report," Tech. rep., MPIAT, March 2000, URL: `ftp://ftp.hq.nasa.gov/pub/pao/reports/2000/2000_mpiat_summary.pdf`.

[3]Lions, J. L., "Ariane 5: Flight 501 Failure," Tech. rep., European Space Aency Inquiry Board, July 1996.

[4]Kortenkamp, D., Bonasso, R., and Schreckenghost, D., "Procedures as a Gateway to Spacecraft Autonomy," *In Spacecraft Autonomy: Using AI to Expand Human Space Exploration. Papers from the 2006 Fall Symposium*, 2006.

[5]Kortenkamp, D., Bonasso, R. P., Schreckenghost, D., Dalal, K. M., Verma, V., and Wang, L., "A Procedure Representation Language for Human Spaceflight Operations," *Proceedings of the International Symposium on Artificial Intelligence, Robotics and Automation in Space (ISAIRAS)*, 2008.

[6]Homeier, P. V. and Martin, D. F., "A Mechanically Verified Verification Condition Generator," *Comput. J*, Vol. 38, No. 2, 1995, pp. 131–141.

[7]Filliâtre, J.-C. and Marché, C., "The Why/Krakatoa/Caduceus Platform for Deductive Program Verification," *Proceedings of the Computer Aid Verification Conference (CAV)*, 2007, pp. 173–177.

[8]Matthews, J., Moore, J. S., Ray, S., and Vroon, D., "Verification Condition Generation Via Theorem Proving," *Logic for Programming, Artificial Intelligence, and Reasoning, 13th International Conference, LPAR 2006, Phnom Penh, Cambodia, November 13-17, 2006, Proceedings*, edited by M. Hermann and A. Voronkov, Vol. 4246 of *Lecture Notes in Computer Science*, Springer, 2006, pp. 362–376.

[9]Barnett, M., Chang, B.-Y. E., DeLine, R., Jacobs, B., and Leino, K. R. M., "Boogie: A Modular Reusable Verifier for Object-Oriented Programs," *Formal Methods for Components and Objects (FMCO) 2005, Revised Lectures*, Vol. 4111 of *Lecture Notes in Computer Science*, Springer-Verlag, 2006, pp. 364–387.

[10]TRACLabs, "PrIDE: Procedure Integrated Development Environment," `http://www.traclabs.com/pride`.

[11]W3C, "Document Object Model (DOM)," World Wide Web Consortium, `http://www.w3.org/DOM`.

[12]CCSDS, "XML Telemetric and Command Exchange (XTCE)," 2005, The Consultative Committee for Space Data Systems.

[13]Dutertre, B. and de Moura, L., "The YICES SMT Solver," `http://yices.csl.sri.com/tool-paper.pdf`.

[14]Dutertre, B. and de Moura, L., "Integrating Simplex with DPLL," Tech. Rep. SRI-CSL-06-01, SRI International, May 2006.

[15]Leino, K. R. M., Nelson, G., and Saxe, J. B., "ESC/Java User's Manual," Technical note, Compaq Systems Research Center, Oct. 2000.

[16]Ayewah, N., Pugh, W., Morgenthaler, J. D., Penix, J., and Zhou, Y., "Using FindBugs on production software," *OOPSLA Companion*, edited by R. P. Gabriel, D. F. Bacon, C. V. Lopes, and G. L. S. Jr, ACM, 2007, pp. 805–806.

[17]Artho, C. and Havelund, K., "Applying Jlint to Space Exploration Software," *VMCAI*, edited by B. Steffen and G. Levi, Vol. 2937 of *Lecture Notes in Computer Science*, Springer, 2004, pp. 297–308.