Simulation-Based Verification of Livingstone Applications

A.E. Lindsey* Charles Pecheur†

Abstract

AI software is viewed as a means to give greater autonomy to automated systems, capable of coping with harsh and unpredictable environments in deep space missions. Autonomous systems pose a serious challenge to traditional test-based verification approaches, because of the enormous space of possible situations that they aim to address. Before these systems are put in control of critical applications, appropriate new verification approaches need to be developed. This article describes Livingstone PathFinder *(LPF), a verification tool for autonomous diagnosis applications based on NASA's Livingstone model-based diagnosis system. LPF applies state space exploration algorithms to an instrumented testbed, consisting of the Livingstone diagnosis system embedded in a simulated operating environment. The article describes different facets of LPF and reports some experimental results from applying LPF to a Livingstone model of the main propulsion feed subsystem of the X-34 space vehicle.*

1. Introduction

AI software is being considered more often as a means to give greater autonomy to automated systems, substituting for humans in places where they cannot or would not go themselves, such as distant planets, deep waters or battle fields. This trend is best exemplified by NASA's need for autonomous spacecraft, rovers, airplanes, and perhaps even submarines, capable of coping with harsh and unpredictable environments in deep space missions.

While autonomous systems offer promises of improved capabilities at a reduced operational cost, they pose a serious challenge to traditional test-based verification approaches, because of the enormous space of possible situations that they aim to address. Before these systems are put in control of critical applications, appropriate new verification approaches need to be developed.

*QSS Group, NASA Ames Research Center, Moffett Field, CA 94035, tlindsey@ptolemy.arc.nasa.gov

†RIACS, NASA Ames Research Center, Moffett Field, CA 94035, pecheur@ptolemy.arc.nasa.gov

This paper describes *Livingstone PathFinder* (LPF), a verification tool for autonomous diagnosis applications based on NASA's Livingstone model-based diagnosis system. LPF applies state space exploration algorithms to an instrumented testbed, consisting of the Livingstone diagnosis system embedded in a simulated operating environment.

Section 2 provides an overview of Livingstone; Section 3 describes the LPF architecture; Section 4 discusses its applicability; Section 5 reviews some experimental results; Section 6 compares LPF to related verification approaches; Section 7 draws conclusions and discusses some perspectives.

2. Livingstone

Livingstone is a model-based diagnosis system that uses a qualitative model of the different components of a physical plant and their interactions, both under nominal and faulty conditions [7]. It tracks the commands issued to the physical system, then compares the state predicted by the model against observations received from physical sensors.

If a discrepancy is detected, Livingstone performs a diagnosis by searching for combinations of faults that are consistent with the observations. Each combination is called a *candidate* and has an associated *rank* estimating its probability, higher ranked candidates being less likely. Livingstone's best-first search algorithm returns more likely, lower ranked candidates first. In particular, when the nominal case is consistent with Livingstone's observations, the empty candidate (of rank 0) is generated.

The Livingstone fault diagnosis and recovery kernel has successfully been applied to the Remote Agent Experiment (RAX) demonstration on Deep Sapce 1 (DS-1) in 1999. A new generation of Livingstone, called L2, includes temporal trajectory tracking. Further extensions supporting more general constraint types (hybrid discrete/continuous models) are under investigation.

3. Livingstone PathFinder

Livingstone PathFinder (LPF) is a simulation-based tool for analyzing and verifying Livingstone-based diagnosis ap-

plications. LPF executes a Livingstone diagnosis engine, embedded into a simulated environment, and runs that assembly through all executions described by a user-provided scenario file, while checking for various selectable error conditions after each step.

The architecture of LPF is depicted in Figure 1. The tool is built in a modular way with generic interfaces, to allow easy substitution of alternative versions of its different parts. Altogether, this provides a flexible, extensible framework for simulation-based analysis of diagnosis applications.

The testbed under analysis consists of the following three parts:

- *Diagnosis*: the diagnosis system being analyzed. LPF currently supports the Livingstone engine; extension to the Titan system [1] is under study.
- *Simulator*: the simulator for the device on which diagnosis is performed. Currently a second Livingstone engine instance is used for the simulator module, but the architecture is open to other alternatives.
- *Driver*: the simulation driver that generates commands and faults according to a user-provided scenario file.

All three components are instrumented so that their execution can be single-stepped in both forward and backward directions. The *Search Engine* controls the order in which the resulting graph of possible executions is explored.

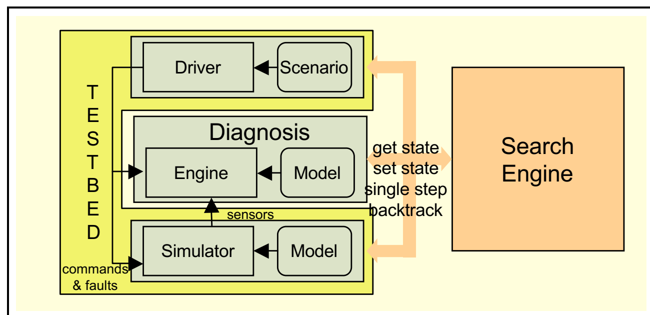

Figure 1. Livingstone PathFinder Architecture

The scenario file is essentially a non-deterministic program whose elementary instructions are commands and faults. Scenarios are built from individual instructions using sequential, concurrent (interleaved) and choice statements. The sample scenario in Figure 2 defines a sequence of three commands `cmd1`, `cmd2` and `cmd3`, with one fault chosen among `fltA` and `fltB` occurring at some point in the sequence. LPF provides a way to automatically generate a scenario combining all commands and faults of a model following the same sequence/choice pattern.

Each event produced by the Driver is passed to the Simulator, which updates its state accordingly. Commands (but not faults) are also passed to Diagnosis. Updated observable

```
mix {
  "command cmd1";
  "command cmd2";
  "command cmd3";
} and {
  choose "fault fltA";
  or "fault fltB";
}
```

Figure 2. Sample LPF scenario

variables are then extracted from the Simulator and passed on to Diagnosis. This cycle is repeated along all sequences covered by the scenario, saving and restoring intermediate states to explore alternate routes. User-selectable properties, such as consistency between the diagnosis results and the actual state of the Simulator, are checked at each step, and a trace is reported if a violation is detected.

3.1. LPF Error Conditions

In each state along its exploration, LPF can check one or several *error conditions* among a user-selectable set. Currently, LPF supports the following error conditions:

- *Simulator consistency*: Simulator reaches an inconsistent state, typically occurring after executing a command or injecting a fault that results in conflicts among the model constraints and assignments.
- *Diagnosis consistency*: Diagnosis reaches an inconsistent state, after failing to find any candidate consistent with previous commands and observations.
- *Mode comparison*: compares the modes of all components in the Simulator to those assumed by Diagnosis, and reports any discrepancy.
- *Candidate matching*: checks that at least one Diagnosis candidate matches (i.e. has the same faults as) the Simulator state.
- *Candidate subsumption*: checks that at least one Diagnosis candidate subsumes (i.e. has its faults included in) the Simulator state.

While the first two conditions can prove to be useful debugging tools, the subsequent conditions address the core of diagnosis correctness at three different levels of generality, from the most restrictive to the least. They constitute three successive refinement steps of the intuition that diagnosis should properly track the state of the (simulated) system. *Mode comparison* only considers the most likely candidate and reports errors even if another reported candidate matches the state, which is overly restrictive in practice. Instead, *candidate matching* considers all candidates. Even then, a fault may often stay unnoticed without causing any

harm, as long as its component is not solicited. Experience shows that this is a frequent situation, causing a large proportion of spurious error reports. In contrast, *candidate subsumption* only reports cases where none of the diagnosed fault sets is *included* in the actual fault set. In particular, the empty candidate subsumes any fault set and thus never produces an error. While this will not detect cases where Diagnosis misses harmful faults, it will catch cases where the wrong faults are reported. This condition has proven to be the most productive so far, as further discussed in Section 5.

3.2. Simulators in LPF

The modular design of LPF allows the use of different simulators through a generic application programming interface (API). As a first step, we have been using a second Livingstone engine instance for the Simulator (used in a different way: simulation infers outputs from inputs and injected faults, whereas diagnosis infers faults given inputs and outputs).

Using the same model for simulation and diagnosis, as this implies, may appear as circular reasoning but has its own merits. It provides a methodological separation of concerns: by doing so, we validate operation of the diagnostic system under the assumption that the diagnosis model is a perfect model of the physical system, thus concentrating on proper operation of the diagnosis algorithm itself. Incidentally, it also provides a cheap and easy way to set up a verification testbed, even in the absence of an independent simulator.

On the other hand, using the same model for the simulation ignores the issues due to inaccuracies of the diagnosis model w.r.t. the physical system, which is a main source of problems in developing model-based applications. We are currently considering integration of higher fidelity simulators (e.g. Matlab or Simulink models). We have already been studying integration of NASA Johnson Space Center's CONFIG simulator system [2]. Note that higher fidelity simulators are likely to be less flexible and efficient; for example, they may not have backtracking or checkpointing.

3.3. Search Strategies

LPF supports alternative state space exploration strategies. The first implemented search strategy is a straightforward depth-first search; and an implementation of best-first search using configurable fitness functions has just been completed. Other strategies, such as interactive (i.e. user-driven) or randomized search, are under consideration.

Best-first search uses a fitness function to rank different states according to a given criterion and explores them in that order. For example, one heuristic used favors situations where fewer diagnostics were found, as they are more likely to lead to a case where no diagnostic can be found. Further experiments are planned to define and compare different heuristics for this kind of application.

3.4. Implementation Notes

LPF is written in Java and makes heavy use of Java interfaces to support its modular architecture. The core source base has 47 classes totaling 7.4K lines. The top-level software architecture mimics the structure of Figure 1 above, with generic interfaces `MIR` (Mode Identification and Recovery, i.e. Diagnosis) and `Simulator`.

LPF as a whole implements the *Generic Verification Environment* (GVE) API, which defines the interaction between a dynamic transition system and the search algorithm that explores it. Concretely, the heart of GVE is a `VirtualMachine` interface, defining generic methods for enumerating transitions and saving and restoring states. This allows re-use of search engines across different GVE-based applications. In LPF, the Diagnosis and its testbed together implement a `VirtualMachine` that can be searched in various ways. As another example, *Java PathFinder* (JPF) [4] uses a customized Java virtual machine to provide a GVE interface for analyzing Java programs. Indeed, the best-first search code used in LPF was initially developed for JPF. Incidentally, the Driver is also built as an assembly of simple `VirtualMachine` objects.

The Livingstone engine is a C++ program; it is accessed by LPF through a Java Native Interface (JNI). We have worked in close cooperation with Livingstone developers to ensure that all necessary functionality is accessible through the JNI. In particular, a checkpointing mechanism has been added to Livingstone to support backtracking search in LPF. For performance reasons, checkpoints are kept inside the Livingstone engine; only references are passed through the JNI.

The latest version of LPF offers a number of useful auxiliary capabilities, such as generating an expanded scenario tree, counting scenario states, exploring the simulator alone (no diagnosis), generating a default scenario, and producing a wide array of diagnostic and debugging information, including traces that can be replayed in Livingstone simulation tools.

4. Verification Using LPF

Because it executes the actual Livingstone system, LPF is a fairly general verification tool for Livingstone applications, in that it can reveal the following different types of problems:

- *Engine errors*, i.e. inaccurate diagnosis due to errors in the diagnosis program. These are not the main con-

cern, as the engine is re-used across many applications and therefore presumably more mature and stable.

- *Diagnosability errors*, i.e. inaccurate diagnosis due to lack of observability of the system's internal state.
- *Incompleteness errors*, i.e. inaccurate diagnosis due to intentionally incomplete search (e.g. due to limited memory resources).
- *Modeling errors*, i.e. inaccurate diagnosis due to aspects of the system incorrectly or too coarsely represented in the Livingstone model. This a major issue, as Livingstone operates on highly abstracted models of the physical system.
- *Integration errors*, where the diagnosis system fails to properly interact with its simulated environment, and the simulator in particular.

The last two items become relevant only when a higher-fidelity simulation of the diagnosed system is used. Alternatively, using the same Livingstone model for the simulator focuses exclusively on engine, diagnosability and incompleteness errors.

5. Experimental Results

Livingstone is being considered for Integrated Vehicle Health Maintenance (IVHM) for NASA's next-generation space vehicles. In that context, the PITEX experiment has demonstrated the application of Livingstone-based diagnosis to the main propulsion feed subsystem of the X-34 space vehicle [8, 3], and LPF has been successfully applied to the PITEX model of X-34. That Livingstone model contains 535 components and 823 attributes, compiling down to 2022 propositional clauses.

Two different scenarios have been used to analyze the X-34 model:

- The *random scenario* covers a set of commands and faults combined according to the sequence/choice pattern of Figure 2. This scenario is a trimmed down version of the one generated by LPF, and covers over 10,000 states.
- The *PITEX scenario* combines one nominal and 29 failure scenarios, derived from those used by the PITEX team for testing Livingstone, as documented in [3]. This scenario has 88 states.

LPF covers the *random* scenario at an average rate of 50 to 100 states per minute. Early experiments demonstrated the technical viability of the approach and provided informative feedback to the developers. Experience revealed a need for improved post-treatment of generated results, to filter critical information from large amounts of generated data. Using the *candidate matching* condition, both scenarios report an excessive number of errors, most of which are irrelevant.

However, verification over the *random* scenario using the *candidate subsumption* condition reported five violations (the *PITEX* scenario shows none). As an example, one of the reported errors involves a solenoid valve, `sv02`, which sends pressurization helium into a a rocket propellant tank: a command `close` is issued to the valve, but the valve fails and remains open—in LPF terms, a fault is injected in the simulator. This scenario corresponds to the following sequence of LPF events:

```
command test.sv02.valveCmdIn=close
fault test.sv02.rp1sv.mode=stuckOpen
```

At this point, LPF reports a violation of the *candidate subsumption* condition, indicating that none of the candidates found by Diagnosis covers the injected fault. The fault is detected (otherwise no faulty candidates would be generated), but incorrectly diagnosed. Indeed, the candidates and ranks after the fault occurs are:

```
Candidate 0)
      4#test.sv02.openMs.modeTransition=faulty :3
Candidate 1)
      3#test.sv02.openMs.modeTransition=faulty :3
Candidate 2)
      2#test.sv02.openMs.modeTransition=faulty :3
Candidate 3)
      -#test.sv02.openMs.modeTransition=faulty :3
Candidate 4)
      -#test.sv02.rp1sv.modeTransition=unknown :4
```

The first four candidates consist of a faulty open microswitch sensor at different time steps (microswitches report the valve's position). The last candidate consists of an unknown solenoid valve fault mode.

As of this writing, this error is still under investigation. There are a number of explanations that could account for the improper diagnosis: fault ranks in the X-34 model may need retuning; the number of candidates returned or the Livingstone search space (number of candidates Livingstone searches over to find the diagnosis) may need to be increased; or issues with the Livingstone program itself need deliberation.

6. Related Work

Livingstone PathFinder is a hybrid between testing, in the sense that it executes the real program code, and model checking, because it can backtrack among alternative executions to explore a non-deterministic state graph. As mentioned in Section 3.4, its top-level architecture is shared with *Java PathFinder* [4].

The VeriSoft tool [5] is another example where a backtracking search is applied to real code. Under a different perspective, the *Livingstone-to-SMV translator* [6] is another complementary verification tool for Livingstone applications, focusing exclusively on the Livingstone model but allowing true exhaustive analysis through the use of symbolic model checking.

7. Conclusions and Perspectives

Livingstone PathFinder (LPF) is a software tool for automatically analyzing model-based diagnosis applications across a wide range of scenarios. *Livingstone* is its current target diagnosis system, however the architecture is modular and adaptable to other systems. LPF has been successfully demonstrated on a real-size example taken from a space vehicle application.

LPF is under active development, in close collaboration with Livingstone application developers at NASA Ames. After considerable efforts resolving technical issues in both LPF and relevant parts of Livingstone, we are now returning useful results to application specialists, who in turn reciprocate much needed feedback and suggestions on further improvements. The *candidate subsumption* error condition is the latest fruit of this interaction. Directions for further work include new search strategies and heuristics, additional error conditions including capture of application-specific criteria, improved post-treatment and display of the large amount of data that is typically produced.

We are also investigating adapting LPF to use MIT's Titan model-based executive [1], which offers a more comprehensive diagnosis capability as well as a reactive controller. This extends the verification capabilities to involve the remediation actions taken by the controller when faults are diagnosed. In this regard, LPF can be considered as evolving towards a versatile system-level verification tool for model-based controllers.

Acknowledgments

This research is funded by NASA under ECS Project 2.2.1.1, *Validation of Model-Based IVHM Architectures*. The Authors would like to thank Livingstone application developers at NASA Ames, especially Sandra Hayden and Adam Sweet, for their active cooperation, and Livingstone lead developer Lee Brownston for his responsive support.

References

[1] B. Williams, M. Ingham, S. Chung and P. Elliot, Model-based Programming of Intelligent Embedded Systems and Robotic Space Explorers. To appear in *IEEE Modelings and Design of Embedded Software*, (August 2002).

[2] L. Fleming, T. Hatfield and J. Malin. Simulation-Based Test of Gas Transfer Control Software: CONFIG Model of Product Gas Transfer System. Automation, Robotics and Simulation Division Report, AR&SD-98-017, (Houston, TX:NASA Johnson Space Center Center, 1998).

[3] H. Cannon and C. Meyer. PITEX Option 1 Expansion Path Test Plan Version 1.1. Propulsion IVHM Technology Experiment Project, (NASA Ames & Glenn Research Center, 2003).

[4] W. Visser, K. Havelund, G. Brat and S. Park. Model Checking Programs. International Conference on Automated Software Engineering. September 2000.

[5] P. Godefroid. Model Checking for Programming Languages using VeriSoft. Proceedings of the 24th ACM Symposium on Principles of Programming Languages, pages 174-186, Paris, January 1997.

[6] Charles Pecheur, Reid Simmons. From Livingstone to SMV: Formal Verification for Autonomous Spacecrafts. In:Proceedings of First Goddard Workshop on Formal Approaches to Agent-Based Systems, NASA Goddard, April 5-7, 2000. In: Lecture Notes in Computer Science, vol. 1871, Springer Verlag.

[7] B. Williams and P. Nayak, A Model-based Approach to Reactive Self-Configuring Systems. *Proceedings of the National Conference on Artificial Intelligence*, (August 1996).

[8] A. Bajwa and A. Sweet. The Livingstone Model of a Main Propulsion System. In *Proceedings of the IEEE Aerospace Conference*. IEEE, 2002.