Simulation-Based Verification of Autonomous Controllers via Livingstone PathFinder

A.E. Lindsey[1] and Charles Pecheur[2]

[1] QSS Group, NASA Ames Research Center, Moffett Field, CA 94035, U.S.A.
tlindsey@ptolemy.arc.nasa.gov
[2] RIACS, NASA Ames Research Center, Moffett Field, CA 94035, U.S.A.
pecheur@ptolemy.arc.nasa.gov

Abstract. AI software is often used as a means for providing greater autonomy to automated systems, capable of coping with harsh and unpredictable environments. Due in part to the enormous space of possible situations that they aim to address, autonomous systems pose a serious challenge to traditional test-based verification approaches. Efficient verification approaches need to be perfected before these systems can reliably control critical applications. This publication describes *Livingstone PathFinder* (LPF), a verification tool for autonomous control software. LPF applies state space exploration algorithms to an instrumented testbed, consisting of the controller embedded in a simulated operating environment. Although LPF has focused on NASA's Livingstone model-based diagnosis system applications, the architecture is modular and adaptable to other systems. This article presents different facets of LPF and experimental results from applying the software to a Livingstone model of the main propulsion feed subsystem for a prototype space vehicle.

1 Introduction

Complex decision-making capabilities are increasingly embedded into controllers. Robots substitute for humans in hazardous environments such as distant planets, deep waters or battle fields. This trend is perhaps best exemplified by NASA's need for autonomous spacecrafts, rovers, airplanes and submarines, capable of executing in harsh and unpredictable environments. Moreover, increased autonomy can be a significant cost saver, even in more accessible regions, by reducing the need for expensive human monitoring. An important trend in autonomous control is *model-based autonomy*, where control is performed by a generic engine applying automated reasoning techniques to a high-level model of the system being diagnosed. This is the case for the Livingstone diagnosis system on which the work presented here is based. The idea behind model-based solutions is that the system, in any situation, will be able to infer appropriate actions from the model.

While autonomous systems offer promises of improved capabilities at reduced operational costs, a serious challenge to traditional test-based verification approaches occurs because of the enormous space of possible scenarios such systems aim to address. Several factors make testing advanced controllers particularly difficult. First, the range of situations to be tested is significantly larger because the controller itself is more complex and designed to operate over a broad range of unpredictable situations during a prolonged time span. The program implicitly incorporates response scenarios to any combination of events that might occur, instead of relying on human experts to handle off-nominal cases. Second, autonomous systems close the control loops and feature concurrent interacting components. Thus, it becomes more difficult to plug in test harnesses and write test runs that drive the system through a desired behavior. Third, concurrency may introduce hard-to-detect, non-deterministic race conditions.

This paper describes a flexible framework for simulating, analyzing and verifying autonomous controllers. The proposed approach applies state space exploration algorithms to an instrumented testbed, consisting of the actual control program being analyzed embedded in a simulated operating environment. This framework forms the foundation of *Livingstone PathFinder* (LPF), a verification tool for autonomous diagnosis applications based on NASA's Livingstone model-based diagnosis system.

LPF accepts as input a Livingstone model of the physical system and a scenario script defining the class of commands and faults to be analyzed. The model is used to perform model-based diagnosis and will be used to simulate the system as well. The tool runs through all executions specified in the script, backtracking as necessary to explore alternate routes. At each step, LPF checks for error conditions, such as discrepancies between the actual simulated faults and those reported by diagnosis. If an error is detected, LPF reports the sequence of events that led to the current state.

To avoid confusion in reading what follows, it is important to clearly distinguish two similar-sounding but very different notions, that we refer to as *faults* and *errors*:

- *Faults* are physical events causing the controlled system to behave abnormally, such as a stuck valve or a sensor emitting erroneous measurements. They typically result from external degradation, and are handled by making the system fault-tolerant. Detecting and identifying faults is the goal of diagnosis systems such as Livingstone.
- *Errors* are improper behaviors of the controller, such as inaccurately identifying the current state of the controlled system. They typically result from unintended flaws in design or configuration, and need to be eliminated before the system is deployed. Detecting and identifying errors is the goal of verification techniques and tools such as LPF.

This article discusses Livingstone PathFinder on three different levels:

- As a *verification approach*, LPF applies a combination of model checking and testing principles that we refer to as *simulation-based verification*. This is the subject of Section 2.
- As a *program framework*, LPF provides an infrastructure for applying simulation-based verification to autonomous controllers. This is referred to in Section 4.5.
- As a *concrete program*, LPF currently instantiates the framework to applications based on the Livingstone diagnosis system. This constitutes the central part of the paper, mainly discussed in Section 4.

A preliminary account of this work was presented in [6]. The remainder of the paper is organized as follows: Section 2 presents the general simulation-based verification approach; Section 3 provides an overview of Livingstone; Section 4 describes the LPF software tool; Section 5 reviews some experimental results; Section 6 draws conclusions and perspectives.

2 Simulation-Based Verification

The approach we follow in the work presented is a composite between conventional testing and model checking, here referred to as *simulation-based verification.* Similar to conventional testing, it executes the real program being verified rather than an abstract model derived from the system. In order to support its interactions, the program is embedded in a testbed that simulates its *environment.* On the other hand, as in model checking the execution ranges over an entire graph of possible behaviors as opposed to a suite of linear test cases. In the optimal setting, each visited state is marked to avoid redundant explorations of the same state and can be restored for backtracking to alternate executions. Furthermore, sources of variation in the execution, such as external events, scheduling or faults, are controlled to explore all alternatives.

The rationale behind simulation-based verification is to take the advanced state space exploration algorithms and optimizations developed in the field of model checking and apply them to the testing of real code. By doing so, we avoid the need for developing a separate model for verification purposes, and more importantly for scrutinizing each reported violation to assess whether it relates to the real system or to a modeling inaccuracy. Of course, simulation-based verification will in general be significantly less efficient and scalable than model checking an abstract model of the same program. However, it should be seen as an evolutionary improvement to traditional testing approaches, with important potential gains in scalability, automation and flexibility.

To enable their controlled execution, instrumentation is introduced in both the analyzed program and its environment. To perform a true model-checking search, the tool should be capable of iterating over all alternate events at each state, backtracking to previously visited states, and detecting states that produce the same behavior. Some of these capabilities may be supported only partly or not at all, depending on the nature of the different components in the testbed and the needs and trade-offs of the analysis being performed. For example, a complex piece of software may provide checkpointing capabilities on top of which backtracking can easily be built; however, state equivalence might require an analysis of internal data structures that is either too complex, computationally expensive, or infeasible due to the proprietary nature of the software. In addition, this analysis may not be worthwhile because equivalent states will seldom be reached anyway. Even if true backtracking is not available, it can still be simulated by re-playing the sequence up to the desired state. This reduces the exploration to a suite of sequential test cases, but the additional automation and flexibility in search strategies can still be beneficial.

With these capabilities, the program and its testbed constitute a *virtual machine* that embodies a fully controllable state machine, whose state space can be explored according to different strategies, such as depth-first search, breadth-first search, heuristic-based guided search, randomized search, pattern-guided search or interactive simulation. The environment portion of the testbed is typically restricted to a well-defined range of possible scenarios in order to constrain the state space within tractable bounds.

The principle behind simulation-based verification can be traced back to Godefroid's seminal work on the VeriSoft tool [4]. VeriSoft instruments a C program with explicit stop and choice points, to allow its controlled execution over a range of traces. Verisoft makes no attempt at capturing the state of the verified program; instead, trace re-play is used to simulate backtracking. This approach is referred to as *state-less model checking*. The original contribution presented in this paper pertains to the application of that general idea to the LPF framework and tool as well as the experimental results on a real-world application. The same principle can be found in Visser and Havelund's Java PathFinder 2 [10] with a twist: a complete Java virtual machine, optimized for verification, has been built rather than merely instrumenting the Java code to be analyzed.

The models used in model-based autonomy are typically abstract and concise enough to be amenable to formal analysis. Indeed, the *Livingstone-to-SMV translator* [9] is another verification tool for Livingstone applications, focusing exclusively on the Livingstone model but allowing true exhaustive analysis through the use of symbolic model checking. Nevertheless, a good model is a necessary but insufficient condition for a suitable model-based controller. Additional factors may compromise proper operation, such as problems in the engine itself or incomplete diagnosis heuristics used to achieve desired response times. The principles and techniques we discuss here aim to supplement high-level formal analysis with efficient and flexible tools designed to scrutinize the behavior of the actual program.

3 Livingstone

As the name implies, LPF has focused on the *Livingstone model-based diagnosis system* [12]. Generally speaking, *diagnosis* involves observing the input (commands, actuators) and output (observations, sensors) of a physical system to estimate its internal state, and in particular detect and identify faulty conditions that may have occurred. In *model-based diagnosis*, this state estimate is inferred from a model of the different components of the physical plant and their interactions, both under nominal and faulty conditions. Livingstone uses a qualitative finite model. This allows the use of an efficient inference engine based on propositional logic to perform the diagnosis.

A Livingstone model defines a collection X of discrete *variables*, or *attributes*, representing the state of the various components of the physical system.[3] Each attribute $x \in X$ ranges over a finite domain D_x. A *state* s of model associates to each $x \in X$ a value $s(x) \in D_x$. The *command* and *observable* attributes $O \subset X$ capture the visible interface of the system; the *mode* attributes $M \subset X$, one for each individual component, denote the hidden state to be inferred. The domain D_m of a mode attribute m is partitioned into *nominal modes* and a (possibly empty) subset of *fault modes* $F_m \subseteq D_m$. The model also sets constraints between these attributes, both within a state and across transitions. Nominal transitions result from particular command values while fault transitions are spontaneous. Each fault mode $f \in F_m$ has a *rank* $\rho(f)$ estimating its probability (e.g. rank 4 for probability 10^{-4}).

[3] The model may also cover aspects of the surrounding environment, that do not strictly belong to the controlled system but are relevant to diagnosis.

As an example, the following Livingstone model fragments describe a valve component, extracted from the PITEX model that we used for our experiments (see Section 5). Note that Livingstone is supported by a graphical modeling environment, that generates part of this code automatically.

```
class ventReliefValve
  ventLine ventLineIn;
  thresholdValues pneumaticLineIn;
  ventLineTemperature ventLineOut;
  // ... more attributes
  private enum ModeType {nominal, stuckOpen, stuckClosed};
  private ModeType mode;
  stateVector [mode];
  {
    if (valvePosition = closed) {
       ventLineOut.ambient.upperBound = belowThreshold &
       ventLineOut.ambient.lowerBound = aboveThreshold &
       ventLineIn.flow.sign = zero;
    }
    // ... more constraints
    switch (mode) {
      case nominal:
        if (pneumaticLineIn = aboveThreshold) valvePosition = open;
        if (pneumaticLineIn = belowThreshold) valvePosition = closed;
      case stuckOpen:
        valvePosition = open;
      case stuckClosed:
        valvePosition = closed;
    }
  }
  failure stuckOpen(*, stuckOpen, lessLikely) { }
  failure stuckClosed(*, stuckClosed, likely) { }
```

Livingstone performs diagnosis by receiving observations (that is, values of commands and observables) and searching for mode values (possibly including faults) that are consistent with these observations. The result is a list of *candidates* $(c_1, \ldots, c_n)$, where each candidate c assigns a value $c(m)$ to every mode attribute $m \in M$ and has a rank $\rho(c)$ resulting from the faults leading to that candidate. For verification purposes, we will say that a candidate c *matches* a state s, written $c \approx s$, if they have the same mode values, and that c *subsumes* s, written $c \sqsubseteq s$, if faults of c are also faults of s. Formally, $c \approx s$ iff $\forall m \in M\ c(m) = s(m)$, and $c \sqsubseteq s$ iff $\forall m \in M . c(m) \in F_m \Rightarrow c(m) = s(m)$.

Livingstone's best-first search algorithm returns more likely, lower ranked candidates first $(\rho(c_i) \leq \rho(c_{i+1}))$. In particular, when the nominal case is consistent with Livingstone's observations, the empty (i.e. fault-free) candidate (of rank 0) is generated. The search is not necessarily complete: to tune response time, Livingstone has configurable parameters that limit its search in various ways, such as the maximum number of rank of candidates returned.

The first generation of Livingstone (in Lisp) flew in space as part of the Remote Agent Experiment (RAX) demonstration on Deep Space 1 [7]. Livingstone 2 (or L2) has been re-written in C++ and adds temporal trajectory tracking. A utility program, `l2test`, provides a command line interface to perform all L2 operations interactively. `l2test` commands can be stored in *L2 scenario* files for batch replay. A third generation supporting more general constraint types (hybrid discrete/continuous models) is under development.

4 Livingstone PathFinder

The *Livingstone PathFinder* (LPF) program is a simulation-based verification tool for analyzing and verifying Livingstone-based diagnosis applications. LPF executes a Livingstone diagnosis engine, embedded into a simulated environment, and runs that assembly through all executions described by a user-provided scenario script, while checking for various selectable error conditions after each step.

The architecture of the LPF tool is depicted in Figure 1. The testbed under analysis consists of the following three components:

- *Diagnosis*: the diagnosis system being analyzed, based on the Livingstone diagnosis *engine*, interpreting a *model* of the physical system.
- *Simulator*: the simulator for the physical system on which diagnosis is performed. Currently this is a second Livingstone engine interpreting a model of the physical system. The models used in the Diagnosis and the Simulator are the same by default but can be different.
- *Driver*: the simulation driver that generates commands and faults according to a user-provided scenario script. The scenario file is essentially a non-deterministic test case whose elementary steps are commands and faults.

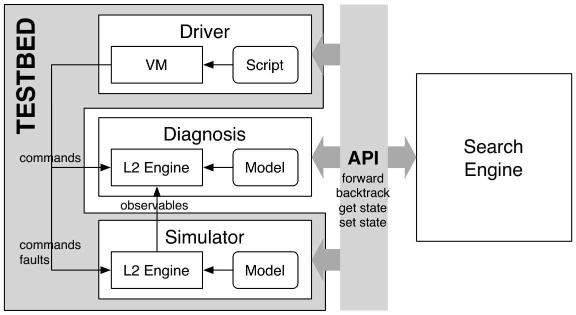

Fig. 1. Livingstone PathFinder Architecture

From a testing standpoint, the Diagnosis is the *implementation under test* (IUT), while the Driver and Simulator constitute the *test harness*. All three components are instrumented so that their execution can be single-stepped in both forward and backward directions. Together, these three elements constitute a non-deterministic state machine, where the non-determinism comes mainly from the scenario script interpreted by the Driver.[4] The *Search Engine* controls the order in which the tree of possible executions of that state machine is explored. In each forward step, LPF performs the following cycle of operations:

```
event := get next step from the Driver
apply event to the Simulator
if    event is a command
```

[4] LPF can also deal with non-determinism in the Simulator.

```
then   notify event to the Diagnosis
else   do nothing (Faults are not directly visible to Diagnosis)
obs := get updated observations from the Simulator
notify obs to the Diagnosis
ask Diagnosis to update its candidates
```

This cycle is repeated along all steps covered by the scenario, according to the chosen search strategy. Before each cycle, the current state is saved (using Livingstone's built-in checkpointing capability) so it can be later restored to explore alternate routes. User-selectable error conditions, such as consistency between the diagnosis results and the actual state of the Simulator, are checked at the end of each cycle, and a trace is reported if an error is detected. Since the cycle needs to be repeatedly rolled and unrolled by the backtracking search, it cannot simply be implemented as a program iteration, but instead needs to be simulated using non-trivial bookkeeping of the search trajectory.

LPF bears some resemblance to *conformance testing*, in that it checks whether the Diagnosis, seen as an implementation, "conforms" to the Simulator, seen as a specification. The "conformance" relation, however, is quite different from classical conformance relations that require the implementation to behave similarly (in some specific precise sense) to the specification (see e.g. [2]). Here, the relation is one of observability, i.e. that the implementation (Diagnosis) can *track* the state of the specification (Simulator). Even if Simulator and Diagnosis are "equivalent", in the sense that they operate on identical models of the controlled system, Diagnosis may still fail to "conform", because the observation it receives or its own computing limitations prevent accurate tracking. This is indeed the situation we consider by using the same model in both the Simulator and the Diagnosis.

4.1 Scenario Scripts

The driver constrains the possible executions to a user-provided combination of commands and faults, defined in a *scenario script.* This is essential to reduce the state space to a tractable size: the state space of the PITEX Livingstone *model*, used in the experiments reported later in this paper, is on the order of 10^{55} states. Although model states do not necessarily map into simulator or diagnosis states, this still gives a fair estimate of the orders of magnitude.

The scenario script is essentially a non-deterministic program whose elementary instructions are commands and faults. It can be viewed as an extended test case (or test suite) representing a tree of executions rather than a single sequence (or a set thereof). Scenarios are built from individual commands and faults using sequential, concurrent (interleaved) and choice statements, according to the following syntax:

stmt ::= "*event*" ; | { *stmt** } | `mix` *stmt* (`and` *stmt*)* | `choose` *stmt* (`or` *stmt*)*

These operators have their classical semantics from process algebras such as CSP [5], that we will not repeat here. For example, the following scenario defines a sequence of three commands, with one fault chosen among three 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 this sequence/choice pattern. More configurable ways to generate scenarios from models or other external sources is an interesting direction for further work.

```
mix {
    "command test.breaker.cmdIn = on" ;
    "command test.breaker.cmdIn = off" ;
    "command test.breaker.cmdIn = on" ;
} and
    choose "fault test.bulb.mode = blown" ;
    or "fault test.bulb.mode = short" ;
    or "fault test.meter.mode = broken" ;
```

4.2 Error Conditions

In each state along its exploration, LPF can check one or several *error conditions* among a user-selectable set. In addition to simple consistency checks, LPF supports the following error conditions: given a list of candidates $(c_1, \ldots, c_n)$ and a simulator state s,

- *Mode comparison* checks that the best (lowest-ranked) candidate matches the Simulator state, i.e. $c_1 \approx s$,
- *Candidate matching* checks that at least one candidate matches the Simulator state, i.e. $\exists k . c_k \approx s$, and
- *Candidate subsumption* checks that at least one candidate subsumes the Simulator state, i.e. $\exists k . c_k \sqsubseteq s$.

where $\approx$ and $\sqsubseteq$ were defined in Section 3. These conditions constitute three successive refinement steps with the notion 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 since the injected fault may not be the most likely one. Alternatively, *candidate matching* considers all candidates. Even so, a fault may often remain unnoticed without causing immediate harm, as long as its component is not activated: for example a stuck valve will not be detected until one tries to change its position. Experience reveals this to be a frequent occurrence, causing a large proportion of spurious error reports. In contrast, *candidate subsumption* only reports cases where none of the diagnosed fault sets are *included* in the actual fault set. For example, the empty candidate is subsumed by any fault set and thus never produces an error using this condition. While this will not detect cases where diagnosis misses harmful faults, it will catch cases where the wrong faults are reported by the diagnosis engine. This condition has provided the best results thus far, as further discussed in Section 5.

4.3 Simulators in LPF

The modular design of LPF accommodates 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 may appear to be 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. In order to address this concern, we have already been studying integration of NASA Johnson Space Center's CONFIG simulator system [3]. Note that higher fidelity simulators are likely to be less flexible and efficient; for example, they may not have backtracking or checkpointing.

4.4 Search Strategies

LPF currently supports two alternative state space exploration algorithms. The first is a straightforward depth-first search. The second performs a *heuristic (or guided) search*. Heuristic search uses a *heuristic* (or *fitness function*) to rank different states according to a given criterion and explores them in that order. Provided that a good heuristic can be found, heuristic search can be used to automatically orient the search over a very large state space towards the most critical, suspicious, or otherwise interesting states. This offers two advantages: The first is that errors are likely to be found earlier on and the second is the production of shorter counterexample paths.

The first heuristic uses the execution depth, i.e. the number of steps to that state. This heuristic results in performing a *breadth-first search*. Although this exploration strategy does not use any application knowledge to improve the search, its main benefit over depth-first search is in providing the shortest counter-examples.

The second heuristic, called *candidate count*, uses the number of candidates generated by Diagnosis. The intuition is that executions with a low number of diagnosis candidates are more likely to lead to cases where no correct candidate is reported.

4.5 Implementation Notes

Backtracking of the Diagnosis and Simulator take advantage of checkpointing capabilities present in Livingstone. Using built-in checkpointing capabilities provide better performance than extracting and later re-installing all relevant data through the API of Livingstone engine.

LPF also offers a number of useful auxiliary capabilities, such as generating an expanded scenario tree, counting scenario states, exploring the simulator in isolation (no diagnosis), generating a default scenario, and producing a wide array of diagnostic and debugging information, including traces that can be replayed using the `l2test` command-line interface or in Livingstone simulation tools. LPF is written in Java and makes extensive use of Java interfaces to support its modular architecture. The core source base consists of 47 classes totaling 7.4K lines. It has been successfully run on Solaris, Linux and Windows operating systems.

Livingstone PathFinder is built in a modular way, using generic interfaces as its main components. The Simulator and Diagnosis components implement generic interfaces, that capture generic simulation and diagnosis capabilities but remain independent of Livingstone-specific references. The top-level class that encompasses all three components of the testbed and implements the main simulation loop discussed in Section 4 only uses the generic abstractions provided by these interfaces. This provides a flexible, extensible framework for simulation-based verification of diagnosis applications beyond the purely Livingstone-based setup (both for diagnosis and simulation) shown in Figure 1. Both the Simulator and Diagnosis modules are replaceable by alternative implementations, not necessarily model-based, provided those implementations readily provide the corresponding generic interfaces.

Livingstone PathFinder is not available for download but can be obtained free of charge under a NASA licensing agreement. Please contact the authors if interested.

5 Experimental Results

The Livingstone diagnosis system is being considered for Integrated Vehicle Health Maintenance (IVHM) to be deployed on 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 [1, 8], and LPF has been successfully applied to the PITEX model of X-34. This particular Livingstone model consists of 535 components and 823 attributes, 250 transitions, compiling to 2022 propositional clauses.

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

- The *PITEX baseline scenario* combines one nominal and 29 failure scenarios, derived from those used by the PITEX team for testing Livingstone, as documented in [8]. This scenario covers 89 states.
- The *random scenario* covers a set of commands and faults combined according to the sequence/choice pattern illustrated in Section 4.1. This scenario is an abbreviated version of the one automatically generated by LPF, and covers 10216 states.

We will first discuss scalability and compare different error conditions using depth-first search, then discuss the benefits of heuristic search and finally illustrate two different errors found during the experiments.

5.1 Depth-First Search

Our initial experiments involved searching the entire state space, letting the search proceed normally after detecting an error, and reporting any additional errors. We used the depth-first algorithm in this case, since heuristic search offers no advantage when searching through all states of the scenario. Table 1 summarizes our depth-first search statistics. It shows that LPF covered the *PITEX baseline* and *random* scenario at an average rate of 51 states per minute (on a 500MHz Sun workstation).

When searching for *candidate matching*, LPF reported an excessive number of errors, most of which were *trivial*, in the following sense: when a fault produces no immediately observable effect, Livingstone diagnosis does not infer any abnormal behavior and reports the empty candidate. For example, although a valve may be stuck, the fault will stay unnoticed until a command to that valve fail to produce the desired change of position. While these errors are indeed missed diagnoses, experience

Table 1. Depth-first search statistics

scenario	search[1]	condition[2]	errors	non-trivial	states	states/min
baseline	all	CM	27	4	89	44
baseline	all	CS	0	0	89	67
random	all	CM	9621	137	10216	51
random	all	CS	5	5	10216	52
random	one	CS	1	1	8648	49
random	min	CS	2	2	8838	44

[1] all=search all errors, one=stop at first error, min=search for minimal trace

[2] CM=candidate matching, CS=candidate subsumption

shows that they are in most cases expected and do not constitute significant diagnosis flaws. In particular, while searching for *candidate matching* error condition violations the *baseline* scenario produced 27 errors of which only 4 were not trivial. The same search over the *random* scenario resulted in 9621 errors of which 137 were non-trivial.

In contrast, verification over the *random* scenario using the *candidate subsumption* error condition reported a total of 5 errors, all of which were non-trivial (the *baseline* scenario reported no errors). Indeed, the *candidate subsumption* error condition will only activate if all candidates report some incorrect fault, but not when faults are missed. In particular, the empty candidate subsumes all cases and will, by definition, never prompt an error.

For illustration purposes, the last two rows in Table 1 show results using two alternative termination criteria: either stopping at the first error (*one*), or continue searching only for shorter error traces (*min*). *Search one* finds the first error (at depth 16) only after 8648 states, i.e. after it has already covered a large part of the state space. As a result, the benefit of using *search min* is rather marginal in this case; it finds a second error, at depth 3, by exploring only 190 additional states.

5.2 Heuristic Search

In this section, we illustrate the benefits of using heuristic search, using the large *random* scenario. In this case, we are interested in how quickly the search can detect an error, and thus stop at the first error found. The fitness function types used were *breadth-first search* and *candidate count*, both described in Section 4.4. We tried both the *candidate subsumption* and *candidate matching* error conditions.

The results are summarized in Table 2, including depth-first search for comparison purposes. They show that, using breadth-first search, LPF mapped through only 154 states before detecting a candidate subsumption error at depth 3, compared to 8648 states to depth 16 using depth-first search. The candidate-count heuristic also explores 154 states, although it goes a little deeper to depth 5. For the less selective candidate-matching error condition, LPF finds an error after just 4 states using candidate-count or breadth-first search, compared to 17 states with depth-first search. As expected, heuristic search (both breadth-first search and the candidate-count heuristic) detected the error significantly faster than depth-first search. These results illustrate that heuristic search can save a great deal of time by skipping large parts of the search space where errors are less likely to be found.

Table 2. Comparison of Heuristic vs. depth-first search

strategy	max. depth	condition[1]	time	states	states/min
DFS	16	CS	02:55:38	8648	49
BFS	3	CS	00:03:56	154	38
CC	5	CS	00:03:42	154	38
DFS	16	CM	00:00:15	17	68
BFS	2	CM	00:00:13	4	20
CC	1	CM	00:00:11	4	24

[1] See Table 1

5.3 Example Scenario 1

Our first example involves the *PITEX baseline* scenario, and was performed with an earlier version of L2 in which, as it turns out, the checkpointing functionality used by LPF was flawed. The scenario involves a double fault, where a valve `sv31` (on a liquid oxygen venting line) gets stuck and a micro-switch sensing the position of that valve fails at the same time. The sequence of events is as follows:

1. A command `open` is issued to `sv31`.
2. A command `close` is issued to `sv31`.
3. The open microswitch of `sv31` breaks.
4. `sv31` fails in stuck-open position.

The microswitch fault remains undetected (Livingstone still reports the empty candidate after event 3), however the stuck valve causes a fault diagnosis: after event 4, Livingstone reports the following candidates (the number before the # sign is the time of occurrence of the presumed fault):

```
Candidate 0)
  5#test.vr01.modeTransition=stuckOpen:2
Candidate 1)
  -#test.vr01.modeTransition=stuckClosed:2
Candidate 2)
  -#test.sv31.modeTransition=stuckClosed:3
Candidate 3)
  -#test.vr01.modeTransition=stuckOpen:2
  3#test.vr01.modeTransition=stuckClosed:2
Candidate 4)
  5#test.forwardLo2.rp1sv.modeTransition=unknownFault:5
Candidate 5)
  -#test.forwardLo2.rp1sv.modeTransition=unknownFault:5
Candidate 6)
  3#test.forwardLo2.rp1sv.modeTransition=unknownFault:5
Candidate 7)
  5#test.sv31.modeTransition=stuckOpen:5
Candidate 8)
  -#test.sv03.sv.modeTransition=stuckClosed:3
  3#test.sv03.openMs.modeTransition=faulty:3
Candidate 9)
  -#test.sv03.sv.modeTransition=stuckClosed:3
  -#test.sv03.openMs.modeTransition=faulty:3
```

At this point, LPF reports a candidate-matching error, indicating that none of the Livingstone candidates match the modes from the simulator. The fault is detected (otherwise no faulty candidates would be generated), but incorrectly diagnosed (note that candidate 7 *subsumes*, but does not *match*, the actual faults). At this point the data seems to suggest that Livingstone is not properly generating all valid candidates.

To further interpret and confirm the results reported by LPF, we ran `l2test` on the L2 scenario associated with this error. The following list of candidates were generated, which differs from those obtained above via LPF:

```
Candidate 0)
  5#test.vr01.modeTransition=stuckOpen:2
Candidate 1)
  4#test.vr01.modeTransition=stuckOpen:2
Candidate 2)
  4#test.forwardLo2.modeTransition=unknownFault:5
Candidate 3)
  3#test.forwardLo2.modeTransition=unknownFault:5
Candidate 4)
  4#test.sv31.sv.modeTransition=stuckOpen:5
Candidate 5)
  5#test.forwardLo2.modeTransition=unknownFault:5
Candidate 6)
  -#test.forwardLo2.modeTransition=unknownFault:5
Candidate 7)
  5#test.sv31.sv.modeTransition=stuckOpen:5
```

Although none of these match the actual faults either (i.e. there is indeed a candidate-matching error in this state), the difference also shows that the results from LPF were flawed. Further analysis revealed that the discrepancy originated from a checkpointing bug. In particular, Livingstone did not properly restore its internal constraint network when restoring checkpoints, resulting in a corrupted internal state and incorrect diagnosis results. With our assistance, the error was localized and resolved in a new release of the Livingstone program.

5.4 Example Scenario 2

The second example considers one of the five errors reported using Candidate subsumption on the Random scenario. It involves a solenoid valve, `sv02`, which sends pressurized helium into a 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. The following sample output lists the candidates reported by Livingstone after the fault occurs in `sv02`:

```
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 injected fault, `test.sv02.rp1sv.mode=stuckOpen`, is detected (otherwise no faulty candidates would be generated) but incorrectly diagnosed: none of these candidates matches or subsumes the correct fault. 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 fault mode. The L2 scenario corresponding to this error was replayed in `l2test` and produced identical results, confirming the validity of the results from LPF. Further analysis by application specialists revealed that fault ranks in the X-34 model needed re-tuning, which resolved the problem.

6 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.

Although the experiments have so far been performed only by the LPF development team, Livingstone specialists from the PITEX project have shown great interest in the results we obtained. In comparison, their verification approach is based on running a fixed, limited set of test cases, as detailed in [8]. LPF shows great potential for radically expanding the number of tested behaviors, with a modest additional effort from the users. Our next stage is to put the tools in the hands of the developers. In this perspective, we have also been adding a graphical user interface and integrating LPF into the Livingstone modeling and simulation tool.

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 contributing 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 benefit from 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.

Future work includes support for determining state equivalence, to allow pruning the search when reaching a state equivalent to an already visited one. The potential benefits from this approach remain to be assessed: as Livingstone retains data not only about the current step but previous ones as well, cases where equivalent states are reached through different executions may be very infrequent. As an alternative, experiments with weaker or approximate equivalences may be performed to reduce the search space, at the risk of inadvertently missing relevant traces.

We are also currently adapting LPF to MIT's Titan model-based executive [11], 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. We are also grateful to reviewers of previous versions of this article for their helpful comments. All trademarks used are properties of their respective owners.

References

1. A. Bajwa and A. Sweet. The Livingstone Model of a Main Propulsion System. In *Proceedings of the IEEE Aerospace Conference*, March 2003.
2. E. Brinksma. A Theory for the Derivation of Tests. In: S. Aggarwal, K. Sabnani, eds., *Protocol Specification, Testing and Verification, VIII.* North-Holland, Amsterdam, 1988, ISBN 0-444-70542-2, 63–74.
3. 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).
4. P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 174-186, Paris, January 1997.
5. C. A. R. Hoare.*Communicating Sequential Processes.* Prentice-Hall, 1985.
6. A.E. Lindsey and C. Pecheur, Simulation-Based Verification of Livingstone Applications. *Workshop on Model-Checking for Dependable Software-Intensive Systems*, San Francisco, June 2003.
7. N. Muscettola, P. Nayak, B. Pell and B. Williams. Remote Agent: To Boldly Go Where No AI System Has Gone Before. *Artificial Intelligence*, vol. 103, pp. 5-47, 1998.
8. C. Meyer, H. Cannon, Propulsion IVHM Technology Experiment Overview, In *Proceedings of the IEEE Aerospace Conference*, March 8-15, 2003.
9. C. Pecheur, R. 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. Lecture Notes in Computer Science, vol. 1871, Springer Verlag.
10. W. Visser, K. Havelund, G. Brat and S. Park. Model Checking Programs. In *Proceedings of the IEEE International Conference on Automated Software Engineering*, pages 3-12, September 2000.
11. B. Williams, M. Ingham, S. Chung and P. Elliot, Model-based Programming of Intelligent Embedded Systems and Robotic Space Explorers. In *Proceedings of the IEEE Modelings and Design of Embedded Software Conference*, vol. 9, no.1 pp. 212-237, 2003.
12. B. Williams and P. Nayak, A Model-based Approach to Reactive Self-Configuring Systems. In *Proceedings of the National Conference on Artificial Intelligence*, vol. 2, pp. 971-978, August 1996.