

Seeing the Invisible: Embedding Tests in Code That Cannot be Modified

Owen O'Malley*

QSS Inc., NASA Ames Research Center, Moffett Field, CA, 94035

Masoud Mansouri-Samani† and Peter Mehlitz‡

Computer Sciences Corporation, NASA Ames Research Center, Moffett Field, CA, 94035

and

John Penix§

NASA Ames Research Center, Moffett Field, CA, 94035

The difficulty of characterizing and observing valid software behavior during testing can be very difficult in flight systems. To address this issue, we evaluated several approaches to increasing test observability on the Shuttle Abort Flight Management (SAFM) system. To increase test observability, we added probes into the running system to evaluate the internal state and analyze test data. To minimize the impact of the instrumentation and reduce manual effort, we used Aspect-Oriented Programming (AOP) tools to instrument the source code. We developed and elicited a spectrum of properties, from generic to application specific properties, to be monitored via the instrumentation. To evaluate additional approaches, SAFM was ported to Linux, enabling the use of gcov for measuring test coverage, Valgrind for looking for memory usage errors, and libraries for finding non-normal floating point values. An in-house C++ source code scanning tool was also used to identify violations of SAFM coding standards, and other potentially problematic C++ constructs. Using these approaches with the existing test data sets, we were able to verify several important properties, confirm several problems and identify some previously unidentified issues.

I. Introduction

VERIFICATION and validation of human-rated software systems is a challenging and expensive task. While there are many methods and tools which can help select and generate test cases, the problem of characterizing and observing valid program behavior is often underestimated.

The standard approach to addressing this issue is to perform verification from the inside-out: starting with unit testing and ending with end-to-end system testing. This approach provides increased visibility into subsystems, but introduces issues as to whether the unit test scenarios correspond to well to the scenarios the unit will experience when integrated. Restricting unit tests to only expected scenarios often requires extensive modeling of the environment around the unit. While testing the unit on a wider range of inputs than anticipated can increase robustness, most often there are simply too many input combinations to test without some restriction on the expected environment.

In an effort to address this issue, we evaluated several approaches to increasing test observability. The evaluation was done as a “shadow” verification effort for the Shuttle Abort Flight Management (SAFM) system, a safety critical application being developed using C++. SAFM evaluates the potential abort options for the Space Shuttle under various contingencies and provides abort recommendations to the crew. Because this was a shadow effort, it

* Senior Computer Scientist, Robust Software Engineering Area, Intelligent Systems Division, M/S 269-2

† Computer Scientist, Robust Software Engineering Area, Intelligent Systems Division, M/S 269-2

‡ Senior Computer Scientist, Robust Software Engineering Area, Intelligent Systems Division, M/S 269-2

§ Computer Scientist, Robust Software Engineering Area, Intelligent Systems Division, M/S 269-2

was very important to not interfere with the main-stream development activities. This led to the requirement that our test instrumentation methods should not require manual modification of the code – specifically in the case where the instrumentation would need to be inserted into new releases.

Adding instrumentation to an application can be problematic because the instrumentation may change the way the software functions, resulting in invalid test results. The goal of this experiment was to evaluate different instrumentation approaches and platforms and see whether they could identify problems without interfering with the software – that is, if they could detect and isolate errors which were also errors in the non-instrumented system. This would indicate that the tools would be effective as initial test platforms to help flesh out problems before proceeding to more high-fidelity testing. We used an Aspect-Oriented Programming (AOP)¹ approach to help automate the insertion of checks into the code.

To enable intermediate test results checking, we needed a set of properties of the intermediate states of the software that should be satisfied by any test. We used a spectrum of properties, from a few generic properties such as "avoid division by zero," to several application specific properties, such as "the size of the backstep should always be between -0.01 and -2.0." The application specific properties were elicited from the development and testing organization using an iterative process.

In addition to inserting checks with AOP, we also applied the following tools to check some of these properties:

- gcov for measuring test coverage
- Valgrind for detecting heap and uninitialized memory errors
- Linux libraries for finding non-normal floating point values
- Dingo, a C++ source code scanning tool, for identifying potential coding issues

In the next section, we provide a brief overview of SAFM and the process used during development and testing. We then discuss how we identified additional properties to test and extended the test environment to check these properties. Finally, we provide technical descriptions of the tools that were used during this effort and provide a summary of the results.

II. Overview of the Shuttle Abort Flight Management system

A. Shuttle Abort Flight Management

To perform shuttle abort flight management, SAFM evaluates potential abort options for the Space Shuttle during various contingencies and provides abort recommendations to the crew. SAFM provides abort performance assessment during the powered flight phase (while the space shuttle main engine is firing) and landing site evaluation and monitoring during the glided flight phases. SAFM was developed as part of the Space Shuttle Cockpit Avionics Upgrade by NASA Johnson Space Center, United Space Alliance and General Dynamics Decision Systems.

B. SAFM Development and Testing

SAFM was developed using C++ and consists of 38K SLOC. It was developed over the course of 4 years and went through 11 incremental releases. Testing was done at the unit level and the application level using test data from generated with a space shuttle simulator. Application test results were evaluated against a SAFM requirements simulator developed by General Dynamics. The SAFM requirements simulator provided both input-output results as well as an assessment of requirements coverage achieved by a test. The requirements simulator was described as being very valuable for evaluating potential requirements changes before adding them to the Software Requirements Specification.

SAFM maintains and evaluates a set of viable hypothetical abort scenarios. Each scenario maintains a set of scenario-specific data, resulting in a relatively large internal state relative to its inputs and outputs. This presents a challenge to testing, because there is limited visibility to the scenario data at the application interface. Most of our effort was aimed at increasing visibility into the SAFM internal state to improve test validation.

III. Identifying critical requirements and design properties.

Adding instrumentation to an application is only adds value if you can identify interesting properties of the internal state of the system. Intermediate results are low-level derived requirements that usually depend on architecture and implementation choices. There is no well-defined accepted process for eliciting these properties. We used a combination of approaches in an attempt to be thorough and to understand which approaches were most useful. We classified the properties in terms of there general applicability versus there specific relevance to SAFM.

A. General Properties

The most general properties are low-level properties related to programming language and platform runtime issues such as memory leaks and array bounds overflows. These issues, while still very problematic in practice, are reasonably well known². We focused most of our time on identifying application specific properties.

B. Application Specific Properties

To elicit application specific properties, we hosted SAFM test lead at ARC for a week to discuss the SAFM architecture and to identify instrumentation points where critical intermediate values could be checked. The discussions were focused on three topics:

- requirements which the test lead felt were relatively under-tested,
- specific problems that had occurred during development which had proven difficult to detect with standard testing, and
- open-ended discussions about SAFM requirements guided by the questions “how can you tell that is working” and “what happens if that is broken.”

It was during these discussions that the general issue of SAFM’s large internal state, relative to its outputs, and the effect this has on testability was identified. The internal state was large in two ways. First, there were a large number of internal variables and second, there were many possible combinations of active scenarios that could be sequenced. To define properties over variables, we identified relationships between the internal variables that could be monitored and checked. These relationships could either be “invariant” meaning the always hold, or the might only hold at specific locations or during certain situations. Defining properties of valid sequences of scenarios was similar, except that it was not always possible to represent the property as a constraint between two variables at the same time or place. In these cases we need to define a “temporal” property, which relates variables across different states of the program execution³.

Some example application specific properties were:

1. All dynamic memory (de)allocation must happen during initialization (shutdown)
2. No scenario uses data from a parent scenario that was not “applicable” or “valid”
3. Only one abort type (Return To Launch Site, Trans-Atlantic and Abort To Orbit) is declared at a time
4. Only one “backstep” per “flyout” should be executed
5. “Flyouts” must run in order
6. Velocity lookup with one engine out is always less than or equal to that with two engines out

IV. Extending the SAFM test infrastructure

A. SAFM build & test environment at ARC

To perform a shadow verification effort, the SAFM application test environment was duplicated at NASA Ames Research Center. This included the SAFM test harnesses, requirements simulator, and test data used by the development and test team, as well as the source code. In addition, some minor changes were made to ported the system to enable the subsystem test driver to run under Unix (e.g. Linux and Mac OS) and added additional testing flexibility. In this environment we could apply several additional testing tools that were not being used by the development team: Memcheck (array bounds and memory leaks), gcov (statement coverage and counts) and

Kcachegrind (performance measurements). All of the testing experiments made use of the test data used by the development and test team that was generated by a space shuttle simulator at JSC.

B. Gathering data on existing test coverage

We used test adequacy coverage metrics as one method for measuring the impact of our testing extensions. To provide a baseline for comparison, we measured the current coverage of source code by the test suite. The test suite was developed to provide coverage of the requirements, not the code, so it was not expected to provide 100% coverage. To measure test coverage we used gcov, a test coverage program that comes with the GNU CC compiler. The current test suite was reported to cover 83% of the lines source code.

V. Inserting instrumentation for property checking

A. Developing checks for properties

For each property, we had to determine exactly what relationship between program variables should be checked and when/where during execution the checks could be placed. In general, it would be beneficial to design an application with explicit checkpoints where the system state is expected to be stable⁴. For SAFM, identifying these locations required the expertise of the testing team. Examples of how properties were mapped to application variables are shown in the following table:

Property	Check
All dynamic memory allocations happen during startup and shutdown	Overloaded operator new and delete. Used control flags to ensure they weren't called at the wrong time
No scenario uses data from a parent scenario that was not applicable or valid	Insert checks to applicability and valid flags of parent scenarios before scenario execution
Only one abort type (Return To Launch Site, Trans-Atlantic and Abort To Orbit) is declared at a time	RTLS and TAL abort flags in powered flight scenarios must be mutually exclusive. RTLS, TAL and ATO abort flags must be mutually exclusive across the whole of SAFM.
Only one "backstep" per "flyout" should be executed	Instrumented flyout loop to check counter
"Flyouts" must run in order	Inserted check based on a table from the SRS indicating acceptable ordering of scenarios.
Velocity lookup with 1 engine out is always less than or equal to that with 2 engines out	Inserted a value comparison check in the table lookup function

All of these examples properties passed testing with violations when inserted into SAFM Version 9. However, when migrating to Version 11, the "flyouts must run in order" property began reporting violations. Analysis revealed that the cause was the addition of a new requirement which included a situation where the expected order of the flyouts was altered.

B. Inserting property checks

The primary instrumentation approach we used was to insert additional checks on internal data using Aspect-Oriented Programming (AOP) tools¹. AOP is a general software development approach which enable functions to be inserted into specified location in a program. This enables cross-cutting functionality (a.k.a. aspects) to be managed separately and then automatically distributed throughout the program text. By maintaining the instrumentation as aspects, we are able to separately maintain the test code and easily insert it into new SAFM releases.

AOP eased this task and made it less error-prone and time-consuming by providing several practical advantages. First, because the aspect code resides in its own set of source code files, it was not necessary to modify the SAFM source code to support testing. For example, there were several situations where the existing test harness required the re-insertion of test code or modification to integrate with the code for the test environment. Being able to automatically "weave in" the instrumentation makes the testing infrastructure code much easier to maintain,

especially across successive code revisions. Also, having the instrumentation defined in aspect files allowed us to easily switch between the instrumented and un-instrumented code or to select arbitrary groups of instrumentations to be inserted. This provides added ease and flexibility in supporting a variety of test harness configurations during testing. In some cases, the instrumentation for a single property would cut across several source files. In this case the instrumentation qualifies as a cross-cutting concern of the type that the AOP tools were designed to handle.

An additional benefit of using AOP tools was that it provided visibility into the state of objects in the system. One challenge of constructing a test environment for object-oriented software is that the programming language features which promote good design by hiding internal state of objects from other objects in the system can limit what the test harness objects can see. One obvious solution to this is to integrate the test code into the objects so there is greater visibility. However, the result is that the test code is spread throughout the system and is more difficult to maintain and remove if necessary. Another approach is to use language features, such as "friends" in C++, which bypass the features promoting good design. While it might be argued that good design is not as critical for test code as it is for application code, completely disabling the visibility restrictions results in an unmanaged environment, an is likely to increase the time, cost and reliability of testing.

VI. Additional Verification Tool Evaluation

A. Linux

One of the general properties that was identified was to avoid division by zero. The SAFM test platform used for development was configured to adhere to the IEEE Floating Point Standard which specifies that a divide by zero does not cause a runtime exception, but returns the special value NaN (not a number). This makes it difficult to identify divide by zero situations that occur when calculating values for internal variables. Under Linux, we incorporated the `trapfpe.c` library to cause floating point libraries to cause a program trap. This enabled us to detect a divide by zero in one of the test cases that was not being detected on the test platform. We also detected an overflow problem in superfluous test data which had no impact on testing.

B. Valgrind

Valgrind⁴ is an open-source program monitoring framework which operates on x86 binaries compiled for Linux. The basic framework includes a suite of tools providing defect detection and profiling capabilities. Valgrind works by inserting instrumentation into the binary and then executing it on a simulated CPU (a virtual machine) which interpret the instrumentation.

Memcheck is a Valgrind tool which targets memory management errors such as memory leaks, use of uninitialized memory and improper use of standard library memory management functions. To accomplish this, Memcheck instruments all reads and writes of memory. This instrumentation results in a slow down of about 20-40x. Execution of the test suite under Memcheck did not reveal any memory leaks in the application. We did identify one memory leak in the test driver when it was used to drive multiple runs of the SAFM application without allowing normal application termination.

C. Automated code inspections

Automated code inspection technology involves the use of tools to check project-defined coding standards. The primary roles of coding standards are to establish consistent coding conventions across a project to improve readability and maintainability of the code and to restrict the use of specific coding patterns which may be error prone or difficult to maintain. There are several sets of industry defined coding standards as well as several popular books providing coding conventions^{5,6}

Coding standards checking is done statically - at compile time, before program execution - and is therefore not a program monitoring technology. However, the objective of many of the coding rules is to avoid program constructs and patterns which may cause errors that would be very difficult to detect or isolate at run time. For example, the object oriented features of C++ instruct the compiler to introduce various standard functions when they are not explicitly defined. If you were to (perhaps unintentionally) write code which requires one of these standard methods which is undefined, the compiler will happily generate this code for you; it will not complain that this function is not defined. To prevent this scenario, a commonly recommended coding convention is to provide definitions for the standard functions (single argument 'copy' constructors, address-of operators, etc...) but make them invisible outside the object class. If this is done, the compiler will complain if one of these methods is needed.

As part of the Cockpit Avionics Upgrade, SAFM was required to adhere to the CAU coding standards defined by United Space Alliance. These standards stated a number of rules ranging from naming conventions to conventions on ensuring some of the more sinister features of C++ are not accidentally invoked. One of the most subtle issues

that we found fell into the second category. Specifically, the coding standards required the “address of” operator (&) to be declared “private” in class definitions to prevent objects of other classes from obtaining the memory address of an object. Our standards checker found several places in the CAU system software where this operator is declared with an extra parameter, which makes it the bitwise-and operator instead for the address-of operator. Therefore, the desired restriction on taking the address of the object was not achieved.

VII. Summary and Conclusion

Using these approaches, we were able to verify several important properties and identify some issues that had not been identified by the engineering team. We were also able to provide additional data to confirm several problems found during standard testing. All of this was done using the same input test data that had already been used in formal testing; the issues had not been noticed because they were either not visible in the output data or were difficult to isolate and identify. Thus, increased observability of test execution was able to provide significant additional error-detection capabilities with existing test data. We are continuing to work with the SAFM team to make these tool and techniques more cost effective so that they can be used in next-generation autonomous flight system.

Acknowledgments

We would like to thank Tanya Lippencott, Matthew Walters and Mark Coats of General Dynamics C4 Systems for their technical support on SAFM and Howard Hu of NASA’s Johnson Space Center for technical support on the SAFM requirements and for making SAFM available to Ames. Support for the SAFM verification tools case study was provided by NASA’s Software Assurance Research Program funded by the NASA’s Office of Safety and Mission Assurance and managed by NASA’s Independent Verification and Validation Center in Fairmont West Virginia. Development of tools was provided by NASA’s Engineering for Complex Systems program.

References

- ¹ Robert Fillman, et. al, *Aspect Oriented Software Development*, Addison-Wesley Professional, 2004
- ² Zohar Manna and Amir Pnueli, *The Temporal Logic of Reactive and Concurrent Systems: Specification*, Springer, 1991
- ³ Peter Mehltz and John Penix, "Design for Verification: Using Design Patterns to Build Reliable Systems," Workshop on Component-Based Software Engineering at International Conference on Software Engineering, May 2003
- ⁴ Nicholas Nethercote and Julian Seward, "Valgrind: A Program Supervision Framework," *Electronic Notes in Theoretical Computer Science* 89 No. 2, 2003.
- ⁵ Herb Sutter and Andrei Alexandrescu, *C++ Coding Standards : 101 Rules, Guidelines, and Best Practices*, Addison-Wesley Professional, 2004
- ⁶ Scott Meyers, *Effective C++, Third Edition*, Addison-Wesley Professional, 2005