Development, Validation, and Demonstration of Technologies for the Health and Usage Monitoring Systems Airborne- and Ground-Based Automated Testing and System Functionality Partition

December 2009
Final Report

This document is available to the U.S. public through the National Technical Information Service (NTIS), Springfield, Virginia 22161.
NOTICE

This document is disseminated under the sponsorship of the U.S. Department of Transportation in the interest of information exchange. The United States Government assumes no liability for the contents or use thereof. The United States Government does not endorse products or manufacturers. Trade or manufacturer's names appear herein solely because they are considered essential to the objective of this report. This document does not constitute FAA certification policy. Consult your local FAA aircraft certification office as to its use.

This report is available at the Federal Aviation Administration William J. Hughes Technical Center’s Full-Text Technical Reports page: actlibrary.tc.faa.gov in Adobe Acrobat portable document format (PDF).
DEVELOPMENT, VALIDATION, AND DEMONSTRATION OF TECHNOLOGIES FOR THE HEALTH AND USAGE MONITORING SYSTEMS AIRBORNE- AND GROUND-BASED AUTOMATED TESTING AND SYSTEM FUNCTIONALITY PARTITION

When applied to Health and Usage Monitoring System (HUMS) software, automated testing and building environments have the potential to significantly reduce development costs by improving quality and adding automation to portions of the software certification/recertification process.

This report relates those technologies to HUMS in two ways: (1) by describing the benefits of automated testing, and how they apply to HUMS and (2) by recommending processes, environments, and function partitioning related to automated testing. Both ways are backed by GE Aviation’s own experiences in applying automated testing to HUMS on the Bell/Augusta Aerospace 609 Tiltrotor Aircraft BA609 HUMS program and by a demonstration project that went through the tool qualification of an automated testing framework.

This report investigates and summarizes current tools and the latest techniques and methods available to assist the automated building and testing of HUMS software. The results of an industry survey included an evaluation of how the software industry typically employs these tools and their approaches to integrating the tools into a usable environment. The report summarizes the current technology level of the research and makes recommendations for additional research.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXECUTIVE SUMMARY</td>
<td>xi</td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>2. TRENDS IN SOFTWARE DEVELOPMENT</td>
<td>1</td>
</tr>
<tr>
<td>3. INDUSTRY TECHNIQUES AND BEST PRACTICES AND THEIR APPLICATION WITHIN DO-178B</td>
<td>2</td>
</tr>
<tr>
<td>3.1 Iterations</td>
<td>2</td>
</tr>
<tr>
<td>3.2 Test-Driven Development</td>
<td>3</td>
</tr>
<tr>
<td>3.2.1 Overview</td>
<td>3</td>
</tr>
<tr>
<td>3.2.2 Graphical User Interface Testing</td>
<td>5</td>
</tr>
<tr>
<td>3.2.3 Embedded Software and Hardware Testing</td>
<td>5</td>
</tr>
<tr>
<td>3.3 Customer Tests</td>
<td>5</td>
</tr>
<tr>
<td>3.4 Documentation</td>
<td>6</td>
</tr>
<tr>
<td>3.4.1 Face-to-Face Communication</td>
<td>6</td>
</tr>
<tr>
<td>3.4.2 Design, Assessments, and Written Artifacts</td>
<td>6</td>
</tr>
<tr>
<td>3.4.3 Collaborative Knowledge Management</td>
<td>7</td>
</tr>
<tr>
<td>3.5 Collective Code Ownership/Continuous Integration</td>
<td>7</td>
</tr>
<tr>
<td>4. BENEFITS</td>
<td>8</td>
</tr>
<tr>
<td>4.1 General Benefits</td>
<td>8</td>
</tr>
<tr>
<td>4.2 Benefits to HUMS</td>
<td>10</td>
</tr>
<tr>
<td>4.2.1 Examples of the Potential Benefits of Automated Testing</td>
<td>10</td>
</tr>
<tr>
<td>4.2.2 Examples of Proven Benefits of Automated Testing</td>
<td>12</td>
</tr>
<tr>
<td>5. PLANNING ISSUES</td>
<td>13</td>
</tr>
<tr>
<td>5.1 Tool Qualification</td>
<td>14</td>
</tr>
<tr>
<td>5.1.1 Relevant DO-178B Guidelines</td>
<td>14</td>
</tr>
<tr>
<td>5.1.2 Tool Classification</td>
<td>14</td>
</tr>
<tr>
<td>5.1.3 Qualification Data</td>
<td>14</td>
</tr>
</tbody>
</table>
9. RESEARCH RESULTS AND CONCLUSIONS

9.1 Conclusions
9.2 General Results
9.3 Status of Technology Readiness Levels

9.3.1 Automated Testing of the HUMS Ground Station
9.3.2 Automated Testing of the HUMS Airborne System
9.3.3 Automated Testing of the HUMS Configurations

10. REFERENCES

APPENDICES

A—Automated Testing of Health and Usage Monitoring System Configurations
B—Examples of Process Changes
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>18</td>
</tr>
<tr>
<td>2</td>
<td>18</td>
</tr>
<tr>
<td>3</td>
<td>23</td>
</tr>
<tr>
<td>4</td>
<td>23</td>
</tr>
<tr>
<td>5</td>
<td>34</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Qualification Data Required for a Software Verification Tool</td>
<td>15</td>
</tr>
<tr>
<td>2</td>
<td>Causes of Broken Builds</td>
<td>25</td>
</tr>
<tr>
<td>3</td>
<td>Technology With Cost and Benefit of Retrofitting</td>
<td>34</td>
</tr>
</tbody>
</table>
## LIST OF ACRONYMS

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AW139</td>
<td>AgustaWestland 139 Helicopter</td>
</tr>
<tr>
<td>BA609</td>
<td>Bell/Agusta Aerospace 609 Tiltrotor Aircraft</td>
</tr>
<tr>
<td>CI</td>
<td>Condition indicator</td>
</tr>
<tr>
<td>CMMI</td>
<td>Capability Maturity Model Integration</td>
</tr>
<tr>
<td>COTS</td>
<td>Commercial off-the-shelf</td>
</tr>
<tr>
<td>DBMS</td>
<td>Database management system</td>
</tr>
<tr>
<td>DCM</td>
<td>Developmental Configuration Management</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical user interface</td>
</tr>
<tr>
<td>HGS</td>
<td>HUMS ground station</td>
</tr>
<tr>
<td>HUMS</td>
<td>Health and Usage Monitoring System</td>
</tr>
<tr>
<td>IM</td>
<td>Instant Messenger</td>
</tr>
<tr>
<td>MVP</td>
<td>Model View Presenter</td>
</tr>
<tr>
<td>OS</td>
<td>Operating system</td>
</tr>
<tr>
<td>PC</td>
<td>Personal computer</td>
</tr>
<tr>
<td>PSAC</td>
<td>Plan for Software Aspects of Certification</td>
</tr>
<tr>
<td>RTB</td>
<td>Rotor Track and Balance</td>
</tr>
<tr>
<td>SIL</td>
<td>Safety Integrity Level</td>
</tr>
<tr>
<td>SUM</td>
<td>Structural Usage Monitoring</td>
</tr>
<tr>
<td>TAS</td>
<td>Tool Accomplishment Summary</td>
</tr>
<tr>
<td>TDD</td>
<td>Test-Driven Development</td>
</tr>
<tr>
<td>TRL</td>
<td>Technology Readiness Level</td>
</tr>
</tbody>
</table>
EXECUTIVE SUMMARY

The primary objectives of this research were to develop, validate, and demonstrate automated testing and system functionality partition methods for Health and Usage Monitoring Systems (HUMS) airborne- and ground-based stations. Particularly, this effort shall support Advisory Circular 29-2C, Section MG-15.

This report relates those technologies to HUMS in two ways: (1) by describing the benefits of automated testing, and how they apply to HUMS and (2) by recommending processes, environments, and function partitioning related to automated testing. Both ways parts are backed by GE Aviation’s own experiences in applying automated testing to HUMS on the Bell/Agusta Aerospace 609 Tiltrotor Aircraft HUMS program and by a demonstration project that went through the tool qualification of an automated testing framework.

This report summarizes current tools and the latest techniques and methods available to assist the automated building and testing of HUMS software. The results of an industry survey included an evaluation of how the software industry typically employs these tools and their approaches to integrating the tools into a usable environment. The report summarizes the current technology level of the research and makes recommendations for additional research.
1. INTRODUCTION.

When applied to Health and Usage Monitoring System (HUMS) software, automated test and build environments have the potential to significantly reduce development costs by improving quality and adding automation to portions of the software certification/recertification process.

This report presents detailed research and should serve as a complete foundation for a development team to introduce automated test and build to their Health and Usage Monitoring Systems (HUMS) software.

Sections 2-4 introduce automated build and test techniques and the benefits to HUMS.

Sections 5-8 describe details of the technology as well as guidelines for incorporating the technology planning through full implementation and the related processes, while satisfying DO-178B [1] guidelines at every stage.

Section 9 describes the research conclusions, current level of the research, and future research recommendations.

2. TRENDS IN SOFTWARE DEVELOPMENT.

Software development has existed as a discipline for only a matter of decades. Early in its history, the nature of software development was quickly recognized. Software development is far less constrained than other technical disciplines. Compound this reality with changing development requirements to yield a recipe for bug-ridden software and cost overruns.

In an attempt to manage the inherent risks of software, formal development processes emerged. Change was generally seen as an enemy to be combated with specification and significant upfront design. The classical and prototypical “Waterfall” process emerged to introduce structured design to manage the risks of software. The basic Waterfall process progresses through requirements gathering, design, implementation, testing, release, and maintenance in distinct, sequential phases. Accomplish each step to a sufficient degree and success would seem assured. However, even the author who named this approach, Winston Royce, recognized the risks of the Waterfall approach. Royce stated the following on page two of Managing the Development of Large Software Systems: “I believe in this concept, but the implementation described above is risky and invites failure.” Though largely unrecognized, Royce went on to advocate within his paper a means for iterating and allowing for feedback between the stages of software development. The Waterfall method as commonly understood is, in fact, the very thing Royce was attempting to avert.

The past decade of software development has seen a general and growing recognition that change cannot be designed out of the software development process. Instead, it must be embraced. Requirements will inevitably change over the course of a project due to unforeseen business, technical, or user circumstances. In addition, technical insight gained during the course of a project will directly affect the implementation of the system under development. Ignoring these changes or attempting to design change out of the system invites brittleness in the form of systems that are difficult to maintain and dissatisfaction in the features eventually delivered.
More agile methodologies are taking root in industry to align the development process with the change inherent to it [2].

Best practices (and the supporting tools) discussed in this report include short iterations, test-driven development, customer tests, documentation techniques, collective code ownership, and continuous integration.

3. INDUSTRY TECHNIQUES AND BEST PRACTICES AND THEIR APPLICATION WITHIN DO-178B.

Dealing with change effectively (i.e., in an agile manner) manages the financial risk of traditional software and produces a quality product. In contrast, preservation of human life in aerospace software is not a simple extension of dealing with change or producing economically sustainable quality. The integrity and security necessary in life-critical systems is orthogonal to the standard financial and quality concerns of the broader software industry. However, the agile methods taking hold in the broader industry have yielded success and are generally accepted as worthy of merging with the needs of high-integrity software systems such as those guided by DO-178B [3-8].

Agile techniques may seem, on the surface, to be risky and unstructured. This is not the case. Even highly structured processes and certifications such as Capability Maturity Model Integration (CMMI) (popular in aerospace settings) have found compatibility with agile methods [9 and 10].

The best practices and tools have been used successfully by the authors of this report through direct experience over several years. In this context, success has been measured anecdotally by a lack of bug reports or downtime in sizable software systems put into production use. For instance, a software system put in use worldwide in multiple automobile production plants required nine months and three developers to implement. To date, it has not generated a bug report. Other organizations with whom the authors have worked have had similar experiences. Burke E. Porter Machinery of Grand Rapids, Michigan, has had success with two projects using the described tools and processes. DaimlerChrysler’s Core Tools and Processes Group has used the tools and techniques discussed within this report to weather severe cutbacks and the recession of the late 1990s and early 2000s.

The following best practices are tied directly or indirectly to automated build and automated test techniques. Combined with supporting tools and adapted to the life-critical needs of aerospace software, these practices have the potential to deliver great value within aerospace applications. The following discussion does not concentrate on any particular level of certification (A-E) within DO-178B. General recommendations are made with the understanding that real-world process experimentation and certification evaluations of the tools recommended must take place.

3.1 ITERATIONS.

An iteration is a compressed cycle of software development; design, implementation, testing, and delivery are all completed within a single iteration for a subset of system requirements. All that is necessary to accomplish that limited set of requirements is performed in a single iteration. The resulting progress is delivered to the customer or a customer proxy for exercise and feedback. In
this way, large projects are divided into manageable chunks. Measurable progress metrics (also known as velocity) can be gathered and monitored. Requirement changes and system knowledge gained during development can be incorporated into the project. Automated test and automated build techniques are intimately tied to the capability of delivering working software within the time confines of a single iteration. Modern software processes concentrate on short iterations of development. These iterations can last from 1 to 4 weeks (current trends favor 1 to 2 week iterations).

Within DO-178B, short iterations have much promise. In fact, DO-178B is clearly compatible with iterative methods. The Spiral Model of software development is used to meet DO-178B objectives [11]. In essence, the Spiral model breaks down large projects into multiple, smaller projects and milestones generally on the order of 6 months. Taken to its extreme, short iterations are the maturation of this approach.

The stringent certification needs of aerospace software potentially require certain modifications to the short iterations used elsewhere in the software industry. For instance, longer iterations of 3 or 4 weeks may be necessary to make room for documentation and certification artifacts. The involvement of a certification assessor and/or safety engineer in each iteration may also be necessary as part of creating the certification assurance case [12 and 8]. By embracing requirements and architectural changes within iterations and including the activities of assessors and safety experts, final certification assessments could be pushed well to the end of the project. The involvement of certification assessors and the incremental inclusion of requirements changes in this iterative fashion would likely limit the overall cost of certification assessment and prevent the cost of incorporating requirement changes after certification.

3.2 TEST-DRIVEN DEVELOPMENT.

3.2.1 Overview.

In traditional software development, processes, requirements, and features are specified upfront followed by design, implementation, and then testing in distinct phases. There is a growing recognition within industry that design, implementation, and testing are best accomplished in an iterative and concurrent manner. In fact, the industry is moving toward a practice known as test-driven development (TDD), where the testability of source code drives the low-level design of the software [13].

Requirements broadly specify the functional goals that software is to accomplish. Some measure of architectural work will lay a foundation for the implementation of the source code. In TDD however, low-level decisions on how particular source-level functions and methods are written are dictated by the demands of testability. At the lowest level, testability refers to unit tests assembled into a test framework and run under automation.

TDD is counterintuitive. It prescribes that test code be programmed before the functional code those tests exercise is implemented. Practicing TDD means designing software such that it can be tested at any time under automation. Designing for testability in TDD is a higher calling than designing “good” code, because testable code is good code.
Traditional testing strategies rarely impact the design of production code, are onerous for developers and testers, and often leave testing to the end of a project when budget and time constraints threaten thoroughness. TDD systematically inverts these patterns.

Practicing TDD at the unit test level follows these basic steps:

1. From the system requirements, identify a single piece of system functionality (a single function or method) necessary to implement a feature or part of a feature.
2. Program a unit test to verify that functionality; add the test to the automated test suite.
3. Stub out the production code under test (to allow the test code to compile).
4. Compile; run the test and see it fails (due to the nonexistence of production source code).
5. Flesh out the production code.
6. Compile; run the test.
7. Refactor the production code.
8. Repeat steps 6 and 7 until the test passes and the production code is cleanly implemented.
9. Repeat steps 1-8 until all features are implemented.

The test-first approach of TDD encourages testing to pervade all aspects of a project. TDD demands that design decisions pave the way for testability. This extends from low-level unit testing on through integration, system, and acceptance testing. Each type of testing views the system from a different perspective, builds upon the success of the previous level of testing, and increases the overall confidence in the system. Test code and production code grow in parallel yielding fully tested code that implements the goals of the software requirements.

TDD is compatible with DO-178B. Low-level requirements can be directly related and traced to individual and groups of automated unit tests. Successfully passing unit tests directly demonstrate that low-level requirements have been met. High-level requirements can be directly related and traced to automated acceptance tests. System tests (i.e., executable test programming) written in response to feature requirements and certification requirements become automated acceptance tests that demonstrate the certification requirements have been met.

The needs of high-level DO-178B certification can be met by requiring the certification assessments to be expressed in terms of specific, defined tests. In such a format, automated system tests can verify that these requirements have been met. For example, an automated test could assert that the output of a coverage analysis tool indicates full coverage of the code. As such, these system tests become certification acceptance tests that can be run at any time.

The automated unit and system tests described in the preceding paragraphs constitute dynamic analysis (i.e., running the production code and testing it with running test code). For higher levels of DO-178B verifiability, static analysis can also be included in the automated testing.
Static analysis tools enforce coding standards and inspect source code for known problems including such things as uninitialized variables, buffer overruns, etc. A static analysis tool can easily be added to an automated build system that executes both dynamic and static analysis tests before releasing compiled artifacts.

Dynamic and static analysis test frameworks (available both commercially and via open source) can be qualified. Running these tools against source code with traceability between individual tests and requirements should satisfy many of the DO-178B objectives [5].

3.2.2 Graphical User Interface Testing.

Graphical user interface (GUI) testing presents special challenges. A GUI design is subject to frequent changes as an application is developed and subjected to customer approval. Further, GUI code is tightly linked to application functionality. Effectively introducing input and inspecting output through on-screen graphical widgets is difficult to do thoroughly without automation, yet automation is difficult to accomplish with on-screen graphical widgets.

The Model View Presenter (MVP) approach is a design pattern [14]. The MVP pattern (and particularly a method of use known as Presenter First) can be successfully employed to accomplish GUI testing [15 and 16]. MVP decouples the GUI widgets (View) from control logic (Presenter) and system state (Model). Using this structure, the presentation logic and system state can be unit tested apart from on-screen widgets. The View wraps widget functionality in the thinnest wrapper possible. Widgets can be assumed to function free of the need of direct testing, or more rigorous tools and techniques can test the widgets directly apart from underlying application logic.

MVP segregation allows the majority of the GUI to be unit tested alongside the application code in an automated test suite. Use of this pattern does not eliminate on-screen testing. However, it does lessen the extent of that testing requiring only verification of the underlying linkages. Where automated GUI-testing tools are insufficient for testing GUI logic, these same tools can accomplish much, if not all, of the basic linkage verification left by the MVP pattern. Traditional human-driven, exploratory testing remains beneficial and can proceed quite quickly given that most of the bugs in the GUI code will have been eliminated by unit testing.

3.2.3 Embedded Software and Hardware Testing.

Similar to MVP testing, embedded software can be implemented with the Model Conductor Hardware design pattern. This pattern segregates hardware, system state, and logic in a testable manner that allows for automated unit tests of hardware, system state, and logic [17].

3.3 CUSTOMER TESTS.

The practice of customer tests places the onus on the customer to specify, in terms of concrete tests, the set of conditions a piece of software must meet. Automated system tests are programmed in accordance with these requirements. As such, these system tests become acceptance tests and define the contract between the deliverable requirements and the production software. A system created with the practice of TDD will have been incrementally architected in such a way that automated tests of this nature will be possible to create and execute.
In terms of DO-178B, this type of concrete acceptance test can codify the stringent safety, integrity, and certification requirements of the software under development. If a safety engineer and/or certification assessor has participated in the previous iterations and all system requirements have been implemented as automated acceptance tests, certification of the software system approaches the formality of pushing a button to trigger the automated build system [3 and 8].

3.4 DOCUMENTATION.

3.4.1 Face-to-Face Communication.

Modern software development practices favor face-to-face communication over all other forms. Face-to-face communication significantly limits misinterpretation common in written communication and capitalizes on the positive effects of bringing humans within contact while working. Ignorance of system aspects under development is limited by gathering those knowledgeable of the system and thus, increases project velocity. Ultimately, face-to-face communication reduces the need for written communication and documentation and narrows the scope of written documentation to only that which is absolutely required.

3.4.2 Design, Assessments, and Written Artifacts.

While a software development effort is underway, the system is in a state of flux and will surely be perturbed by changing requirements or knowledge gained during development. Thus, reality tends to quickly diverge from system documentation created at project onset. As such, it is best practice to write as little documentation as possible upfront (within the constraints of the customer’s documentation requirements) while allowing tools to generate as much system documentation as possible. Upon the system’s completion, existing documents and any not yet written can be made to reflect the final system accurately. In this way, system documents are much more likely to reflect reality than reams of documents generated before the system has been completed.

Practicing TDD affects documentation in two significant ways. First, TDD allows testing to drive the low-level design of software. A great amount of design effort up front is likely to be spent in vain as the true design will evolve over time. Attempting to capture this up front design in detailed documentation will eventually yield obsolete documents. Rather, allowing the design to evolve and then capturing reality in documents of limited scope towards the end of the project is more efficient. Second, TDD prescribes test suites. A suite of up-to-date unit tests is, in effect, executable documentation that captures the lowest levels of a system’s design. Because this vital information is captured in this way, the need to represent it in detailed documents is further reduced. If low-level system documents are required, automated tools can generate this documentation. Automated documentation tools are able to represent relationships among functions, methods, classes, and modules and gather programmer comments into consistently formatted, push button generated documents.

Ultimately, the documentation requirements of the customer and auditors dictate what is written and delivered. A desire to write more than what is necessitated by these parties will often lead to irrelevant documentation and hours needlessly spent. Using test suites and automated tools to
generate system documents complemented by high-level architectural overviews provides the most efficient means towards usable documentation.

Best practices advocate limiting documentation as much as is practical. In the case of DO-178B, documentation is critical to the certification process. Two approaches can be used to handle the documentation requirements of aerospace software. First, as much of the documentation generation as possible should be automated; this document generation can be included as part of the automated build system. Additionally, it is quite conceivable that an automated tool could be created to meet the needs of requirements traceability by linking unit tests to individual project requirements. Second, for the documentation that simply cannot be automatically generated but must be written, an agile approach exists for generating these documents. As previously advocated, short iterations allow software teams to manage changing requirements and changing system knowledge. This, of course, stresses traditional, documentation-heavy certification processes. Instead of the normal linear approach to writing with a single primary author, the RaPiD7 method advocates gathering stakeholders together and generating all documentation concurrently over the course of several focused workshops [3 and 18]. Final editing is left to a single person. With this approach, certification documentation can be generated and kept current in step with short iteration cycles.

3.4.3 Collaborative Knowledge Management.

Much of the knowledge of a system under development should be collected and shared among the development team. The type of knowledge discussed here includes, for example, step-by-step instructions on configuring support tools or information on how to set up particular hardware components within the system. This knowledge is not typically well suited to be placed in source code comments or system documents. Checking some form of developer documents into and out of a version control repository is generally too cumbersome and too difficult to search or navigate to be useful.

If developer knowledge is not kept in an easily accessed and centrally located place, it is of little use to the team. Further, if the means of adding to and updating a growing knowledge base are not simple and quick, important knowledge will not be collected. Information critical to testing and build setup must be captured and shared. Knowledge management tools make this an efficient process. Knowledge management tools reduce the need for team members to relearn what was once known or what has been discovered by another team member.

3.5 COLLECTIVE CODE OWNERSHIP/CONTINUOUS INTEGRATION.

Source code must be community property among a software development team. Individual investment in certain parts of any system is certainly valuable. However, this must be balanced against the tendency to “own” source code and withhold it from the rest of the team. In the end, owning source code in this way is likely to introduce unsound programming practices, given a lack of review by peers. Further, source code managed by a single person usually causes significant problems when all the system’s source code is brought together in final integration. While developers may have certain expertise or have responsibility for certain system features, all team members must be able to view, refactor, and test all the source code. This prevents costly design mistakes and costly integration efforts.
Continuous integration is the practice of rebuilding and testing an application frequently. This practice ensures that flaws introduced into the application’s development are found and corrected quickly and that disparate system features are integrated into a whole as smoothly as possible. Continuous integration is most often aided by tools such as a source code repository, unit test frameworks, and automated build systems. Continuous integration allows a system to be built, tested, and packaged at moment’s notice. As such, the most recent working system is always at hand. This provides value to the customer, eases final testing, and increases confidence in the system. Continuous integration and automated testing complements collective code ownership by quickly exposing integration problems.

Within the context of DO-178B, an automated build tool enables the practice of continuous integration. Referring to all previous practices already discussed in this report, it is the automated build tool that runs all unit and static analysis tests, performs automated acceptance testing, generates automated documentation, and builds the release executables.

4. BENEFITS.

4.1 GENERAL BENEFITS.

There are many benefits to establishing an automated build and test environment, some of which are well publicized among continuous integration advocates. Other benefits have been discovered through the experiences of GE Aviation Systems LLC, doing business as GE Aviation, and may be particular to HUMS. The benefits that GE Aviation currently cites are listed below.

1. Ensures system robustness through the ability to run the test suite repeatedly (regression testing) throughout development.

   Traditionally, an application may only need to pass one test on one piece of hardware prior to release to the customer. Only later, after repeated use in different environments, do problems emerge related to the robustness of the software. An automated test will typically test throughout development on a number of different machines.

2. Enhances the design for maintainability and extensibility.

   Testing is an important part of any maintenance of software, and automating testing makes maintenance much simpler. Often the person maintaining the software is different from the original author. Any changes made can be quickly verified giving confidence that the changes are good.

   The automated tests will clearly demonstrate the software interface to the software maintainer. Unit tests, in particular, serve as examples of correct usage and show how the original author intended a class to be used. Thus, they serve as documentation that is always current and accurate, which contrasts written documentation that may contain errors and inconsistencies. This helps with both maintaining and extending the software. TDD will usually lead to simpler designs that can be easily grown one step at a time [19].
3. The latest software application build is consistently available to support system-level testing and customer discussions.

The latest version of a software project that is in development will be, at best, unstable and at worst, unable to build. That usually results in an older, tested version of the project being used to support system-level testing. With an automated build and test, one is able to showcase, system test, or field new functions as soon as they are implemented, even if development is still going on in other areas.

4. Frequent software releases to the customer are achievable.

Without an automated build and test, each additional software release will add time and cost for building, integrating, and testing. With automated testing, there is very little cost to each additional release, allowing frequent releases.

5. Provides an accurate method for tracking software development progress.

When measuring software development, progress credit is claimed before integration and testing. Further development will often break existing functionality causing that measured progress to be artificial. With automated testing in place, credit can only be claimed when a function is tested, and that function will be continuously tested as each new function is added.

6. Increases confidence in the quality of the integrated application.

Trade-offs need to be made in an attempt to avoid the cost of manual testing. For example, a test fails on 1 procedure out of the 100 tested. The code is reworked, but the cost of manually testing all the functions appears disproportionate to the benefit, so instead, an analysis is performed that assumes that the code change will only affect two functions, and only they need to be retested. Unfortunately, these assumptions are not always correct, and unexpected effects often occur from software changes.

Even if the problem fix was successful, if a simple error is made during the building of the software (for example a file not copied to the correct installation directory) it will not function correctly. An automated build removes the risk of human failures and ensures consistent rebuilding of software. This is especially important for larger project where a software build includes many steps.

7. Provides confidence and easy introduction of new technologies.

There is a lot of risk and resistance to the introduction of new technologies. A minor change to a low-level function could result in the need to retest large portions of the system. With automated testing in place that is no longer an issue; even minor improvements in technology (e.g., an operating system (OS) service pack) can be confidently introduced.

8. Provides the ability to keep working up to the release date.
With a traditional life cycle, a period of time is usually reserved prior to the release date for a cycle of integration, testing, and rework. With automated build and test, it is always possible to release a tested application at any time, with whatever functionality is in it at the time. The related benefit is that if things go badly relative to the plan, you have the option of sacrificing functionality to make a release date.

9. Requires less manual testing effort.

This is the most obvious benefit of automated testing.

10. Finds problems as early as possible in the development cycle.

Because a manual test takes time and effort to execute, it is typical to only test software immediately prior to release. The problem with this is twofold. First, the timing is bad, and the release may need to be postponed at short notice. Second, it may not be clear how or when this problem was introduced. With automated testing, any failure will result directly from the last iteration of code changes, making debugging the problem a much simpler task.

4.2 BENEFITS TO HUMS.

Each benefit above translates to new benefits that are particular to HUMS. (General Benefit numbers from 0 to 10 are shown in parenthesis).

1. Time and cost to initial certification could be reduced. (1, 3, 4, 5, 6, 8, 9, and 10)

2. In general, problems with fielded systems would more likely be addressed. (1, 2, and 9)

3. In general, problems with fielded systems would be addressed earlier. (4)

4. In particular, problems related to performance (e.g., speed, reliability) with fielded systems would more likely be addressed. (7)

5. Increasing the scope of HUMS by interfacing to other aircraft systems would become an easier task. (7)

6. Could allow HUMS to move to a higher degree of certification, and increase its effectiveness. (1, 6, and 9)

Each benefit is covered in part by the examples from the following section.

4.2.1 Examples of the Potential Benefits of Automated Testing.

4.2.1.1 General Benefit for an Evolving HUMS.

Over the last 15 years, evidence has clearly shown that the most effective systems are ones that have modified the HUMS on a regular basis as a result of field experience. These changes can fall into the categories of diagnostic changes in the light of fault case experience or rig test work, man-machine interface changes to better present the information to the end user, and data
collection/regime/signal processing changes that modify the way the data is collected on the vehicle. Good systems design and configurability can help make this process of change easier. However in practice, with the growth rate of knowledge with regard to the application of HUMS being so rapid, it is essential to have a cost-effective process for making software changes to the HUM system, particularly the ground station.

Fault cases are typically classified in two categories: Classical and Novel. Classical faults have diagnostic algorithms that were expressly designed to be detected, and whose behavior could have been predicted in advance. An example would be a gear tooth crack or bearing raceway damage.

As an example of learning from experience, GE Aviation has reviewed fault case histories from over 2 million HUMS flight hours and has been able to determine that at least 50% of all faults detected, or detectable by HUMS, have fallen into a Novel category.

Novel faults are detected in a way that is hard to predict through theory. An example would be detecting airframe cracking from a change in gearbox vibration signatures. The importance of this almost equal split between Novel and Classical is that its highly possible that some Novel faults may be missed on the first occurrence; however, if there is a clear signature/pattern in the data, then there is no excuse for missing the fault twice. Typically, this will require changes to the system software and hence a need to be able to make software changes in a timely and cost effective manner.

Automated testing is vital in making these small and incremental changes cost effective. Section 7.3 discusses how partitioning can be used in these areas of high change to further reduce certification costs.

4.2.1.2 General Benefit for a Higher Software Certification Level.

With a desire to move toward certified credits with HUMS, there has been a move toward a higher software certification level. Typically, a move from DO-178A level D to level C or B.

Systems that have already been developed to this higher level have resulted in higher software development and modification costs. It is interesting to note that, for the GE Aviation’s systems fielded to these two different levels, there has been less modification and update of the higher certified systems, primarily due to the higher cost of change. Ironically, this has resulted in the lower certification systems with the higher rate of change being more effective systems on the aircraft.

The move to higher certification levels is going to be necessary to exploit the full benefits that HUMS can offer, but one key challenge will be to adopt the higher certification levels without compromising the ability to change and evolve the systems in a cost effective manner.

The use of automated testing, with its proven reduction in the costs of software retest, is likely to be a key enabler to adopt higher system certification levels.
4.2.1.3 United Kingdom Ministry of Defence Merlin HUMS.

The United Kingdom Ministry of Defence has two HUMS in service, which provide an interesting contrast in terms of validation approaches. The report “Contrasting Approaches to HUMS Validation—A Military Perspective” [20] shows how the validation approaches ended up affecting both the initial functionality and the maintainability of the system.

The Merlin HUMS was certified to a high level (Safety Integration Level (SIL) S4) and was tightly integrated into the critical aircraft systems. By contrast, the GenHUMS, initially introduced on the Chinook, was certified to a lower level (SIL S2).

As an example of the functionality differences, the GenHUMS for Chinook had up to 50 condition indicators (CI), whereas the Merlin HUMS was limited to just 4. This was due to the anticipation of the extensive qualification requirements for each CI algorithm’s software.

After the Merlin HUMS was certified and in service, changes could not be incorporated without a reissue of the full aircraft software suite, so a software ‘bug’ that prevented the onboard system from writing data to the data transfer cartridge could not be corrected for over a year.

With automated testing, the negative aspects related to achieving a higher certification should be reduced by eliminating the testing costs of both the initial certification and subsequent recertifications.

4.2.1.4 AgustaWestland 139 Helicopter HUMS.

This is purely an example of where time and effort would have been saved by automated testing. The Augusta Wesland 139 Helicopter (AW139) HUMS ground station (HGS) was initially required to work on the Microsoft® Windows 2000® OS. By 2004, it became clear that it would be difficult to purchase personal computers (PC) that came with anything other than Microsoft Windows XP®. No changes were required to the software; GE Aviation had internally used the HGS on Windows XP for over a year without any problems. The documentation had to change OS from Windows 2000 to Windows XP, which was straight forward. The only significant cost of the change was manually repeating the test in the new environment.

This is an example of a common issue in the HUMS ground environment, where retesting is required when updating commercial off-the-shelf (COTS) software such as database management systems (DBMS), OS service packs, .NET framework, and similar application frameworks.

4.2.2 Examples of Proven Benefits of Automated Testing.

The Bell/Augusta Aerospace 609 Tiltrotor Aircraft (BA609) HGS has highlighted many automated testing benefits. In general, the benefits are visible on a day-to-day basis. For example, additions to functionality and problem report fixes have been implemented in less time and cost than prior to the introduction of automated testing.

The following are more examples from the BA609 HUMS where the benefit is greater than just a time and cost savings.
4.2.2.1 BA609 HUMS DBMS.

Recently on the BA609 HUMS, GE Aviation was able to change DBMS from Borland’s® InterBase® to Foundation Inc.’s Firebird®. Although InterBase was sufficient for the job, the installation and licensing of it was extremely awkward, and thus unpopular with current customers. However, neither issue was great enough to justify the change in previous iterations. But after the BA609 HUMS had automated testing in place, it was possible to implement and verify the change easily.

Not only was the change easier to verify with automated testing, but it was also easier to implement the change because problems from introducing the new DBMS are noticed immediately in failed tests. Without automated testing, the implementation would only uncover a fraction of the problems immediately. Some may only be spotted at the next complete system test. At this stage, it may be difficult to associate the problems with one particular change. At worst, the problems are unsolvable, meaning that the change and everything built on that change would need to be rolled back. With automated testing, any unsolvable problem would be spotted immediately, and the change would be removed before anything was built upon it.

4.2.2.2 BA609 HUMS Download Speed.

Poor performance is often a problem that is first experienced by the customer. However with automated testing, any performance problems first become an issue for the software development team. Since the automated test suite is run each time a developer checks in code, it is important for the development team that this test runs quickly. Well-designed tests are crucial; however, if the underlying HUMS functionality is slow, the test cannot be fast.

Often, performance problems are fixed by optimizing foundational code that is executed repeatedly. The issue with changing this sort of code is that it often impacts more than just the area that requires optimizing. Normally, this would require a lot of testing effort and hence be avoided. Automated tests providing good coverage of the changed code can detect introduced errors and thereby reduce the aversion to making important performance-improving changes.

When automated testing was introduced to the BA609 HUMS, GE Aviation was able to make simple (but high-impact) changes to the foundational code, and the automated tests alerted to any unwanted side effects that were introduced. As a result, the overall download time was improved threefold for a large dataset. Previous attempts on the BA609 HUMS to improve download speed had been limited because of the risk of introducing these side effects.

5. PLANNING ISSUES.

Allowing automated testing techniques to be used as part of an aircraft certification where they replace human processes requires rigorous processes similar to those used when writing and verifying traditional manual tests. The need to consider these processes will require additional planning work.
5.1 TOOL QUALIFICATION.

5.1.1 Relevant DO-178B Guidelines.

The requirement for any tools used during a software development is normally revealed during the planning stages. For a DO-178B project, this will mean that it is documented in the Plan for Software Aspects of Certification (PSAC) [1].

DO-178B states: “The Plan for Software Aspects of Certification (PSAC) should include a listing of all software tools and justification for why each tool does or does not require qualification.”

5.1.2 Tool Classification.

DO-178B describes verification tools as “Tools that cannot introduce errors, but may fail to detect them.” Clearly, automated testing tools are relied upon to detect errors and potentially could fail to detect them. DO-178B describes development tools as “Tools whose output is part of airborne software and thus can introduce errors.”

Automated test frameworks do not and cannot affect the contents of built software items, because they are used after the application is built and therefore are not classified as software development tools. It is theoretically possible that a test framework could be used as a software development tool on project, which is why this analysis needs to take place on each project that uses a tool. The successful qualification of a tool for one project does not imply that it is qualified for all projects. However, the qualification data produced for one project can be used for other projects as long as the tool is being used in the same way.

Another option is to use automated testing tools only during development and to perform the formal verification by other means. Therefore, this does not eliminate, reduce, or automate any DO-178B processes and hence requires no tool qualification. However, since this report demonstrates that qualifying automated testing tools is relatively simple on all but the smallest projects, when using automated testing, it is most cost-effective to use this for formal verification also.

In conclusion, for most projects, automated testing tools should be classed as a software verification tools and not development tools.

5.1.3 Qualification Data.

Table 1 shows the data required for tool qualification of a software verification tool [2].
Table 1. Qualification Data Required for a Software Verification Tool

<table>
<thead>
<tr>
<th>Data</th>
<th>Available/Submit</th>
<th>DO-178B Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSAC</td>
<td>Submit</td>
<td>12.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12.2.3.a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12.2.4</td>
</tr>
<tr>
<td>Tool operational requirements</td>
<td>Available</td>
<td>12.2.3.a(1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12.2.3.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12.2.4</td>
</tr>
<tr>
<td>Software accomplishment summary</td>
<td>Submit</td>
<td>12.2.4</td>
</tr>
<tr>
<td>Tool verification results</td>
<td>Available</td>
<td>12.2.3.c</td>
</tr>
</tbody>
</table>

Note that there are no differences in the tasks required based on the criticality level of the software that is being verified by the tool. However, it would be expected that more attention would be given to the qualification data for higher levels of software criticality.

5.1.3.1 The PSAC.

The PSAC is where the automated testing tools will first be acknowledged as being used for software verification and will identify the qualification data that is to be produced.

5.1.3.2 Tool Qualification Plan.

GE Aviation used their document titled “Tool Qualification Plan” as a holder for the tool operational requirements and the test procedures used for verification.

5.1.3.2.1 Tool Operational Requirements.

The operational requirements for the tool specify the functionality of the tool that is being used, and the environment in which it is used.

For vendor software, as with most automated testing tools, the requirements usually have to be reverse engineered from the documented functionality.

It is normal to capture requirements for the full functionality of the tool, even if beyond those actually needed. One may restrict the operational requirements to only cover a subset of the tool functionality that is being used. However, if this is done, then the use of the tool needs to be restricted within the project to make sure that undocumented functions are not used.

By qualifying a tool completely, it is far more likely that the tool can be reused on other projects with the same qualification data.

5.1.3.2.2 Test Procedures.

DO-178B states: “The qualification criteria for software verification tools should be achieved by demonstration that the tool complies with its Tool Operational Requirements under normal operational conditions.”
So for an automated testing tool, the test procedures need only test the tool under normal operating conditions on the target development environment. Typically, for the most basic automated testing tool this will mean:

- Passes, failures, and errors (when the testing abnormally exits) are reported correctly.
- Tests that assertions pass and fail correctly given the appropriate arguments; e.g., tests that assertEquals(1,2) causes a failure and assertEquals(1,1) causes a pass.

New test procedures can be created to test the operational requirements under normal operational conditions. However, most automated testing tools come with a test suite that fully tests all functionality. These vendor-supplied tests usually go far beyond the testing required and perform white box testing of the internal functionality. When using vendor-supplied tests, the tests should be carefully verified to ensure that all requirements are fully tested. Since these white box tests have knowledge of the software internals, this verification work can be significant and may be costlier than writing one’s own tests.

5.1.3.3 Test Results.

Automated tests may be an unconventional practice in some organizations, and it is wise to discuss the best way to execute and document the tests with the software quality assurance personnel.

Since the tool needs to be tested in its operational environment, tests may need to be repeated on different platforms, or when a new OS is used. Therefore, it is good practice to ensure that the test procedures are developed to be independent of OS.

5.1.3.4 Tool Accomplishment Summary.

The Tool Accomplishment Summary (TAS) will identify a version of the tool and the documentation set that goes along with it. This may include identification of the test results for each tested platform. It will include any known problems and any restrictions on usage. The TAS should be developed separately from the main software accomplishment summary, which will reference the TAS.

5.2 STAFF PLANNING.

The learning curve for developers is greater than that for the typical software engineering change of switching programming languages. The change here is that of process followed rather than just new syntax and updated design philosophies.

The two greatest issues for developers are learning how to write good tests and understanding the continuous integration philosophy.
5.2.1 Learning to Write Good Tests.

Developers will initially find it difficult to write automated tests. The tests are often misapplied by developers as a means to execute the code to prove that it is working. The developer then has to evolve these informal and experimental tests into a well-structured test.

Training in writing good tests is useful, and due to the popularity of Agile and Microsoft Windows XP methods, there are many training materials available.

When practicing continuous integration, it is common for a developer who has introduced new functionality to have to modify an automated test produced by another developer. This can be beneficial as an additional form of review. Where possible, it is best to plan the development team so that at least one person on the team has strong experience in developing automated tests.

Harder still is the transition to TDD (see section 6.4.2) where tests are written prior to application code, a transition that yields many benefits.

5.2.2 Learning to Follow Continuous Integration.

The challenge here is to quickly learn the continuous integration build rules (see section 6.4.1). If not learned and followed, this can adversely affect the entire development team in addition to the individual. Unfortunately, these rules can only be successfully learned on the job, and should be enforced strictly to encourage new developers to adapt quickly.

Section 6.3.2.5 discusses the common causes of broken builds, which are often caused as result of developer inexperience.

6. IMPLEMENTATION.

There will be common aspects to all implementations and the processes used. Figure 1 shows the basic infrastructure and the steps taken for each build cycle.
4. Results of build are reported to developers

Figure 1. Simple Automated Build and Test System

6.1 TEST STRUCTURE.

Automated tests are part of an automated test system compiled separately from the system application software. The application software tested is the final application binaries, including any nondevelopmental items or COTS software.

The system application software must have no dependencies on the test system. For DO-178B verification, any tests must be performed on the final application binaries and on the target platform.

The automated test system stimulates functionality within the application, and then compares the actual outputs against verified expected results (see figure 2).

Figure 2. Software Structure of an Automated Test System
The diagram above shows the link between the automated tests and the application under test. For acceptance testing, which verifies that the software meets its functional requirements, the application is used through its normal interfaces (from the outside). For unit testing, which tests a single class or module in isolation, the tests run against isolated parts of the system by interfacing at the class or module layer.

Automated test frameworks are described in more detail in section 6.3.3.

6.2 HARDWARE AND NETWORK INFRASTRUCTURE.

A network of machines is required to achieve all the functional requirements of a continuous integration and automated build and test environment. This section documents the role of each of those machines.

6.2.1 Build Machine.

The Build Machine is the name commonly given to the computer that controls the execution of the automated build. This computer will have continuous integration software (see section 6.3.2) running to detect changes to the Developmental Configuration Management (DCM) (i.e., when developers update the source code).

Often, build execution will be the bottleneck of the automated build and test environment. Therefore, it is often worth investing more in the hardware capabilities of this machine. In particular, multiprocessor capabilities can give great benefits, but will require independent processes or threads in the software (similar to distributed builds) to be properly exploited.

6.2.2 Web Server.

Other than the version control tool the web server will be the main human interface to the continuous integration environment. A build status page will typically show the status of each build with controls to get more data or override the scheduling options. The web server functions require little computing power and do not require a dedicated machine per project. Typically, the infrastructure is set up so a single web server provides build results from a number of different projects. Alternatively each project may have a combined build machine and web server.

6.2.3 Developmental Configuration Management Server.

Typically, DCM will already be standardized in most software development organizations. A continuous integration environment requires good performance from the DCM, which in part is influenced by the choice of hardware.

6.2.4 Developer Machines.

When a software developer checks in changes that render the project unbuildable, or cause tests to fail, then the build has been broken (see section 6.3.2.5), and it impedes the progress of others on the team. To avoid this, each developer is expected to run the tests on his local build before checking in. The tests must pass before he is allowed to check in.
Therefore, developers will now run automated build and tests as part of their everyday development. Although this is highly beneficial, developers cannot spend too much time waiting for tests to complete. Developers will run only the tests related to the software they are working on until they are ready to check-in, when they must run the full suite of tests. This can be time-consuming, so it is recommended to invest well in optimizing the test design and the hardware capabilities of developer machines, too.

6.3 TOOLS.

6.3.1 One-Step Build.

One-step build tools allow source code and tests to be compiled and run without direct user interaction. These tools often replace or supplement project files created by integrated development environments (e.g., Microsoft Visual Studio®).

For existing projects, setting up the one-step build will often be a time-consuming task. The starting point will be the build instructions. On some projects, it may not be practical for individual engineers to ever build and integrate the entire system. The build instructions are often a more of a log of the tasks carried to build the software, and any attempt to recreate the build will run into difficulties.

Using GE Aviation’s preferred one-step build tool, Apache Ant®, it is possible to call other processes directly as one would in a batch file. However, it is usually worth looking to see if Apache Ant has built-in support for what you are trying to do. For example, Apache Ant has built-in tasks for

- most of the common OS command line tools
- compilation-related tasks
- many Sun Microsystems® Java™ related tasks (Apache Ant is designed to be extendable using Java)

Moving to a one-step build instantly documents most of the build process, so builds should be consistently reproducible. Some tool setup instructions will still need capturing even with a one-step build.

6.3.1.1 Typical Tasks Within the One-Step Build.

The one-step build procedure is needed to give as much confidence as possible in the status of the modified source code. Anything that can automatically give a pass/fail indication of the modifications status should be included in this build. The following are typical tasks that are included during a one-step build.

- Clean—Prepare the build environment by deleting any artifacts from previous builds. These artifacts may include: test results, intermediate objects files, and executable files.
• Compile Application—Compile and link the application source code to produce executable binaries.

• Compile Tests—Compile and link the test source code to produce executable binaries.

• Build Configuration Data—Build the source configuration data into files that are delivered with the application.

• Build Test Data—Build the test data used in the testing of the application but not delivered with the application. This may include alternative configurations.

• Static Analysis—Use a tool to perform a static analysis on the application and test source code (see section 6.3.7).

• Execute Tests—Execute the test binaries that were produced at the compile test task.

• Code Coverage—Depending on the code coverage analysis tool used, this may involve instrumenting the original source code, recompiling, and re-executing the test binaries (see section 6.3.6).

• Build Installer—PC-based software, such as HUMS ground stations, is typically delivered as a single setup file. This task typically relies on tools such as Microvision InstallShield®, Altris® Wise Installation Studio®, Nullsoft Scriptable Install System, or Inno Setup for producing the installation package.

The tasks that are typically not included in the main one-step build are:

• Installing the software for the build environment, since this only needs to be done once.

• Other one-time setup tasks.

• Generating documentation (see section 6.3.8). This is an important task; however, it will not give an automatic pass/fail indication.

• Retrieving source from version control. This is a useful task to automate; however, it would interfere with any local modifications and is usually a task performed only on the continuous integration server.

A one-step build tool may be used to automate these tasks; however, they will not be part of the main scheduled build.

Any failure of any build step must be reported unambiguously. Avoid or minimize nonfailure messages so the pass/failure notification will not be obscured.

6.3.1.2 Multithreaded Build.

Multithreading allows the build to complete in a shorter time span by running some tasks in parallel. To achieve this benefit, a build machine with multiple processors (or processor cores) is
required to allow tasks to run in parallel. The ability to design the build by selecting independent
tasks that could be run in parallel is also required.

To implement this, Apache Ant provides a `<parallel>` tag, which will execute the tasks inside it
in parallel. There is also a `<sequential>` tag, which will execute the items in it in sequence
(useful when placed in a parallel tag). For instance, one thread could be building the source code
while the other is populating a database with test data.

6.3.2 Continuous Integration.

Continuous integration systems work in conjunction with version control and one-step tools to
monitor changes in a source code repository and automatically build the software under
development. Choice of a continuous integration system may depend on the choice of other
tools being used and the software development environment.

6.3.2.1 Status Information.

Build rules (see section 6.4.1) determine when developers are permitted to check-in and check-
out source code. Adhering to these rules requires that developers have access to the current build
status at any time.

Most continuous integration tools provide several mechanisms for doing this, with personal
preference usually dictating which method the developer uses. Typically, continuous integration
tools use build logs to store the build results, which can then be presented in a number of
different ways.

6.3.2.1.1 Web Page.

The most widespread way to present the status is via a web page. This way it is easily accessible
by all developers and can be linked to from other web-based resources.

The CruiseControl™ web page, as shown in figure 3, displays each project (build) along with the
result of the last build. Since CruiseControl is open source software, GE Aviation has
customized it to expose a button on the status page, which allows one to pause the build (to
overcome a common problem with our version control system) in addition to the standard forced
build button.

This page will be accessed frequently by developers and, as shown in figure 3, can be used as a
portal to add links to other project content, in this case the Sun Javadoc™ documentation and the
code coverage analysis.

The status page will usually contain links to historical build information, allowing the
development team to track important metrics such as build time and percentage of failed builds.
6.3.2.1.2 Other.

Among the other popular methods of displaying status is the use of the Microsoft Windows taskbar notification area. The screenshot shown in figure 4 is taken from CCTray, a utility provided for CruiseControl.NET.

![Microsoft Windows Taskbar Notification](image)

Figure 4. Microsoft Windows Taskbar Notification

The advantage of the system tray icon is that it is always running and active, and it provides an instant reporting mechanism if there are any changes to the system.

Really Simple Syndication Feeds are available from CruiseControl and other continuous integration tools, which can be displayed in a web browser or directly on the desktop.

6.3.2.2 Instant Reporting.

For developers who have just changed the build, seeking out status information is not suitable. If a developer has broken the build, it is important that they not ignore it.

This is particularly important for developers who are new to continuous integration and might otherwise ignore the status of the build machine.

Experience has showed that at the start of projects, or when new developers are introduced to continuous integration, strong encouragement is required to ensure that the build is not broken because of carelessness, and that if it is broken, it is fixed promptly.
6.3.2.2.1 Email.

Email is often the most efficient way of instantly and automatically communicating the build information in a way that gets noticed. Typically, software developers will have an email client continuously running, and read mail as it arrives.

6.3.2.2.2 Instant Messengers.

Instant messengers (IM) are becoming more popular in offices and, in many cases, are replacing email as the standard form of messaging. In such cases, emails may be checked infrequently and IM may be the preferred choice for communicating the build information.

Note that IM is currently not a standard part of most information technology infrastructures and, worse still, may be explicitly blocked from use like email once was.

6.3.2.3 Backup Strategy.

The reason for backing up the build machine is not because it contains anything that is not stored elsewhere (since the build machine should reflect the developer’s machines), but rather because it is an essential resource that needs restoring immediately if it ever fails. Thus, a backup strategy that backs up whole partitions rather than selections of individual files is preferred. Backing up selections of files may miss important configuration information, which then has to be laboriously recreated.

6.3.2.4 Build Rules.

Build rules are required to ensure that the use of continuous integration environment is consistent and reliable for each developer. These rules are detailed in section 6.4.1.

6.3.2.5 Broken Builds.

A broken build occurs when any task in the one-step build fails.

The broken build is a useful indication that something requires fixing; however, these errors should have been spotted by the developer running the exact same one-step build prior to checking in.

The problematic side effects of a broken build are that, while the build is broken, other developers are unable to check-in and need to wait for the changes that fix the build before rerunning their tests. This usually means additional testing runs prior to check-in, and the developer having to start on a new task while waiting to complete the previous task.

Table 2 lists some of the general causes of broken builds. The first three cases are almost always caused by human errors, which can be avoided. For these reasons, developers who are new to continuous integration or the project will break the build far more often than experienced developers.
Table 2. Causes of Broken Builds

<table>
<thead>
<tr>
<th>Error</th>
<th>Cause</th>
</tr>
</thead>
</table>
| Developer version control error            | The developer forgets to check-in all changes. In particular, when a developer checks in all the files that were checked-out, and forgets to add new files.  
A common cause for a broken build, especially if the Version Control System is unintuitive or labor intensive. |
| Developer assumptions                     | The developer performs an impact analysis of the change made and performs regression testing rather than running a complete build. The assumptions made are not correct, and the failure is caught by the build machine.  
A likely cause of build failures when the build length is too long and developers try to short circuit the process. |
| Build machine setup                       | When the build machine setup is different from the developer’s causing an automated test to fail. Usually, this occurs when a new tool is introduced but not installed on the build machine. |
| Intermittent failures                     | An intermittent failure is one that does not occur in every build. These are difficult to detect, and a temptation is to forget these when they disappear following another build. Often, these occur because of the increased memory usage or other resource issues when running automated tests in sequence without idle time.  
Nightly builds (0) can help detect intermittent failures.  
If left unresolved, intermittent failures can significantly reduce productivity. |
| Build process differences                  | Broken builds can occur due to subtle differences in build process between the developers’ machines and the build machine. Sometimes these differences are inadvertent and once spotted can be easily resolved; however on other occasions, the differences may be by design.  
For example, during the BA609 HUMS development, the build process executed on the developer machines compiled the source code with debug information (critical to allow developer debugging), whereas the build machine ran a build without debug information. This caused a number of builds to fail, mostly where the compilation failed due to warnings that were not generated in debug mode.  
These build process differences should be avoided where possible. However on the BA609 HUMS, the additional time for developers to compile and run tests without debug information was considered prohibitive for the benefit it would give. |

6.3.2.6 Dependant Builds.

Dependant builds occur when one project has a software dependency on another project, requiring the other project to be built first. In some situations, it will not make sense to perform a build unless another build has succeeded. For example, a successful build of the HUMS Rotor Track and Balance (RTB) diagnostics may trigger separate integration tests for each HUMS. CruiseControl and other tools have features to directly support this.
6.3.2.7 Nightly Builds.

A nightly build is a build that is run at a scheduled time during the night, often regardless of whether or not changes have been detected. The benefit of a nightly build is that it does not take any valuable processing time from the build machine during working hours.

Nightly builds are usually used for tasks that do not require immediate feedback to the user or are heavily time consuming. Nightly builds can be used for the main application build and test giving greater confidence in the integrity of the build by being able to detect intermittent failures.

6.3.3 Automated Test Framework.

A test framework allows tests to be written in same language as the code being tested by providing simple functions for comparing actual results to expected results.

Unit test frameworks provide standard interfaces and automated methods for running and reporting the success of developer written unit tests [8]. The framework hosts one or more test classes in a suite comprised of individual unit test methods. These test methods are separate from the production code under test but mirror the naming structure of that production code in some way. Frameworks are tied directly to individual programming languages.

Test frameworks provide test setup and teardown functions for test classes and suites that can be overridden by the developer for any needed test initialization and cleanup. Unit test frameworks for object-oriented languages usually provide an abstract test case class that provides setup/teardown functionality, test calling, suite registration, and helper functions.

Test functions within the suite are written to perform a single action and then assert that a state was realized or an action occurred in the code under test. Unit test frameworks provide a set of assertion functions, or macros, that allow the framework to compare the expected results of a method under test with the actual results. These assertions usually exist for the all the basic types in the native language, for example, Assert.AreEqual(34.2, x, 0.0001). This NUnit (.NET) example shows how the double-precision, floating point value x matches the expected value of 34.2 within the delta of 0.0001.

A test framework also provides facilities for reporting the results of the tests. Frameworks are usually able to report the results in a way that is supported by continuous integration tools.

Since the test framework itself partly automates DO-178B verification processes, it is classified as a verification tool and thus must be qualified, see section 5.1. Also, the tests themselves must be reviewed, see section 6.4.3.

6.3.4 Mock Tools.

Mocking is used to simulate (mock) other parts of the system that are being used but not being tested and is used in conjunction with unit test frameworks.
Mocking serves to eliminate application classes from tests of other classes by replacing the application class with a mock. Consequently, the complexity of individual tests is reduced as one piece of a system can be tested in isolation from the rest of the system.

In the absence of a mock tool, the mocked version of a class must be produced through manual coding. Mock tools aid mocking by reducing the amount of code needed to produce a mock. They help the readability by allowing all the test code to be in one place with no additional classes required. Mock tools help to ensure that all mocks are produced in a consistent manner aiding readability and reducing maintenance costs.

Currently, mock tools are available for run-time-hosted languages like Sun Microsystems Java and Microsoft C#. Run-time-hosted languages provide a means for dynamically inspecting and creating objects through reflection. These features lend themselves well to the idea of mock objects. Mock tools for compiled languages may eventually become prevalent as well.

However, like a unit test framework, a mock tool would be classified as a software verification tool by DO-178B and hence, would need qualifying. This would be a relatively straightforward task (depending on the amount of functionality the mocking tool has), although it needs to be balanced against the small increase in productivity of using a mocking tool over hand-coding of mocks.

6.3.5 Developmental Configuration Management/Version Control.

A version control system tracks, groups, and stores all sets of changes to the source and configuration files comprising a project. Typically, a server and client cooperate to allow developers to commit files to a repository, manage file change collisions among multiple users, and revert files to specific versions.

To have an automated build and test environment, a version control tool is required for DCM. The version control tool will be the main interface between the software developers and the main build.

Since CruiseControl and other continuous integration tools have support for most popular version control systems, a software organization’s standard tool can normally be used. If a tool is not supported, CruiseControl allows implementation of plug-ins to add support for further tools. GE Aviation has performed this task to allow use of Serena® Dimensions®, which was previously unsupported.

6.3.6 Code Coverage.

Code coverage tools are already widely used in the development of avionics software to identify portions of system not well-tested or even to find obsolete and unused pieces of production code.

Execution of a code coverage tool is readily automatable. Including it in the one-step build allows the addition of code coverage in the pass/fail criteria.

By integrating code coverage into the continuous integration build, one is able to understand code coverage issues earlier in the lifecycle. This can be done by producing a report with every
build showing the code coverage analysis. Or, taken to an extreme, a build could be made to fail unless there was complete code coverage.

6.3.7 Static Analysis Tools.

Static Analysis tools analyze the source code or object code of the software without executing the software.

Most static analysis tools provide functions for detecting design flaws, naming violations, common performance problems and language specific errors. Many problems that static analysis tools detect are often called out in coding standards. Static analysis tools can be used to demonstrate adherence to coding standards.

More advanced static analysis tools use a set of rules to evaluate source code for looping errors, buffer problems, pointer issues, and related matters.

Static analysis tools can be an excellent complement to dynamic analysis (as performed by automated tests). Static analysis attempts to prevent outlier conditions, such as those that may not otherwise show themselves until the system has been run continuously for long periods of time.

When used as part of the continuous integration, the errors that static analysis tools can detect are caught earlier and cost less to fix.

6.3.8 Documentation.

Documentation systems generate developer and system documentation from the source code of a project. This type of functionality usually requires that comments be placed in the source files using a standard notation. Each documentation system will often have its own preferred documentation notation standard while also supporting several others. Comment notation styles are often simple formats readable by both a developer and the documentation tool.

The choice of documentation system often depends on the specific language in which software is being developed, but most systems can also be configured to generate documentation from any file using the proper notation.

It may not be desirable to perform document generation as part of the immediate build cycle, since it is time-consuming and can give no pass or fail indication.

It is better employed as parts of a build scheduled overnight, thereby consuming fewer build machine resources, but still being up-to-date enough to be of benefit.

6.3.9 Wiki.

Wikis are knowledge management systems that take the form of a collaborative website that can be edited directly.
Wikis can be usefully employed as part of the development process to capture information that is outside the scope of requirements and is not specific to system documentation. Examples of such information include instructions for setting up development systems, coding standards, contact lists, etc.

Wikis are particularly useful since they encourage up-to-date documentation by requiring little effort to update them. When continuous integration is employed, it is crucial that the build can be accurately recreated, for example when a new developer joins the team. A Wiki allows this information to be easily documented and easily updated when errors or omissions are discovered. This information can be easily transferred from the Wiki to formal documentation such as a version description document when development is complete.

6.4 PROCESS CHANGES.

6.4.1 Build Rules.

Build rules are required to ensure that the use of continuous integration environment is consistent and reliable for each developer.

The rules employed will vary between organization based on existing company processes and roles and the company culture. These rules only apply within the development cycle of design, implementation and test, prior to formal testing.

Typically, the rules will suggest that the individual who checked in code that broke the build will be asked to fix it (other punishments will vary by organization). Other rules will be introduced to reduce the risk of breaking the build. See table B-1 in appendix B for an example.

6.4.2 Test-Driven Development.

As the name suggests, with TDD the tests come before all other parts of the development. In fact, the tests will also be a driver for some of the requirements analysis.

One obvious issue with TDD is that it cannot be retrofitted. However, TDD can be practiced when developing new functionality for a legacy project, although existing code may need to be modified to make it testable.

In general, the main benefits of TDD (as part of automated build and test) are covered in section 4.1. The key advantages of TDD (test first), overwriting the code and then writing the test, are as follows:

- When deriving low-level requirements, test cases are a useful communication tool between the development team and the domain experts.
- TDD reduces the probability of developing software that does not meet the undocumented requirements of the system.
TDD forces the developers to produce testable software. The test case will specify the interface to the software. Testable software usually equates to better designed, more maintainable, and more extensible.

The tests can take the place of debugging often making it easier to find and fix problems in the software [21].

In theory, practicing TDD should result in 100% code coverage of the developer’s tests. This is because each line of application code is in response to a test case, and the practice encourages the developer to construct the simplest possible solution (i.e., not adding code for features not currently required or tested).

6.4.3 Verification Guidelines

The verification process for automated tests is similar to that for manual tests. The main added complication is that of qualifying the verification tools to be used. Tool qualification is covered in section 5.1.

Once the tools are qualified, anything built on top of those tools must be manually verified. For the BA609 HUMS, a checklist was developed that was refined as part of this research.

The BA609 HUMS checklist often refers to “Verified” calls. These are calls made by the automated test code to report acceptance tests that have passed, which help with the traceability between requirements and tests.

For the BA609 HUMS, this was used to report Dynamic Object-Oriented Requirements System (DOORS) requirement identifiers that uniquely identified each requirement. This allows one to compare requirements tested versus actual requirements, ensuring full functional test coverage and to provide a list of all the requirements tested.

6.4.3.1 Checklist

6.4.3.1.1 Check Test Consistency With Test Documentation or With Tested Functionality

This will vary depending on how the test is documented. A document may contain a list of the test procedures along with traceability to the requirement it intends to test.

The code itself may provide the links to complete the traceability to the requirements.

There are many scenarios in which this check is important. Causes for this check to catch an error include requirements that have changed or been removed without the test code having been updated accordingly or copy and paste errors where a whole test is copied but either the assertion or verified call is not updated correctly.
6.4.3.1.2 Ensuring That the Test Fails if Functionality Fails.

If functionality fails, it is imperative that the test cannot pass. This usually happens if the assertions are incorrect or are avoided by the code flow. Code coverage analysis helps spot the former case; the latter case can only be detected manually.

An example of this verification error was detected where the requirement was to send a message if an event happened. The automated test called the functionality to raise the message, and then had an assertion in the message handler to assert that the message was correct. However, if no message was sent, the test still passed. This was not the case in the example since the application code did work. What was missing was an assertion after the functionality call to assert that the message handler had been entered.

6.4.3.1.3 Requirements Verification Criteria.

Requirements may be marked as verified even though they have not completely tested within the test fixture. This can happen if a requirement needs two test cases to be verified, but it is still desirable to mark it in the code as tested.

For example, the requirement may read “Serial numbers and start times shall be displayed for Left & Right Engines,” but the serial numbers and start times are tested separately in a different test.

The solution is usually to either split into two requirements, or to move all the test code into the same test fixture.

6.4.3.1.4 Adequate Test Data.

Where applicable, it is necessary to check that the test data inputs are adequate for the test and that the expected results are consistent with the inputs.

6.4.3.1.5 False Positives.

It is necessary to check for tests that can give false positives (i.e., certain conditions are considered passes that should be considered failures). This often happens because of inadequate understanding by the tester of the control flow through the software.

False negatives will cause problems by causing the tests to fail unexpectedly or intermittently; they are discovered when they happen, and do not need to be explicitly sought in review.

6.4.3.1.6 Boundary Cases.

As with manual testing, it is important to test boundary cases. Take the example of a function that takes a name and returns true if the person is male. To adequately test the function, one will at least need to pass in both a male and female name. There may also be failure cases, for example, a name that is not recognized.
Selection of boundary cases is implementation specific and requires understanding of the algorithm implementation.

6.4.3.1.7 Test Source Code Follows Coding Standards.

Coding standards for test source code should be employed mainly for consistency of style and to outlaw dangerous practices. The quality of the test code is vital since it is often used as a guideline for using the application code, and will be frequently updated when performing continuous integration.

6.4.3.1.8 Other.

If the code is being read for the verification review, it is cost-effective to combine this review with other review activities. An example of this is section 6.4.3.1.7, since this check is not strictly required for a verification review but is of great benefit during development.

7. SYSTEM FUNCTIONALITY PARTITION.

HUMS are made up of many different functions. To partition different functions of the HUMS means that the functions are designed to not affect each other except through the defined interface. For example, if a HUMS has two functions, RTB and Structural Usage Monitoring (SUM), it may be possible to partition those functions, allowing the assumption that a change to RTB will have no impact on SUM and vice versa.

The partitioning method allowed will vary by certification level. For example, for lower levels of certification, it may be sufficient to have the functions partitioned by running in separate processes within the operating system. Whereas, for higher levels, those functions may have to run on dedicated processors or in a partitioned operating system.

Some functions of a HUMS may be independent by definition. For example, the ground-based RTB function typically will be completely independent of the SUM. Other functions may be partitioned by running in different processes within an OS or by running on a dedicated processor.

Automated testing and partitioning complement each other, and can both be used to reduce the time and cost of certifying a HUMS.

7.1 PARTITIONING BY CRITICALITY.

Different functions of the system may have different criticality levels arising from the Functional Hazard Analysis.

The requirements for certifying (and hence, the time and cost) increase proportionally with the criticality level. By isolating functions that have lower criticality levels, the effort of certifying these functions can be reduced.

Often, an investment is made to partition functions that may change criticality even if their current criticality levels are the same.
7.2 PARTITIONING BY TESTABILITY.

Ideally, the entire system would be automatically tested. There may be factors, however, that stop certain parts of the system from being tested this way. Typically, these are parts of the HUMS with complicated external interfaces to the users, to the HUMS sensors, and to other aircraft hardware.

In these situations, partitioning can be used to isolate those parts of the system that are difficult to test.

7.3 PARTITIONING BY PROBABILITY OF CHANGE.

As discussed in section 4.2.1.1, the HUMS is a continually evolving system. However, certain areas of the system will be far less likely to change after initial certification.

Again, ideally the entire system would be automatically tested, in which case the amount of change is no longer an issue.

However, for parts of the system not covered by automatic testing (or as an alternative to automatic testing), partitioning can be used to isolate parts of the system that are likely to change from those that will not.

7.4 CURRENT PARTITIONING.

Typically, HUMS is only partitioned at a high level, usually the natural partition of ground-based and aircraft-based. In some cases, the aircraft-based portion is partitioned further by criticality of function; however, this is rarely the case for the ground-based station.

Often, the scope and usage of HUMS is reduced to allow a lower criticality. This reduces initial time and cost, and time and cost of subsequent maintenance. However, higher certification levels are going to be necessary to exploit the full benefits that HUMS can offer.

The authors currently have no knowledge of any attempts to partition HUMS to a more granular level or of any attempts to partition by testability or probability of change.

8. COST/BENEFIT ANALYSIS AND RECOMMENDATIONS.

8.1 RETROFITTING FOR LEGACY HUMS.

The definition of legacy HUMS here refers to HUMS that have fully developed functionality but without any incorporation of automated build and test technologies.

The steps in table 3 are successive stages of implementation of a completely automated and tested build in the typical order that a project would implement them. For the most part, later steps build upon the earlier steps.

A project may implement as many of the stages as it sees fit, stopping when the returns are less than the cost of implementation.
Table 3. Technology With Cost and Benefit of Retrofitting

<table>
<thead>
<tr>
<th>Technology Level</th>
<th>Estimated Cost (in hours)</th>
<th>Cumulative Cost (%)</th>
<th>Cumulative Benefit (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. One-step build</td>
<td>100</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>2. Continuous integration (build machine)</td>
<td>50</td>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td>3. Execution of target application</td>
<td>50</td>
<td>4</td>
<td>25</td>
</tr>
<tr>
<td>4. Few broad shallow automated tests</td>
<td>100</td>
<td>6</td>
<td>40</td>
</tr>
<tr>
<td>5. Automated tests for all requirements</td>
<td>2200</td>
<td>50</td>
<td>85</td>
</tr>
<tr>
<td>6. Unit tests for all classes</td>
<td>2000</td>
<td>90</td>
<td>99</td>
</tr>
<tr>
<td>7. 100% code coverage for test execution</td>
<td>500</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>

Along with each technology is an estimate of the cost and benefit to a project of a similar size to the BA609 HGS (~800 thousand source/software line of code). The BA609 HGS is just above level 5, with 98% requirements testing coverage and a reasonable number of unit tests written.

The cost of each stage will differ depending on the complexity of the project. For example, a one-step build is simpler if all source is compiled using the same compiler.

Figure 5 shows that after level 4 (Few Broad Shallow Automated Tests) the returns start to diminish, and for legacy projects the decision on how far to go will probably depend on the life of the project. It is expected that it will always be a good investment to get to level 4.

Figure 5. Cost and Benefit of Retrofitting Automated Build and Test Technologies
8.2 GREEN-FIELD HUMS.

Green-field HUMS are new HUMS that introduce these autobuild and test technologies from the outset. The majority of the initial start-up cost is primarily due to the setup of the build machine (since there is initially no code to build or test), therefore getting to level 3 is straightforward.

From GE Aviation’s experience on adding the new functions required for the BA609 HGS, having continuous integration means that the new functions can be developed for less, even if both unit tests and acceptance tests are added as part of the task. In fact, the actual cost was only 51% of the original budget for these tasks. The original budget had assumed a large cost of reintegrating, retesting, and fixing other areas of the system affected by the change. The recommendation is to go to at least level 5 for green-field systems.

9. RESEARCH RESULTS AND CONCLUSIONS.

9.1 CONCLUSIONS.

Automated build and test tools are already quite mature and, once technical expertise is achieved, the main roadblock to rollout is developer understanding. Developers need training in the process as well as on the tools.

The benefits of automated build and test are well understood and exploited throughout the software industry. The assumed difficulties of initially aligning with DO-178B certifiable processes were actually straightforward. All the benefits of automated build and test still hold true when accomplished under DO-178B.

Automated testing and continuous integration should go hand in hand. Most benefits of automated testing will not materialize unless continuous integration is employed.

There are benefits for both legacy and green-field Health and Usage Monitoring System (HUMS).

All HUMS software developers should try to exploit this technology.

9.2 GENERAL RESULTS.

Following on from this research, future HUMS software developments should be able to:

- Implement automated testing of HUMS ground software and replicate most processes in this research and demonstration project.
- Implement automated testing of HUMS airborne software; replicate many processes in this research and demonstration project, and translate others to the embedded domain.

This should realize, to some extent, all HUMS benefits outlined in section 4.2.
9.3 STATUS OF TECHNOLOGY READINESS LEVELS.

The Technology Readiness Levels (TRL) and assessment guide used were those supplied by the Federal Aviation Administration, adapted from GAO/NSIAD-99-162 Best Practices, Appendix I, Technology Readiness Level Descriptions.

The technical proposal for the research identified the initial TRL for automated testing of HUMS in general as TRL 6 (system/subsystem model or prototype demonstration in a relevant environment). This was based on the introduction of the technology during the BA609 HUMS Ground Station development by GE Aviation.

Sections 9.3.1-9.3.3 identify a particular TRL for each HUMS subsystem where automated testing could be applied.

9.3.1 Automated Testing of the HUMS Ground Station.

The 2006-2007 research focused on this area and provided an easy to follow demonstration project along with DO-178B software verification tool qualification. Internally at GE Aviation, this has been used as a basis for tool qualifications of similar tools by other teams. In the ground environment, there is a wealth of tools and accessible knowledge available. Other vendors should be able to easily introduce automated testing to their ground products.

The automated testing of the HUMS ground station had a TRL of 6 at the start of the program, and a TRL of 8 upon completion.

9.3.2 Automated Testing of the HUMS Airborne System.

The 2006-2007 research covered this area and discussed techniques for dealing with typical problems encountered during automated testing. The unit-testing framework tool qualification will simply follow from our JUnit qualification; with no extra artifacts required for higher DO-178B safety criticality levels (see section 5.1.3) that are more likely to be required for airborne software. The majority of tools and techniques used in the embedded environment are identical to the ground environment. However, until verified by a targeted demonstration project, the use of this technology in the embedded environment may be more difficult for other vendors to exploit.

The automated testing of the HUMS airborne system had a TRL of 3 at the start of the program, and a TRL of 6 upon completion.

9.3.3 Automated Testing of the HUMS Configurations.

This is a subsystem of the HUMS, which is not software itself, and is outside the scope of this research. However, the benefit of automated testing in this area is clear, but there has been no investigation into the challenges involved. Appendix A discusses this in detail.

Due to the lack of research, the automated testing of the HUMS configurations did not have a TRL at the start of the program, and had a TRL of 2 upon completion.
REFERENCES


APPENDIX A—AUTOMATED TESTING OF HEALTH AND USAGE MONITORING SYSTEM CONFIGURATIONS

A typical Health and Usage Monitoring System (HUMS) will allow configuration of the measurement and diagnostic so that it can be reused on different aircraft types. This configuration forms a subsystem of the HUMS that requires validation as part of the certified system. The costs of verifying this configuration are disproportionately higher than the development costs.

Testing a HUMS configuration presents unique challenges to those of software.

- **Amount of configuration to test**
  The HUMS monitors many components in many different flight conditions, and produces many health indicators for each component.

- **Knowing what is the expected result**
  Prior to use on the aircraft, testing has to be based on known fault cases, with most test data needing to be synthesized.

- **Execution and Simulation**
  The real HUMS vibration system has many inputs and integrated systems. Ideally, the configuration could be tested in a partly simulated software environment. However, there would be many challenges to this simulation.

- **Frequency of Change**
  The HUMS vibration configuration needs to be matured based on flight testing on the aircraft. During flight test the configuration will change many times. During the first year of maturity on revenue flights, the configuration may need to change another few times. After that, the changes will become less frequent and may occur only when new fault cases become detectable.

- **Long Duration of Measurements**
  HUMS vibration measurements may take a long time to execute. If the testing needs to be manned, this becomes an issue.

- **Availability of Target Hardware**
  The HUMS target hardware may be expensive or in short supply. Ideally, a HUMS applications engineer could test a vibration configuration from their own personal computer (PC).

- **Awkward Current Test Environment**
  Current test environments will include signal generators, PC-based simulators, and the target hardware. These systems can take time to set up and may need to be changed depending on which aircraft configuration is being tested.
• Configuration is an overlooked aspect of HUMS development. The development costs of the HUMS Configuration are often underbudgeted and left until the end of the development. As a result, the verification strategy is often compromised for short-term savings.

During flight test, it is best to avoid loss of confidence in the HUMS and maximize the amount of time the HUMS is functioning on the aircraft. A broken HUMS configuration could cause the entire HUMS to fail.

Testing should be considered in system and software design so that simulation software can be easily added and there is no reliance on the target hardware. Ideally, all testing should be able to occur on the desktop PC of the application engineer who is developing the configuration.

Software will need to be developed for the following functions.

• Simulating the acquisition software that acquires the raw data from the accelerometers, tachometers, rotor blade trackers, and other HUMS sensors.

• Capturing the HUMS outputs. In the case of HUMS data written to file, no work is needed since these outputs can be easily compared with those expected. In a case where the HUMS provides a real-time response (e.g., to a cockpit display unit), those outputs need to be diverted by software and compared to expected results.

• Producing synthetic raw aircraft data for each sensor. A tool is useful here (since it may be time consuming) although not absolutely necessary since this is not a repeated step.

• Executing the test and comparing actual with expected results
APPENDIX B—EXAMPLES OF PROCESS CHANGES

Table B-1 shows examples of build rules that GE Aviation used for the BA609 Health and Usage Monitoring System (HUMS).

Table B-1. Example Build Rules—BA609 HUMS

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Before committing any changes, update your local copy to get other developers’ changes, then re-run the build and test. If the build machine is currently building, wait until it is done before checking in.</td>
</tr>
<tr>
<td>2</td>
<td>After committing any changes, check that the build machine is still running before leaving.</td>
</tr>
<tr>
<td>3</td>
<td>If you have broken the build, it is your responsibility to fix it.</td>
</tr>
<tr>
<td>4</td>
<td>Do not commit when the build is broken, unless you are fixing the build. Rule 1 should prevent this.</td>
</tr>
<tr>
<td>5</td>
<td>If you run out of time when fixing the build machine, it is better to undo what has broken the build than to leave it broken.</td>
</tr>
</tbody>
</table>

Table B-2 shows a HUMS ground station (HGS) with the following requirement:

The HGS shall display separate sorted lists of Components, Accelerometers, Flight Regimes, and Condition Indicators (CIs).

Table B-2. HUMS Ground Station Requirement Example

<table>
<thead>
<tr>
<th>CIs</th>
<th>Components</th>
<th>Accelerometers</th>
<th>Flight Regimes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1R</td>
<td>Left Engine</td>
<td>Lateral</td>
<td>Ground</td>
</tr>
<tr>
<td>2R</td>
<td>Right Engine</td>
<td>Vertical</td>
<td>Hover</td>
</tr>
<tr>
<td>3R</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

By giving a developer a requirement like this, they will most likely notice that all values are strings and write code to sort all the strings alphabetically. However, by looking at some test cases, things may not be so straightforward. Table B-3 shows some inputs for CIs.

Table B-3. Example Test Cases for Sorting CIs

<table>
<thead>
<tr>
<th>Test</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[1R, 2R, 3R]</td>
<td>[1R, 2R, 3R]</td>
</tr>
<tr>
<td>2</td>
<td>[3R, 1R, 2R]</td>
<td>[1R, 2R, 3R]</td>
</tr>
<tr>
<td>4</td>
<td>[1T, 1R, 2R, 2T, 3T, 3R]</td>
<td>[1R, 2R, 3R, 1T, 2T, 3T]</td>
</tr>
</tbody>
</table>

R = Main rotor    T = Tail rotor
The first two cases are straightforward. The third case makes one realize that a standard computerized string comparison will not suffice; 10R would be in between 1R and 2R.

The fourth case shows that the harmonics should be grouped by Rotor. Table B-4 shows some inputs for components.

Table B-4. Example Test Cases for Sorting Components

<table>
<thead>
<tr>
<th>Test</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[Left Engine, Right Engine]</td>
<td>[Left Engine, Right Engine]</td>
</tr>
<tr>
<td>2</td>
<td>[Right Engine, Left Engine]</td>
<td>[Left Engine, Right Engine]</td>
</tr>
<tr>
<td>3</td>
<td>[Right Engine, Left Engine, MWGB]</td>
<td>[Left Engine, MWGB, Right Engine]?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[Left Engine, Right Engine, MWGB]?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[Left Engine, Right Engine, Left Prop Rotor, Right Prop Rotor]</td>
</tr>
<tr>
<td>5</td>
<td>[MWGB, TGB, Right Engine, Left Engine, Right Prop Rotor, Left Prop Rotor]</td>
<td>?</td>
</tr>
</tbody>
</table>

MWGB = Mid-wing gearbox
TGB = Tail gearbox

By the third test case, it has been identified that the developer just may not want to sort the lists alphabetically. The correct behavior will depend on the application, and possibly on the cost of implementation.

The issues shown in this example are not new. The developer may or may not face these with any traditional development method. The benefit of test-driven development is that it forces the developer to think about the detail of the requirements upfront.