EuroSTAR Software Testing Community

Automated Reliability Testing Using Hardware Interfaces

Bryan Bakker

case study

AUGUST 2013

Introduction

Bryan Bakker tells of automating testing for medical devices, an area with stringent quality requirements and difficulties in accessing the software embedded in devices such as X-ray machines. Bakker and his team's starting point was simple tests that assessed reliability; functional testing came later. The automation was developed in increments of increasing functionality. The chapter contains many interesting observations about management's changing views toward the automation (e.g., management was surprised when the testers found a lot of new bugs, even though "finding bugs" is what management expected of them). The team developed a good abstraction layer, interfacing through the hardware, and were even able to detect hardware issues such as the machines overheating. The results in the test logs were analyzed with inhouse tools. The reliability testing paid for itself with the first bug it prevented from being released—and, in all, 10 bugs were discovered. Subsequent functional testing was smoother, resulting in cutting the number of test cycles from 15 to 5. This story is from the Netherlands, and the project had excellent success using commercial and inhouse tools with just two people as the test automators. See Table 1.1.

Classification	This Case Study
Application domain	Medical device (with embedded software)
Application size	500,000 LOC
Location	The Netherlands
Lifecycle	V model
Number on the project	50
Number of testers	10
Number of automators	2
Time span	1.5 years
Dates	2008–2009
Tool type(s)	Commercial, open source, inhouse
Pilot study undertaken	Started on one project, later spun off to other projects
ROI measured	Yes, based on defects found and test cycles reduced from 15 to 5
Successful?	Yes
Still breathing?	Yes

Table 1.1 Case Study Characteristics

Background for the Case Study

Sioux Embedded Systems supplies trend-setting services in the area of technical software that is subsequently embedded in equipment. At Sioux, we focus on added value. We continuously look for new skills and expertise, new ways to increase our productivity, new ways to do business: We are driven to create innovative solutions for our customers. Testing is one of the ways to ensure the high quality of the systems we deliver. Our team of testers specializes in test design and execution as well as testconsulting services. Sioux strives to provide these testing services in close cooperation with development in order to have short feedback loops to the developers

One of our customers manufactures medical X-ray devices for patient diagnosis and patient surgery. The application described in this case study is used during all kinds of surgical procedures. The system generates X-rays (called exposure), and a detector creates images of the patient based on the detected X-ray beams (called image acquisition). The image pipeline is in real time with several images per second, so the surgeon can, for example, see exactly where he or she is cutting the patient. During an operation, X-ray exposures are performed several times. The frequency, duration, and characteristics of the beam can be adjusted and optimized for the type of surgery being performed. The X-ray exposure should be as small as possible but should still result in sufficient image quality.

In the R&D department of this specific customer, new devices are being developed—the software as well as the hardware, electronics, and mechanics. Quality is of utmost importance in these applications because failure of the system can have negative consequences. Testing is a critical part of each project and consumes about the same amount of effort as the development tasks. All testing activities are performed manually, resulting in a developer–tester ratio of about 1 to 1.

My role was to improve the test process by finding ways to introduce automated testing. The focus of automation included but was not limited to software testing.

1.2 The Need for Action

Although testing is considered important in the customer's organization, existing systems at customer sites had reliability issues. The system was always safe, but the functions executed by the customer (e.g., transfer of acquired images to external devices) now and then failed to perform as expected, resulting in crashes of the system. The development department had devoted a lot of money and effort to fix these issues and release them to the field. This necessary effort greatly impacted other projects that were working on new functionalities and other systems. These projects had to delay their milestones because of restricted resources, both development and test resources.

When a new project was started with new functionality to be developed on comparable systems, reliability was one of the attention points of the project. Senior management demanded action to increase system reliability. Several improvements were considered: FMEA (failure mode and effects analysis), reliability requirements, reliability agreements with suppliers, design measures to increase robustness, and (automated) reliability testing.

To avoid long discussions about requirements that were not yet fully understood, we did not try to define reliability requirements early on but would address them once the notion of reliability became clearer. Therefore, we focused on automated reliability testing first to demonstrate that some problems could be addressed before the product was released. The other ideas would be tackled in future projects. Our rather small reliability team consisted of a hardware engineer, a developer, and a tester, all also working on other projects. Because the available effort was very small, we decided to develop a "quick and dirty" solution at first. The idea was that with a first increment of a test framework, we could run simple test cases repetitively in order to execute duration tests focusing on the reliability of the system. The test framework would not need to schedule test cases or evaluate and report the pass/fail criteria in a nice way. In fact, the test cases would not have pass/fail criteria at all. The test framework would only deliver a log file provided by the system under test (SUT). Test case maintenance was not yet important. The only requirement was that the system would be "driven" via hardware interfaces, so no special test interfaces were implemented in the software.

The expectation was that the first few test cases would identify several reliability failures, which would need to be analyzed and fixed. This would create awareness in the project, freeing up more resources.

GOOD POINT
A small-scale start can help to raise awareness of the benefits of automation.

1.3 Test Automation Startup (Incremental Approach)

The first increment of the test framework consisted of hardware interfaces from the system (such as buttons on the keyboard, hand switches and foot switches to start the X-ray acquisition) that were accessible from LabVIEW by simulating hardware signals. This approach was chosen because it was not the intention to change the embedded software of the system. The software development crew was busy with the implementation of new features. New feature requests from the test team would certainly be rejected. LabVIEW was chosen because it was already used by the hardware development team to perform measurements and tests on different hardware components. The architecture of our first increment is shown in Figure 1.1.

The first few test cases were designed in LabVIEW using the graphical language that is part of the tool. We focused on duration tests because they are small and quick to implement but can be executed for several days on the system. They require little effort but result in a lot of testing time and such test cases can reveal some nasty defects. Within several weeks of development, the first test cases could be executed at night and over the weekends when the systems were not used. In the first weekend of testing, several different startup problems were discovered. In the following weeks, the test cases were slightly extended: More X-ray acquisitions were performed before the system was switched off again, and more defects related to the increased system usage were found. The development crew was busy analyzing and fixing the problems, and as a consequence, the next milestone (an internal delivery) was delayed. Some bug-fixing time had been scheduled in the plan, but it was by far not enough to fix all the issues. Senior management was not pleased about the missed milestone, but they also realized that the quality of the delivery was being improved, which is exactly what they asked for in the first place.

LESSON
Don't forget to include time to fix defects and retest.

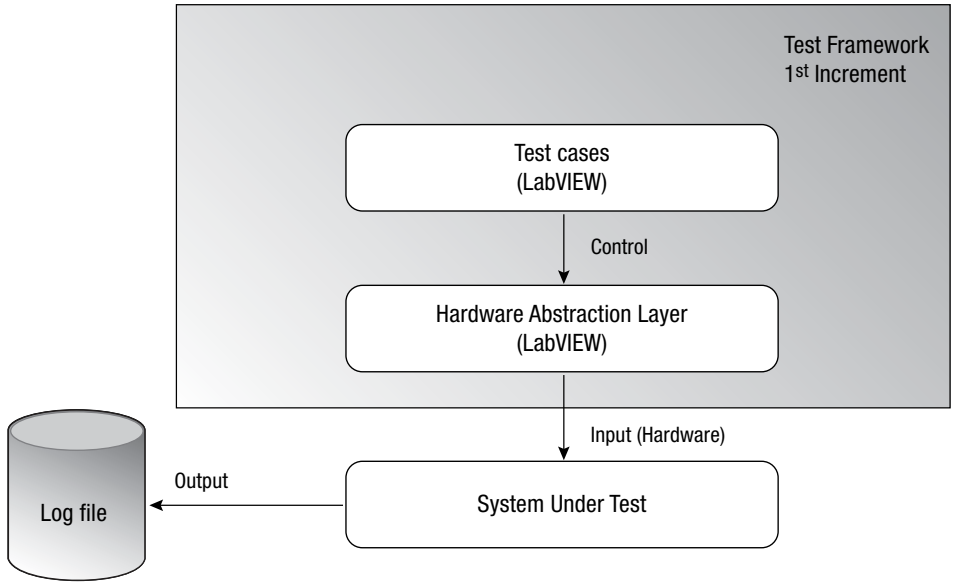

Figure 1.1 Architecture of the first increment

The chosen framework could only perform actions on the SUT, but no verification of whether the system indeed performed as expected was done at this time. The system generated a log file containing all kinds of information, both for service engineers and development engineers. This log file was analyzed for failures after the test ran. This, of course, took some extra effort but also provided evidence of failures for the development department.

1.4 Buy-In from Management

Now it was time to get some buy-in from management. Management heard about defects being found by the "new way of testing" but they still did not thoroughly understand what was going on. Some defects looked like duplicates of defects already communicated by end customers using old versions of the system. It seemed that the reliability tests detected several issues already known to be present in the field and also present on the yet-to-be-delivered system. These defects were not yet reproduced in the development department, so no solutions were available yet. With the automated testing, these issues were fairly easy to reproduce. As we all know, reproducibility is very important to fixing defects.

GOOD POINT
Automated tests are reproducible, which is essential for knowing that a defect has been correctly fixed.

Several presentations were given to senior management, project leaders, and product managers about the approach taken with automated testing and the most recent results, mainly the number and the kinds of defects detected. It was also clearly communicated that the test cases developed and to be developed focused on reliability only, not on functional or other nonfunctional items.

A definition of reliability for our system was specified on the basis of the primary functions of the system. Failures that impact the (defined) reliability are called reliability hits (see Figure 1.2).

Following are some examples to illustrate the different categories:

- Primary failures (problem in primary function):
  - System startup results in a nonworking system (failed startup).
  - Running acquisitions are aborted.
- Secondary failures (problem outside primary function but impacts primary function):
  - Viewing old/archived images was not defined as a primary function, but when viewing these images results in a nonworking system (e.g., due to a software crash). This does count as a reliability hit.
- Tertiary failures (no direct impact on primary function):
  - Not all images are stored correctly on a USB medium. This is not a primary or secondary failure because USB storage media are not considered to be used for archiving. For archiving of medical images, other independent and more reliable systems exist. Thus a failure in this storage function does not count as a reliability hit.
  - X-ray images cannot be printed due to an internal failure.

Defects detected by our reliability tests would almost always be critical defects because they affected one of the most important characteristics of the system. On the other hand, the automated test cases did not eliminate the need for manual test cases; they were

performed in addition to manual tests. As a consequence, the contents of the upcoming test phases (system test and acceptance test) remained the same.

Management demanded action toward increased reliability but had no idea how to address the challenges—nor did the quality assurance department, under whose purview reliability fell, know how to approach the issues. The ideas came from engineers (lower in the hierarchy) and focused on the short term.

The early solution showed senior management that the project team was making progress. Even with the first increment of the test framework, the added value was evident.

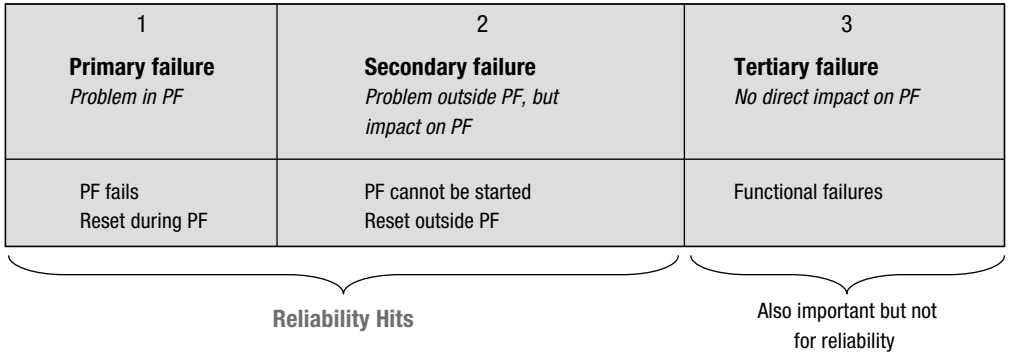

Figure 1.2 Definition of a reliability hit related to Primary Function (PF)

With our definition of reliability, it was also possible to define measurable reliability requirements. Because of the chosen focus on automated reliability testing, developing measurable reliability requirements was postponed and defined as an improvement opportunity for the future.

1.5 Further Development of Test Framework

1.5.1 Increment 2: A Better Language for the Testers

After management were convinced that the test automation approach was the road to follow, the team received resources and funding to further develop the test framework (the test framework, consisting of dedicated hardware and LabVIEW, was not cheap). The next step (increment 2) was to abstract from LabVIEW because its programming language was unsuitable for defining test cases. LabVIEW turned out to be difficult for our test engineers to learn, and we expected that maintainability of the test cases would be an issue in LabVIEW. Our test engineers had only basic programming knowledge, so using the same programming language as the software developers used was not an option. We decided to use a scripting language for its flexibility and ease of use. Ruby was chosen because it is a dynamic open source scripting language with a focus on simplicity and productivity. It has an elegant syntax that is natural and easy to write, lots of support information was available, and Ruby interfaces well with LabVIEW. The hardware abstraction layer was still programmed in LabVIEW, but the test cases could be programmed in Ruby now, hiding the complexities of LabVIEW. Standard functionality of the test cases was implemented in libraries, so the test cases were straightforward, and maintainability of the test cases increased. Because we intended to use the framework in the future not only for reliability tests but also for functional tests, we were already taking maintainability into account.

Figure 1.3 shows the architecture of the second increment of the test framework.

1.5.2 Increment 3: Log File Interpretation

The third increment added log file interpretation. The log file provided by the SUT contained lots of information, such as all actions performed by the user, and important internal system events. This log file was now provided to the test framework (via the hardware abstraction layer), and test cases could interpret the log file during the execution of the test case. The test cases scanned the log file looking for specific regular expressions and could thereby implement pass/fail criteria. In the first two increments, test cases

consisted only of actions; now the corresponding checks were added to the test case.

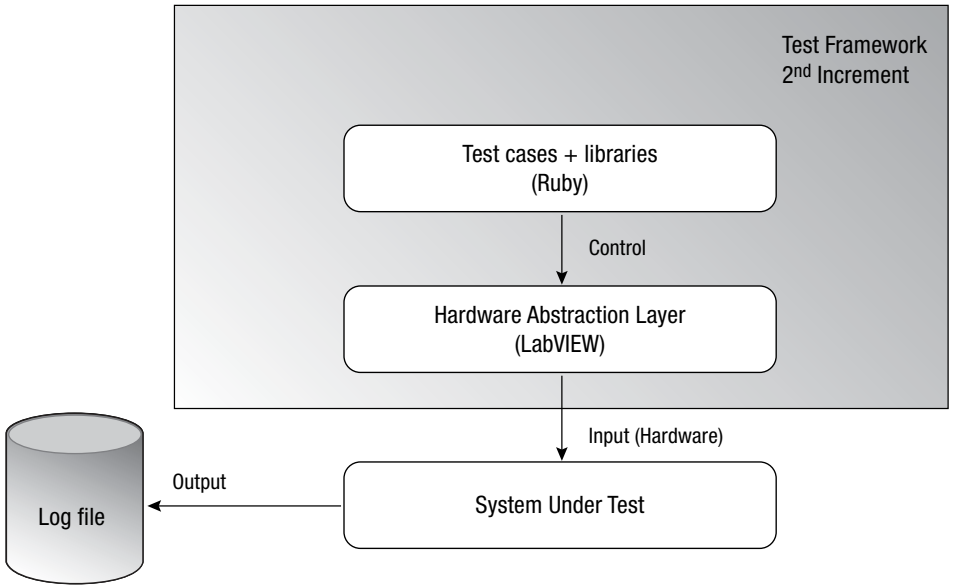

Figure 1.3 Architecture of second increment

Analyzing the log file also allowed the test case to follow different paths based on the information in the log file (although this makes the test case more complex).

This also enabled us to test much more efficiently. When testing the system extensively, as is done during reliability testing, certain parts of the system become too hot, causing the system to enter a cooling down period. In this state, no X-ray exposure can be performed. Most test cases would fail because of a hot system. Now it was possible to detect this condition, and in the cooling down period, power-cycle tests were performed. These tests switch off the system and switch it on again, and then check via the log file whether the startup was successful. The framework monitors the system temperature, and when the system is sufficiently cool, it executes other duration tests. Efficiency was improved because other valuable test cases could be executed during the cooling down period.

Part of the evaluation of the test results was now performed during the test execution by checking the pass/fail criteria. However, the log files were still manually analyzed afterward for undetected error conditions and unwanted behavior. (For example, hardware failures are not detected by the test cases but should be detected in the subsequent manual analysis.) Several different small, home-made log file analysis tools were developed to scan and parse log files from the SUT. It was possible to:

- Detect errors in log files.
- Count occurrences of different errors.
- Derive performance measurements from the log file (such as the number of images acquired per second).

TIP

Nonintrusive tests drive the system from external interfaces so that the software is not affected by the test.

Software interfaces were integrated with the framework without changing the software in the system itself. This made it possible for the test cases to use internal software functions to determine certain internal states in order to verify whether an expected state equals the actual state. During test case design, the following rules hold:

- For test case actions, use the external interfaces as much as possible to simulate a real user as closely as possible. Using internal interfaces to trigger actions should be avoided.
- The internal interfaces should be used only to check certain states or variables when they cannot be derived from the log file.

With this approach for the framework, other kinds of test cases could be implemented, such as smoke tests, regression tests, functional tests, and nonfunctional tests. Because of resource restrictions, these tests were not yet implemented. Focus remained on the reliability tests, but nevertheless the framework also supported these kinds of tests.

A test scheduler was implemented to facilitate the definition and execution of test runs. A facility to generate a report that gave an overview of the executed test cases and their result was also implemented. The test cases (scripts), the test framework, and the test results were stored in a central repository, the same as used for the software and documentation of the project. This ensured that the set of test cases was consistent with the software and the documentation and prevented incorrect versions of the test cases from being used to test the software and the system. This was a further step into professionalism.

Other improvements to the test framework were already defined, such as reading in

hardware signals and retrieving images from the image pipeline. The implementation of these improvements would be performed in upcoming increments.

Reliability is not a binary characteristic. You cannot say, "The system is reliable." It is a quantitative characteristic expressed, for example, in mean time between failures (MTBF). We wanted the new system to be more reliable than the legacy system, so we needed to measure the reliability on these legacy systems with the same set of test cases. Because these test cases use hardware interfaces to trigger actions on the system, the test cases could also be run on older versions of the system, as these hardware interfaces were never changed. We could do so without altering the framework or the test cases. Now it was possible to clearly compare the existing (legacy) system with the newly developed system with respect to reliability.

GOOD POINT
Reliability is not binary but a characteristic measured by a scale of varying values. A good comparison point is the previous system.

Figure 1.4 shows the architecture of the third increment of the test framework.

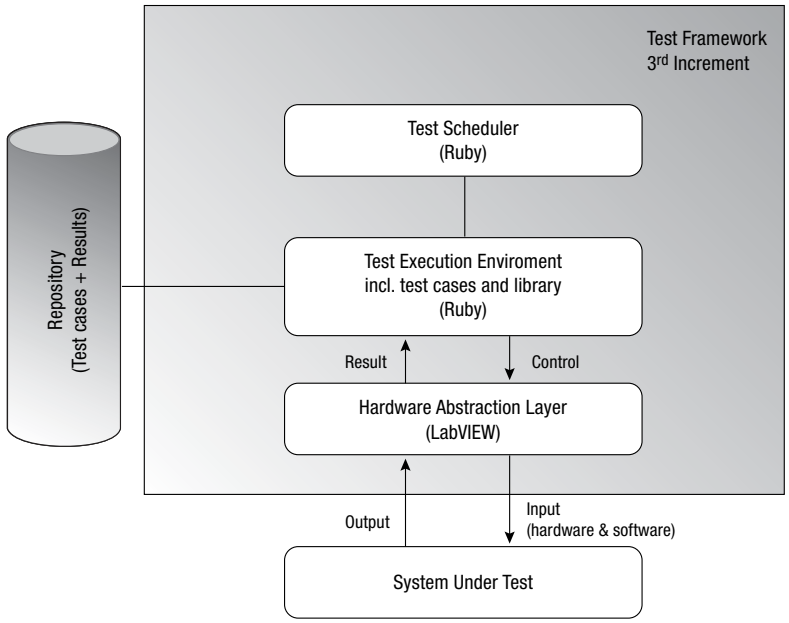

Figure 1.4 Architecture of the third increment

1.6 Deployment and Improved Reporting

During the development of the different increments of the test framework, a return on investment (ROI) calculation was performed to further convince the management of the added value of this reliability test approach and to justify further investments into it. An overview was made of all the defects found with the reliability tests. Together with a service engineer and a systems test engineer, an analysis was performed to identify the defects that would not have been found by later test phases but would certainly be noticed by end customers. The management team was asked how expensive one such defect would be (including costs like penalties, development costs, release costs, service costs). Of course, the real damage would be greater than such a dollar amount—just think about a dissatisfied customer or the impact on the brand of the company. But these are difficult to measure, so we decided to only take the "hard" costs into account. On the other hand, we did calculate the expenses for hardware equipment and resources. Here is our ROI equation:

$$\frac{(\text{NumberOfDefects} \times \text{CostOfDefect}) - (\text{CostEffort} + \text{RemainingCost})}{(\text{CostEffort} + \text{RemainingCost})}$$

where,

- Number Of Defects: Number of defects detected by reliability tests that would not have been detected by later test phases but would be encountered by end customers and would require field updates.
- Cost Of Defect: Average (hard) costs of resolving a defect found in the field and re-releasing the product to the field.
- Cost Effort: The total cost of the effort spent on the development of the test framework and the development of the test cases.
- Remaining Cost: Other costs such as hardware purchased, hardware developed, and licenses.

When this equation becomes larger than 1, the added value of the test framework has exceeded its costs, so we have positive ROI. In our situation, that was already the case when only one defect had been found, because the solution of defects in the field is very expensive. The fact that 10 potential field problems had been prevented was positive for the visibility of the test framework in the organization. This was communicated to the entire organization by means of mailings and presentations.

GOOD POINT
Make the benefits of automation visible. For example, show how defects were prevented from escaping to the field (and to customers).

To improve communication about the results of the tests to management, reliability growth plots were generated that depicted the growth (or decline) of the reliability of the system over time. The Crow-AMSAA model for reliability measurements was chosen because the test results contained different kinds of tests on different system configurations with different software versions installed. Other models are more applicable for test results on fixed system configurations.

A fictitious example of a Crow-AMSAA plot is shown in Figure 1.5. The X-axis represents the cumulative number of startups of the system. The Y-axis shows the MTBF of the startup behavior of the system. Each triangle in the plot depicts one test run, and the solid line is the line with a best fit through all triangles. This line can be used for reliability predictions by extrapolating it to the future. The exact meaning of this plot is not related to this case, but it can be seen that these plots are fairly easy to read and very understandable, even by nontechnical personnel, after some explanation. Management can monitor the reliability trend in the project and take corrective actions when needed.

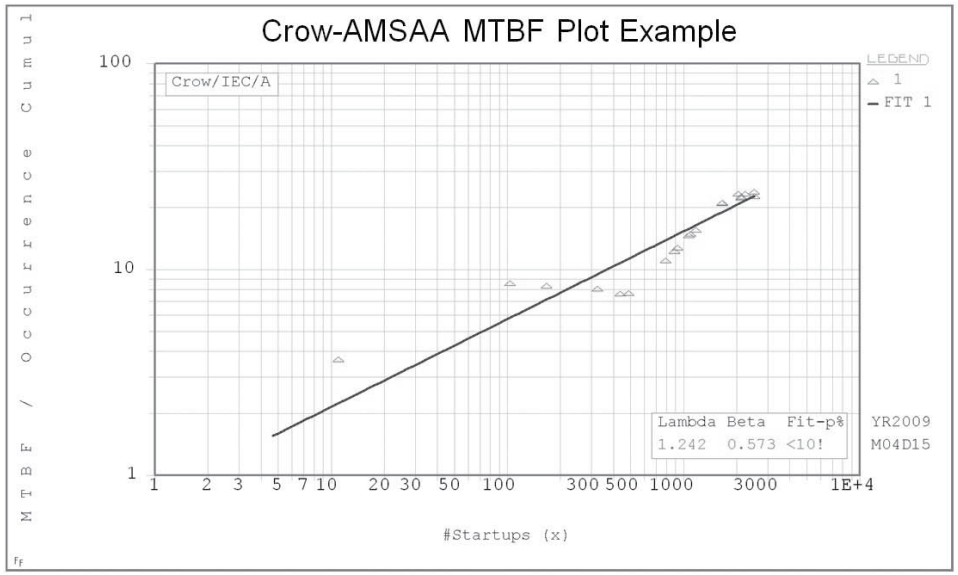

Figure 1.5 Crow-AMSAA plot

All the information needed for these plots was automatically extracted from the log files, which are stored during the reliability tests overnight and over the weekends.

GOOD POINT
Automate the reporting of test results in a way that is easily understood by managers.

Several different plots were designed, not only for the reliability of the startup behavior but also for the reliability of the exposure functionality and the reliability of mechanical parts that can be moved.

A pitfall with these graphs is that people who do not understand them may draw incorrect conclusions from them. We avoided this problem by offering several training sessions to explain how to read the graphs. The graphs are distributed only to people who have had this training.

1.7 Conclusion

The results for the project were clear. The test phases following the development phase (system test and acceptance test) ran a lot smoother than on previous projects. A comparable project with respect to size and complexity needed 15 system test cycles to reach the exit criteria. In the project described in this case, it took only 5 system test cycles. Fewer defects were found during the system test, but more important, the issues found were nonreliability issues. Reliability issues tend to be really nasty and can take quite some effort and lead time for analysis and solution. This effort and lead time were still needed, not at the end of the project but during the development phase, and we know that fixing defects earlier is cheaper. Other things had changed in this project, so not all of the benefits could be attributed to the reliability approach, but it is evident that the improved approach contributed considerably.

After several systems had been used by end customers for several weeks, no complaints from the customers were received about the reliability of the system, and not even a single failure had been reported. Previous releases could not make this claim. Also the product managers who had regular contact with customers were very positive about the stability of the system. The product managers became so enthusiastic that they wanted to add a

feature to the sales points of the system: improved reliability. After some discussion, it became clear that doing so was not a good idea because it would have a negative impact on older systems being used in the field and that could still be bought by customers. It was decided that customers should experience that stability for themselves.

GOOD POINT
Benefits are both quantitative (5 instead of 15 cycles) and qualitative (happier customers). Both are clear benefits from the automation approach.

Other projects also noticed the improvements and asked for reliability tests. More test setups were purchased, and dedicated test cases were implemented for other projects and products (including legacy products that needed maintenance). The decision to interface with the product at the hardware level made the conversion to other products fairly easy; although the software of these systems is different, the hardware interfaces are the same.

GOOD POINT
The best way to spread automation practices inside the company is for other groups to want what you have!

All issues identified with the test framework were indeed failures. No false negatives were identified. The fact that no failures were related to the test automation approach was mainly due to the use of hardware interfaces that interacted with the product. This results not only in high coverage (not only software is part of the test, but also hardware, electronics, and mechanics are used during the test execution) but also in a very low probe effect. A probe effect is an "unintended alteration in system behavior caused by measuring that system" (http://en.wikipedia.org/wiki/ Probe_effect). When too many false negatives are reported, the confidence in a test approach (whether automated or not) quickly declines.

As a result of the awareness in the organization about the importance of system reliability, a critical aspect of milestone reviews became the reliability. The reliability team became an important player during these milestone reviews during which it was decided whether or not milestones were passed.

The contents of the reliability test cases were chosen by best guess and in close cooperation with product managers. Future plans include an analysis of log files from the field to define real user behavior. This information can be incorporated into the test cases to improve the simulation of the end user.

For the success of this test automation approach, the following points were crucial:

- Choose the right project at the right time.
- Develop the test framework incrementally in order to show the added value as early as possible.
- Communicate the approach and the results by frequent presentations.
- Use clear, simple, and understandable reporting of the reliability growths (e.g., Crow-AMSAA plots).

What's nice to notice is that none of these points handles technical approaches or tool choices.

GOOD POINT
The tool is not the most important part of automation.

Biography

"After his Master's Degree in Computer Science (1998), Bryan Bakker has worked as Software Engineer on different technical systems. Since 2002, Bryan has specialized in testing of embedded software in multi-disciplinary environments; in these environments the software interfaces with other disciplines like mechanics, electronics and optics. He has worked on e.g. medical products, professional security systems, semi-industry products, and electron microscopy systems. In recent years Bryan focuses as a Test Architect on test automation, integration testing, reliability testing and design for testability. Bryan frequently speaks at (international) conferences."

Join the EuroSTAR Community...

Access the latest testing news! Our Community strives to provide test professionals with resources that prove beneficial to their day-to-day roles. These resources include catalogues of free on-demand webinars, ebooks, case studies, videos, presentations from past conferences and much more...

Follow us on Twitter @esconfs
Remember to use our hash tag #esconfs when tweeting about EuroSTAR 2013!

Become a fan of EuroSTAR on Facebook

Join our LinkedIn Group

Add us to your circles

Contribute to the Blog

Check out our free Webinar Archive

Download our latest eBook

Download our latest Case Study

www.eurostarconferences.com