System Verification by Automatic Testing

Asgeir Øvergaard

Kongsberg Maritime AS

Kongsberg, Norway

asgeir.overgaard@kongsberg.com

Gerrit Muller

Buskerud University College

Kongsberg, Norway

Copyright © 2012 by Asgeir Øvergaard. Permission granted to INCOSE to publish and use.

Abstract. This study elaborates the application of automated testing on an integrated software intensive system. We have been able to research the topic using a Kongsberg Maritime (KM) system, K-Master, as data provider. We have integrated an automated test framework into the test environment. The test team can use this framework to perform testing automatically.

The test team can use the effort released by automation to increase the amount of tests run. The test team can also choose to spend the released effort on other impending test activities. If we increase the amount of tests run, it can result in uncovering a higher number of problems in an earlier phase of the project.

This study shows that spending resources on automatic testing, resources, and save cost throughout the development project, while increasing coverage and frequency of testing. The result of this study will help justifying such decision.

Introduction

Kongsberg Maritime (KM) is a Norwegian company providing solutions for merchant marine, offshore, subsea, and other areas. After many years of stakeholder need analysis, KM has developed a concept for integrated vessel operation, the K-Master.

K-Master is a chair-based workstation designed to interface the vessel control systems used on the bridge of a vessel. This paper refers to the development state of the K-Master the fall 2011. We have not taken further development, improvements, and change in organizational setup into account in this paper. Numbers presented here can, for the same reason, deviate from the situation at a later point in time.

At this moment, the test team conducts a significant manual test effort prior to a K-Master release to be able to verify that the system complies with the given regulations. Manual testing is time and effort intensive, which limits the frequency of running the required tests. In the future, manual testing can become a limitation for new functionality that the company can release.

This paper will show the reader how System Verification by Automated Testing (SVAT) can help increasing the number of test executions and reduce the amount of effort needed to achieve sufficient coverage, as needed in the future.

The K-Master system typically consists of four computers running the K-Master Touch Control Panel (TCP) software. This is the Graphical User Interface (GUI) of the K-Master, and contains the most used features from the vessel control systems. These TCP computers interface several instances of the control system computers as depicted in Figure 1.

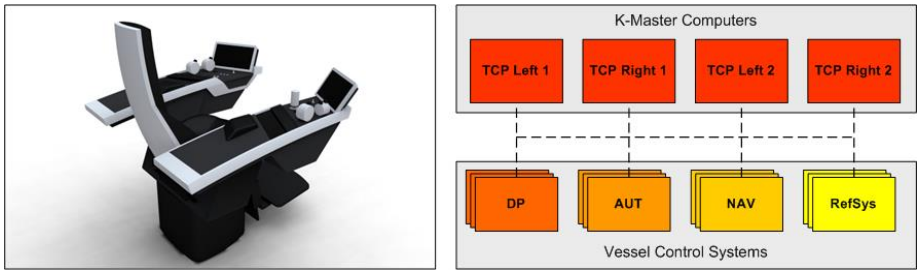

Figure 1 – System under test

An integrated system, like the K-Master, opens the way for new vessel control applications. If we foresee such an increase in integrated applications, the manual testing required maintaining the coverage, will increase accordingly. In this case, testing might quickly become a bottleneck in the development.

The major benefit of SVAT is the reduction of manual test time. The K-Master test team can spend this released labour on other test activities that can extend the test coverage. Another benefit of SVAT is the ability to run the same test several times on different environments and configurations.

This study shows that we are able to reduce the test time used for manual testing by more than 90% by automating the test procedures that are suitable for automation. We also expect that we, to some extent, can increase the test coverage solely by increasing the number of test runs.

KM has made some leading recommendations regarding tools that apply also to this project. The emphasis from the company was to use the Microsoft Team Foundation Server (TFS). TFS facilitates a more holistic view on the development process, with full traceability and linkage between project entities like requirements, user stories, test cases, tasks, bugs etc. Microsoft has designed a testing tool to create and store test entities in TFS. We selected this tool, the Microsoft Test Manager (MTM), as the top-level test tool for this project. Accompanied by Microsoft Visual Studio (VS), MTM has a quite good capability when it comes to automatic test execution. It can automatically execute a test case on a physical or virtual machine, and can collect test results, system performance, status, screenshots, video recordings etc. upon test failure.

Current Situation

Until now, there have been approximately 2-3 releases per year. The development team uses incremental builds during development with a complete build of the system (e.g. recompilation of all source code from scratch) every two weeks; this is the build cycle. The goal is to do a full test run for each build. Furthermore, it is desirable to increase the configuration coverage from one, to three tested configurations per build.

Because of the time intensive testing, the test team has together with the development team and the project management, decided to set up the test system with one “super set” configuration. This means that the test system contains all possible buttons, and function types. This is to be able to cover as many combinations of configurations as possible.

Current test focus is on functional testing. Testing of non-functional requirements takes place, but needs maturing. There is less research data available about the non-functional tests. Therefore, this paper focusses on functional testing.

K-Master is an ongoing development project. The development team is continuously adding new functionality to the system. Methods and techniques that can release time and increase efficiency are of great importance to the project when expanding the test coverage to meet the new functionality.

Problem Statement

Executing functional test cases at the integration level of the K-Master is a user interface intensive task. In practice, this means that the tester has to operate on the real system to be able to execute the test cases. This type of manual testing via the user interface is effort and time intensive. In average, it takes one or two testers at least 16 full working days, or about three weeks, to go through a full set of the functional tests once. Most of the content of the functional test specifications remains the same for long periods, which makes manual execution of these specifications a highly repetitive task.

Examples have shown us that we can expect a significant growth in number of applications contained in an integrated system. With the expected vast growing number of applications, the amount of test effort required to maintain coverage grows in parallel. This level of coverage might be difficult to cover in the future from manual testing alone.

To achieve better configuration coverage, there might be a desire to run tests on other configurations in addition to the "super set" configuration. We expect that the variation in configurations, due to combinatorial explosion, will increase even faster than the number of applications. Hence, the capability to test many configurations will have to increase significantly in the future.

Automated Testing in General

In a software intensive system, there is a multitude of test levels. Tests are less user interface intensive for the lower levels. For example, automating unit or component tests is a relatively straightforward task. The scope of a component is narrow compared to the system level scope, and a single component naturally has few interfaces to consider. At this level, a tests system can most often execute the tests automatically on a local machine. Testing how components, or collection of components integrate with sub-systems, is slightly more effort intensive to automate. Here, a test system can often execute the tests automatically on a local machine, but they are more user interface intensive than the component tests. The system level tests, like the functional test of the K-Master, require a higher effort to maintain than the lower level tests. If the system under test is a distributed system, like the K-Master, the test system most likely has to execute the tests distributed on separate computersIn general, automated testing can benefit a development project in a wide range of areas: less tester effort, testing 24/7, nightly test runs, possible accelerations, shorter elapsed time of testing, which allows for more frequent testing. More frequent testing again opens the way to regression testing, higher coverage, and testing of more configurations.

Literature Review

In a previous book, Software Test Automation (Graham 1999), the authors also emphasize the importance of thoroughly planned test automation. Here, they state that "At first glance, it seems easy to automate testing: just buy one of the popular test execution tools, record the manual tests, and play them back whenever you want to." This statement supports the need for this type of research in KM to find a proper approach to automated testing.

In Experiences of Automation (Graham 2012), Graham and Fewster summarize their key

findings when it comes to automatic testing as follows: “Management support is critical, but expectations must be realistic”, “Automation development requires the same discipline as software development”, and “Use a “translation table” for things that may change, so the automation can use a standard constant term”.

Andreas Lundgren has done research on the topic Graham and Fewster calls “translation table”. In his master paper, Lundgren researches the reasons for creating several “abstraction layers” in an automated test framework (Lundgren 2008).

Jung-Min Kim, Adam Porter and Gregg Rothermel, have published an article (Kim 2000) regarding regression test application frequency. In this article, they state that the test procedure execution schedule must reflect the planned delivery for each new build”.

Expected Benefits of Automated Testing

By automating the functional tests of the K-Master, we expect to see two major benefits. We expect to reduce the amount of time needed to go through a full set of functional test procedures once. Currently, it takes one or two testers three weeks or more per full run. The expected automated time for the same amount of tests is one night for the automatic tests, and one day for the remaining manual procedures. Considering a future increase in functionality, we need to keep the test cycle time significantly below the build cycle. This can be a critical factor in avoiding testing to become the bottleneck when the amount of new functionality grows vastly. The secondary argument for implementing automated testing is the preparation for other testing improvements. Such improvements can be regression testing, more efficient “non-functional” testing, increasing code coverage, increasing configuration coverage, duration testing, load tests and stress tests.

Research Methodology

This study uses a mix of “action research methodology” (O’Brien 1998), and “industry as laboratory” (Muller 2011) (Potts 1993). Industry as laboratory is a term used by Potts to describe research that evolves and validates methods and techniques in industrial practice to ensure that research and practice are well connected. Action research is a research form where the researchers actively participate in the field of application. O’Brien simply states that action research is “learning by doing”. For this project, the industrial case is the functional system level testing of the K-Master. We use action research while automating the functional tests.

By conducting this study, we will be able to measure the effort needed to create an automatic test framework for the K-Master. We will also be able to measure the effort needed to automate an informal test procedure.

In this study, we compare benefits and efforts of automated testing. We determine the current test cost by logging the time it takes to test the current specifications manually. When we run a complete test automatically, we can compare the time it takes to run an automatic test with the time it takes to run the same test manually. The data from the manual test executions reflect an average test execution time. We obtained this average from logging the team’s manual test execution effort over a one-year period.

An end-to-end comparison requires inclusion of the cost of automation. We measure the cost of creating the framework and automating the test specification. We also consider the cost of licensing, expert tool support, labor hours etc.

Project Approach

We ran this study as a short development project. However, we expected the development effort related to this study to be quite heavy. We customized the development process to eliminate project overhead. Our decision was to start with an implementation phase to learn early. Another reason for using this approach was to make use of a certain momentum for testing and test framework that existed in the company at the time, by showing results early.

Automatic Test Design

There are several tools existing in the market today for automated testing that might have solved our problem with test distribution. We briefly explored SmartBear Software's Test Complete at project initiation, and it seemed a possible candidate. As mentioned in the introduction, KM's global tool decisions led us to select MTM as automatic testing tool.

As previously mentioned, MTM has quite good capabilities when it comes to automatic test execution. However, it lacks the capability to run tests on distributed computers to support the high level of distribution in the K-Master system. We added this capability, by developing an intermediate client – host application, The Automatic Test Machine (ATM) see Figure 2. ATM lets MTM execute all the steps of a test case on one single computer as required. ATM triggers the required steps on the other computers. In addition to enabling distributed testing, ATM introduces the "translation table" or "abstraction layer" mentioned earlier. This has two clear benefits. Firstly, it enables the test designer to create test cases for automatic tests regardless of knowledge of the test code. Secondly, it makes the test cases more resistant to changes in the system under test because it enables MTM to call the same automatic test functions, even when the lower layer system functions have changed.

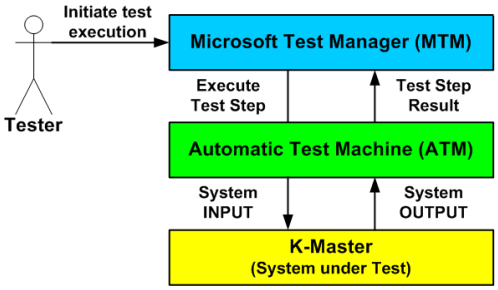

Figure 2 – Test Distribution, shown as three layers.

To give some more perspective into the remaining chapters, we present a simple data model (see Figure 3), describing how the informal test specifications relate to the different entities found in MTM. The left hand gray boxes represent the current Microsoft Word format test specifications. The blue boxes in the middle represent those specifications, converted to MTM entities.

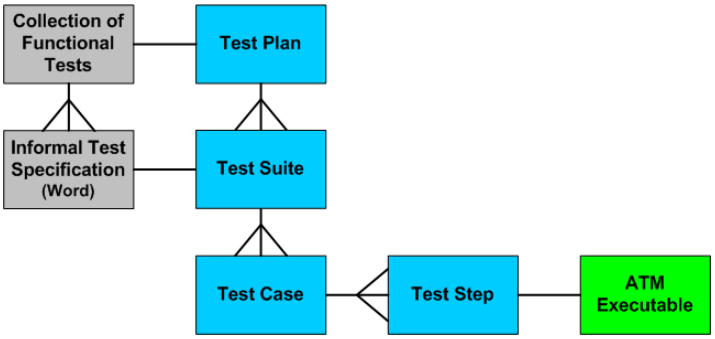

Figure 3 – Simple data structure model

Cost Model

We have created a simplified integral cost model to facilitate reasoning about the cost aspects (Figure 4). The model projects the required efforts in 2014, after two years of operation. The study provided us with measured data for parts of the model. We have estimated the other costs to get a better integral insight in the cost. This model depicts efforts required a few years into framework operation. We assume a yearly increase in functionality to be 50%. To be able to estimate the maintenance effort, we have explored some of the existing research on this topic. In the article Estimating Software Maintenance, Glinz (Glinz 2003) describes the ratio between development of new functionality and system maintenance. Glinz explains that the Putnam SLIM model uses a value of 1.5 for the ratio between maintenance and development (M/D). We use this ratio, 1.5, to estimate maintenance.

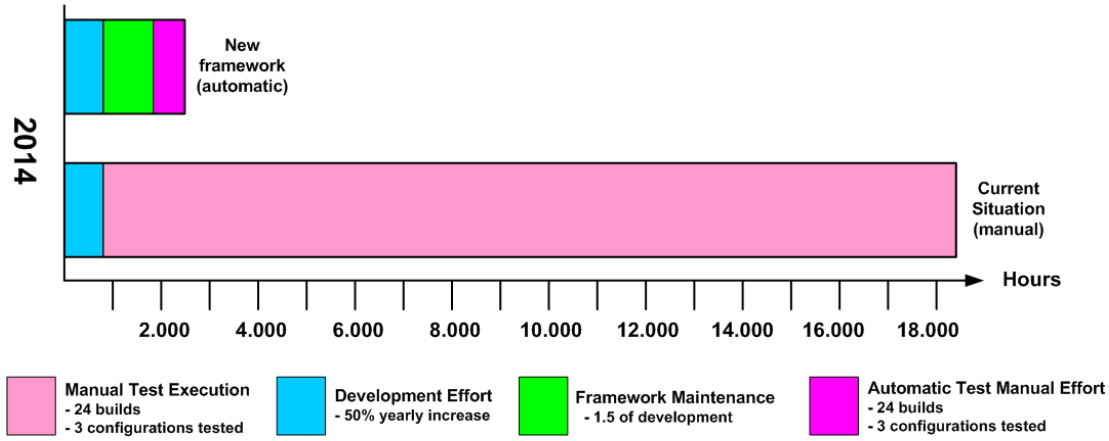

Figure 4 – Integral cost model – yearly efforts

We have calculated this model based on the goal of one full test cycle per build tested on three configurations. The operational effort related to the automatic test system is due to the work that we have to do to prepare for test, minor clean-up activities after execution, and result analysis after testing (see Figure 5). Most of the test result analysis will be equal for manual and automatic testing and disregarded in the calculations. The exception is the bitmap comparison. A bitmap comparison is a comparison of two digital images (or parts of them) to detect differences. From our measurements, we can say that every 10^{th} test step will leave an expected and actual image as test result. We estimate that comparing these two images manually has a maximum average time of 30 seconds. These data are estimates based on the experience with manual testing. For example, it might take 10 minutes to reboot all the vessel control systems, and 5 minutes to set a system back to initial after test.

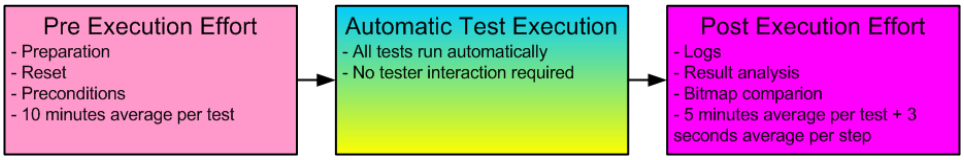

Figure 5 – Automatic test execution effort

In this study, there are several different cost factors involved. Some are investments, and some are recurring. In Figure 6 we have depicted the most significant non-recurring cost factors. The width of the pink boxes indicates the cost of the different factors.

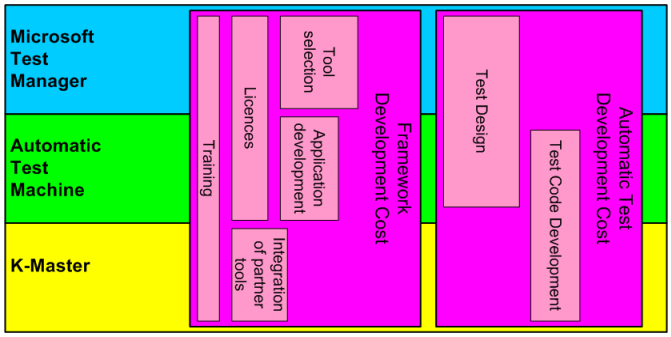

Figure 6 – Simple cost model for nonrecurring engineering costs; wider pink boxes has higher affiliated cost

Observations and Findings

In this study, we have developed an automatic test framework and automated the K-Master Aft Bridge Control Station (ABCS) Test. We have measured the effort needed to develop the framework and test. From this, we have calculated the effort needed to automate the remaining tests in the functional test collection. In Table 1, we present an overview of all the functional tests that are relevant for automation. We have highlighted the ABCS green.

Test specification	Test cases	Test steps
K-Pos Interface Test	18	926
K-Thrust Interface Test	9	1013
Aft Bridge Control Station Test	**53**	**426**
K-Chief Interface Test	13	159
K-Bridge Interface Test	30	114
system Test	7	80
IJS Control Panel Interface Test	11	48

Table 1 - Test specification overview; we automated the ABCS test

Cost

We have logged the hours and licence cost invested in the framework. The framework investment cost (see Figure 6 and Table 3) includes all hours affiliated with the framework development. It also includes the effort related to the tool study. The licence cost is low and we consider it not significant. Table 3 shows the cost pr. layer. We have based this on the accurate number of hours spent, and diverted in retrospect. The use of external resources has had a higher project cost pr. hour. We do not reflect this in our figures. We estimate that the current state of the framework is about 2/3 of the total effort. The remaining 1/3 estimated to cover remaining deployment and fine-tuning costs is included in the figures framework cost.

The test steps have the same form and composition in all the specifications. This enables us to make the calculations depicted in Table 2. We have based the calculations on the average time to automate one test step. We obtained this average from logging the time to automate the ABCS test specification.

Test development cost	
K-Thrust Interface Test	262 hours
K-Pos Interface Test	239 hours
Aft Bridge Control Station Test	**110 hours**
K-Chief Interface Test	41 hours
K-Bridge Interface Test	29 hours
System Test	21 hours
IJS Control Panel Interface Test	12 hours
Total development cost	**714 hours**

Table 2 -Estimated development cost per test, using the ABCS test for calibration.

Framework cost pr. layer	
Microsoft Test Manager	513 hours
Automatic Test Machine	342 hours
Test Execution Code Framework	120 hours
Total development cost	**975 hours**

Table 3 - Approximate framework cost pr. layer

Together with the data from Figure 5 we can make comparisons between manual and automatic test execution as depicted in Table 4.

Test specification	Test steps	Manual test execution	Automatic test execution	Post & Pre execution effort
K-Thrust Interface Test	1013	19 hours	238 minutes	104 minutes
K-Pos Interface Test	926	23 hours	217 minutes	96 minutes
Aft Bridge Control Station Test	**426**	**16 hours**	**100 minutes**	**52 minutes**
K-Chief Interface Test	159	15 hours	37 minutes	29 minutes
K-Bridge Interface Test	114	15 hours	27 minutes	25 minutes
System Test	80	13 hours	19 minutes	22 minutes
IJS Control Panel Interface Test	48	9 hours	11 minutes	19 minutes
Sum	**2766 steps**	**109 hours**	**11 hours**	**6 hours**

Table 4 – Manual test execution time, and predicted effort for remaining tests

Conclusions

In the integral cost model (Figure 4), we peek into the future and estimate 1200 hours of yearly maintenance cost. At the same point in time, we estimate the functionality and thereby the test effort to have increased by 50%. We also estimate a significant increase of integrated system functionality every year. Increase in system functionality will also soon require an increased coverage of tested configuration space.

We can present a simple function for return of investment given the current effort numbers from 2012. The total investment is the sum of the development costs for the framework and the test development cost for the current test specifications. We have measured the

development time per test step. We can calculate the yearly maintenance effort derived from the Putnam SLIM model when we know the development time for the number of new test steps per year. We know the current number of steps, and assume a 50% increase in tested functionally the first year. This leaves us with the yearly maintenance effort of 536 hours. We assume that the testers can do other tasks while the automatic tests are running. Therefore, we consider the automated test execution cost of 11 hours not relevant in this calculation, since the hardware needs to be present for testing anyway. Compared to manual testing, and using the desired test approach with 72 cycles every year, the framework will earn value after 18 test cycles, e.g. over 12 weeks of operation (see Figure 9).

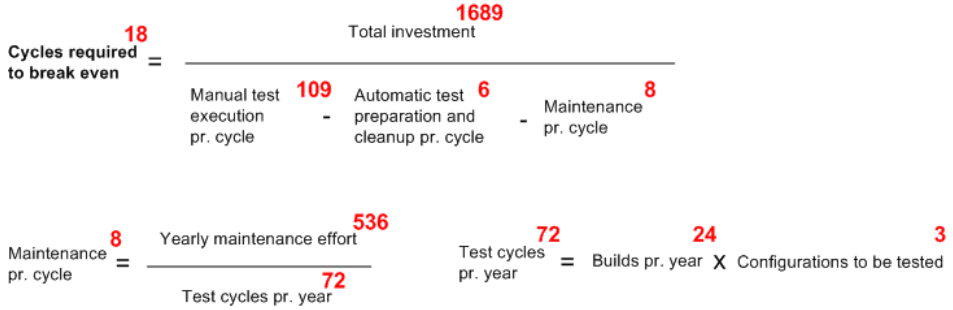

Figure 7 – Simplified function for return on investment, based on desired test frequency; all time numbers in hours

If we consider the desired test frequency and coverage, the numbers speak quite clearly in the favour of automatic testing. However, we have found an even stronger argument. Some years into the future, the system might have twice the current functionality. Then, we might have weekly builds, hence one test cycle a week. We might want to test five configurations instead of three. We depict this future example in Figure 8.

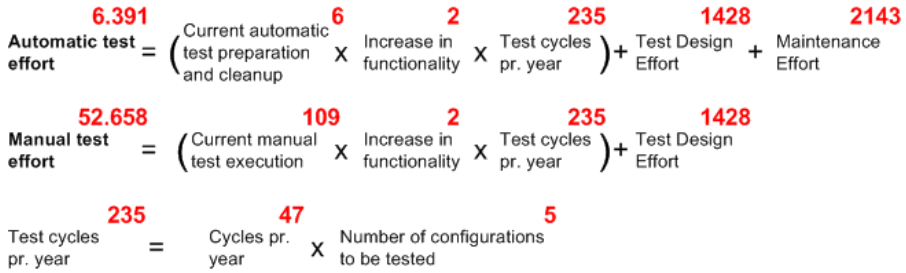

Figure 8 – Estimated future test effort; all time numbers in hours

In this example, 30 people would have to be dedicated 100% to manual testing of this system continuously, on a yearly basis. When the functionality increase rapidly, and it makes sense to increase test frequency and coverage accordingly, test automation quickly becomes the only real option.

Reflection

The goal of this study was to reduce the effort affiliated with functional testing of the K-Master. As mentioned, this level of functional testing is both time intensive, and repetitive. Repetitive tasks however, are good candidates for automation. We expected the outcome of this study to be an inviting figure for return on investment. However, what we have learned is that it is difficult to earn back the investment of the test system assuming a static environment with no change in amount of functionality to be tested or increase of test frequency or coverage.

When we include the likely increase of functionality, and look at the accompanied higher test

frequency, the numbers speak in the favor of automation. Considering the estimates we used in Figure 4, a more precarious question also caught our attention. Is it at all possible to maintain sufficient test coverage in the future, from manual testing alone?

We have created this framework to facilitate layering of the test functions. Graham, Fewster, and Lundgren all emphasize the importance of a layer in the framework to facilitate easier code change and maintenance. Our study has not proceeded long enough to measure actual maintenance efforts, or increase in test coverage. However, while developing the automated test, we have found that layering the framework, leaves the tests much more flexible to changes.

Lessons Learned

We started this six-month long study by jumping straight into development of the test framework. We did not use time to define requirements or design upfront. The reason for this was a certain momentum in the company concerning the need for testing and selection of test framework. By going straight into development/implementation, we could show results quickly. We used these results to gain sufficient resources for the framework development while the momentum was still high.

We gained quite good financial support from this approach. However, the implementation phase suffered from being effort intensive towards the end. In hindsight, we should have used a time-boxing approach some days every other week to model the system architecture. If we had done this from the beginning, the development and implementation could have been less effort intensive towards the end.

Another observation is that we had more focus on implementing the lower layers of the architecture first (ref. Figure 2). We did this, as a continuation of the feasibility study to prove feasibility before implementing the remaining layers. If we had also worked top-down, concurrently with bottom-up, we could probably have gained more overview of the priorities. In addition to the early modelling, this could have contributed to a less effort intensive study finalization.

We started this study because we had a feeling that automated testing would be beneficial to the K-Master. For the same reason we tried to show positive results regarding return on investment. We compared the automatic test system measurements with the current manual test measurements. These numbers came out quite positive until we did a slight increase in estimated maintenance time. This small increase sent the time to return on investment through the roof, and came as a nasty shock. To avoid this shock late in the study, we should have defined the needs and measuring points earlier.

From this study, we have learned that the real value of automatic testing only occurs when test frequency, and configuration coverage is increased significantly. We estimate a significant growth in both functionality, and combinations of configurations. Considering this growth, test automation might be the only way to achieve sufficient test coverage in the future.

Future research

The next important aspect to cover concerning automatic testing is the maintenance cost. The tests have to be maintainable, to adapt to changes in the system under test. The test team have to be able to add new, and change existing test cases, and equally remove obsolete ones. New functions in the system under test might require new functionalities from the test framework. If the system under test changes runtime environment, the test developers might have to maintain the test system to support the changes.

Guided by company policy we have chosen a specific tool. The total automation cost might have been different with another tool of choice. The total automation cost probably also relates to how tool functions match properties of the system under test.

We have stated that an abstraction layer and/or translation table leaves the test more flexible to changes. We should collect data throughout the life of the test system to be able to support this statement.

We have briefly touched upon expected increase in the number of configurations, which could cause a future coverage challenge. A large increase in frequency of test cycles and configuration runs could have a positive impact the test coverage and product quality knowledge. We have to do measurements to support this statement.

The reporting and communication cycle between the testers and the developers are currently quite long. With an automatic environment, it is possible to set up automatic bug reporting. Calculations show that automatic bug reporting can save considerable amounts of resources to the project.

The current automatic tests cover only the functional parts of the testing. Non-Functional tests are also good candidates for automation in the future.

References

Graham, D. & Fewster, M. 1999. "*Software Test Automation. Effective Use of Test Execution Tools*". Pearson.

Graham, D. & Fewster, M., 2012. "*Experiences of Automation: Case Studies of Software Test Automation*". Pearson.

Lundgren, A. 2008. "*Abstraction Levels of Automated Test Scripts*" Master Thesis, University of Lund (Lund, Sweden), Cinnober Financial Technology (Stockholm, Sweden). http://fileadmin.cs.lth.se/cs/Education/Examensarbete/Rapporter/2008/Rapport_2008-11.pdf. (accessed June 2012).

Pedemonte, P., Mahmud, J., Lau, T. 2012. "*Towards Automatic Functional Test Execution*." Research Report, IBM Research Division. http://domino.research.ibm.com/library/cyberdig.nsf/papers/2396E0ADAFE6EFBF8525799C005555C0. (accessed July 2012).

O`Brien, R. 1998 "*An Overview of the Methodological Approach of Action Research*." University of Toronto (Toronto, ON M5S 1A5 Canada). http://www.web.ca/robrien/papers/arfinal.html. (accessed July 2012).

Muller, G. 2011. "*Industry-as-Laboratory Applied in Practice: The Boderc Project.*" http://www.gaudisite.nl/IndustryAsLaboratoryAppliedPaper.pdf. (accessed July 2012).

Potts, C. 1993. "*Software-engineering research revisited.*" IEEE Software Vol. 10, No. 5: 19-28.

Kim, J., Porter, A., Rothermel, G. "*An Empirical Study of Regression Test Application Frequency.*" In Proceedings of the 22nd International Conference on Software Engineering. ACM New York (NY, USA). ©2000.

Glinz, M. 2003. "*Estimating Software Maintenance.*" Seminar on Software Cost Estimation WS 02/03. Requirements Engineering Research Group. University of Zürich. (Zürich, Switzerland). https://files.ifi.uzh.ch/rerg/arvo/courses/seminar_ws02/reports/Seminar_9.pdf. (accessed July 2012).

Biography

Asgeir Øvergaard

Asgeir Øvergaard, born in Norway, received his Master's degree in Systems Engineering from Buskerud University College in 2012. He received his Bachelor degree in Embedded Systems from Buskerud University College in 2009. From 2009, he has been working for the Norwegian company Kongsberg Maritime. He started his career in Kongsberg Maritime as a project engineer for drilling applications. In 2012 he received his Master degree in Systems Engineering from Buskerud University College. From 2012 he has been working as a test engineer for Kongsberg Maritime's Product and Development division.

Gerrit Muller.

Gerrit Muller, originally from the Netherlands, received his Master's degree in physics from the University of Amsterdam in 1979. He worked from 1980 until 1997 at Philips Medical Systems as a system architect, followed by two years at ASML as a manager of systems engineering, returning to Philips (Research) in 1999. Since 2003 he has worked as a senior research fellow at the Embedded Systems Institute in Eindhoven, focusing on developing system architecture methods and the education of new system architects, receiving his doctorate in 2004. In January 2008, he became a full professor of systems engineering at Buskerud University College in Kongsberg, Norway. He continues to work as a senior research fellow at the Embedded Systems Institute in Eindhoven in a part-time position.

All information (System Architecture articles, course material, curriculum vitae) can be found at: Gaudí systems architecting http://www.gaudisite.nl/