A Study on Design for Testability in Component-Based Embedded Software

Teemu Kanstrén
VTT, Kaitoväylä 1, 90571 Oulu, Finland
teemu.kanstren@vtt.fi

Abstract

Effective implementation of test automation requires taking testing into account in the system design. In short, this is called design for testability (DFT). In this paper a study on DFT in component-based embedded software is presented, based on the interviews and technical documentation from two large-scale companies in the European telecom industry. The way test automation is addressed and the different techniques applied to make this more effective at the architectural level are described. The differences and benefits of different approaches are discussed.

1. Introduction

Effective implementation of test automation requires taking testing into account in the system design. In short, this is called design for testability (DFT). This paper presents a study on DFT in two large-scale companies in the European telecommunication industry, both working on similar products, based on the same standards. The way software (SW) test automation is addressed and the different techniques applied to make this more effective at the architectural level are described. The differences and benefits of different approaches are discussed.

The testing discussed is different levels of black-box integration testing. At the lowest level, small components are composed together to larger components, and the internal messages between these components are considered. Properties such as internal structure at the level of code are not considered. At the highest level, all components are fully integrated as a complete system. In each case there is a separate test team dedicated to testing the components/system. For debugging problems, analysis is done at a more detailed level including the use of white-box techniques.

The tested systems are large-scale telecommunication systems. Each system consists of a number of hardware (HW) blades running SW with different functionality. Additionally, each system also interfaces with a set of other standardized systems, and both the internal correctness needs to be tested as well as the external interactions. Different parts of the SW are implemented in different programming languages, such as C/C++ and SDL. The system is divided to different sizes of components, and the testing described is done at the level of these components. While the system size in terms of lines of code was not always given, for example one system consists of about one million lines of production code, with a similar amount of code for the test environment.

As these are embedded systems, external HW measurement devices can also be used. The instrumentation mechanism choice is a trade-off in minimizing the effects of monitoring (HW) and providing more sophisticated views into the systems (SW). In this paper the focus is on the SW solutions.

2. Design for Testability

The term testability in SW testing can be considered from various viewpoints [1][2][5][7]. While some consider the architectural viewpoints [8][9][10], few describe techniques for more effective DFT at the architectural level [1][3]. However, this is commonly identified as an important goal in SW testing research [4].

In this paper two main viewpoints of DFT are considered from the architectural viewpoint: controllability and observability [5]. To test a component, we must be able to control its input, behavior and internal state. To see how this input has been processed, we must be able to observe the components output, behavior and internal states. Finally, the system control mechanisms and observed data must be combined to form meaningful test cases for a system.

The definition of how the interviewed see DFT has some variation, but the basic concepts are similar. These different viewpoints include the possibility to simulate different parts of the system for testing, to isolate the part that is being tested, to control system behavior with specialized test functionality and to access information on system behaviour. When test code in integrated to observe or control a part of the system,

978-0-7695-3302-5/08 $25.00 © 2008 IEEE
DOI 10.1109/SERA.2008.11

IEEE computer society

the location of the test code is called a test point. When testing is run in a desktop simulation environment outside the target HW, this is called host testing.

The presentation and discussion of the test automation and DFT concepts in this paper are described according to the following main concepts: test implementation, control of messaging, simulation strategies and implementation of functionality to support testing. How the different companies address each of these concepts is described and discussed.

3. Research Methodology

The interviewed companies were chosen based on their mutual interest. As they were starting closer collaboration, both had interests to combine strengths of the two companies. The interviews followed a semi-structured format, where questions were grouped into themes. The goal was to allow the interviewed to freely express what they felt were important concepts, while keeping the focus on the matter at hand. The following is a list of the main questions addressed:

Theme	Main Question(s)
Test Automation	• How do you implement test automation? • What solutions do you use to support implementation of test automation?
Observability	• How do you collect information from your system? • How do you address constraints such as real-time requirements?
Controlability	• How do you support controlling system states, behavior and partitioning? • How do you focus on problem analysis?

In both companies, a number of specialists in test automation were interviewed. Each company was asked to select a number of specialists with good knowledge on the interview topics. Some technical documentation was also received, describing the test automation systems. Once the information had been collected, results were checked with the interviewed people.

4. Test Implementation

The basic test implementation in both companies is based on verifying the correctness of message sequences and checking of message parameters. Although the systems have hard real-time requirements, they are considered only at the system testing level and not on the integration testing level. While some load and stress testing is performed during integration testing, it mostly done in system level testing, as in these cases the complete system in composed and problems in high level integration and interoperability can be seen. For testing timing related functionality in integration testing, specific test cases are used that manipulate the timers used in the system. For example, they can be set to expire immediately to test timer related fault handling. The amount of generated test data can also be a problem, as the test bus can become exhausted, causing failures when buffers become full. This requires special considerations on how and where test data is processed.

4.1 Integration Testing

Both companies use basic test scripts to verify the message sequences during integration testing. Verification of message sequences is based on the external interfaces of components, which is seen to help shield the test cases from minor changes in system implementation. As these messages are captured at the component level, several thousand lines of code can be executed between messages. Typically, the information on internal messages is also available, but these are only used when problems are found and need to be analyzed. Failing test cases are executed with more detailed logging to focus on the cause of failure, including internal messages passed and their parameter values. Most difficult problems to debug are seen to be problems that come up during long uptime, slowly consuming resources such as memory or CPU load.

Company 1 (C1) has used a traditional approach of developing test components (stubs) as needed in isolation. Company 2 (C2) has taken a different approach where, during development and integration testing, two versions of the system specification are implemented. One is the actual product and one is the test system. The test system provides simulated versions of all the components in the production system, and is developed using similar development and quality assurance processes as the production system. As two versions of the specification are implemented, they provide validation for each other and the understanding of requirements. In case of a failing test case it is necessary to consider which implementation is wrong, the test system or the production system. As same development processes are used for both systems, the same metrics can also be collected and compared for both. This can provide interesting insight into the effectiveness of test automation development. For good test system implementation it is seen that a ratio of 1 to 1 is good and a ratio of 1 production system error to 2-3 test system errors is more common.

4.2 System Testing

Similar to integration testing, C1 has relied on scripting of input and output also at system test level. C2 used to do the same but has moved to using higher-level abstractions due to difficulties in maintaining the test suites. In this case, the message sequences are encapsulated inside test building blocks, which describe high-level functionality of a system. These are further grouped into test cases, which are grouped into test suites. When system functionality is changed, updating test cases requires changing only some of the building blocks and not all test cases. As the blocks can be further reused over a product family, this has been found to lead to lower maintenance costs. Also, as test cases can be built from higher level abstractions (building blocks), it is easier for a system tester to write test cases without detailed knowledge of system internals. The goal is then similar to approaches such as model based testing [12], with the aim of using a higher level model abstraction to build test cases. In this regard, the implementation of the C2 system test environment takes more effort, but has smaller maintenance effort as changes are contained in the shared test components.

As described earlier, most of performance and load testing is left for the system testing phase. However, while C2 has put more effort into creating an advanced functional test environment for system testing, they have used only basic timing measurements of external interfaces also at the system testing level. For this type of testing, C1 has put more effort on advanced techniques for supporting analysis of resource usage and performance. This is based on analysis of detailed internal information, starting from generic properties and progressing to more detailed analysis based on the findings from the generic properties. At this level, the parameters include task switches, data on resource usages and use of OS services. As these can be monitored from outside application code (at the system level), they do not require as large effort to implement. When more focused information about an identified problem area is needed, more specific tracing is implemented. This data includes properties such as component inputs and outputs, system id values and data streams. Further, analysis tools have been developed to analyze this information using multivariate analysis techniques. This has been found very useful in system optimization.

5. Control of Messaging

Effective implementation of test automation requires being able to control the system execution and observe the results. In this regard, it must be possible to create different compositions of the system and its components, including the use of test components (stubs) as replacements for actual components. In component-based SW, a common means to compose components together into larger systems is through middleware [11]. Both C1 and C2 have taken a similar approach to make this possible, by controlling the messaging between the components, through their middleware. This section reviews these approaches.

5.1 Company 1

C1 uses a commercial off the shelf (COTS) third-party operating system (OS), targeted especially at embedded systems, in their products. For enabling creation of system test compositions and the use of simulation, rerouting of the system communication mechanism is used. In system execution, the execution of components is mapped to the OS processes, which run them as tasks. All communication between components is done through the system messaging interface, which is an inter-process communication (IPC) interface. The communication is handled by a system internal routing component that delivers the messages to the correct processes and components. As the same messaging interface is used over all the components and is based on a standard protocol, it is easier to build generic and reusable test services for this interface.

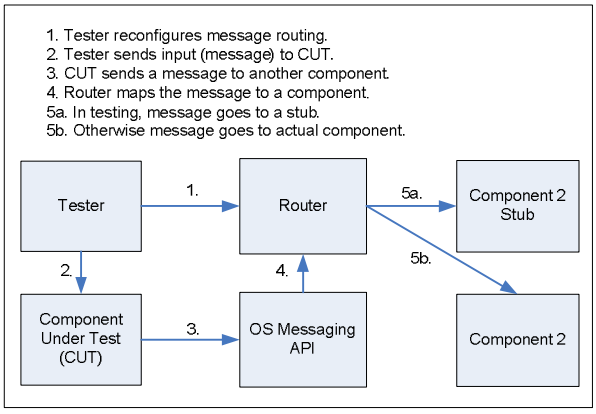

Figure 1. C1 message flow control and testing.

The process of using this mechanism to attach simulated components to the test target is shown in Figure 1. The system router component contains a routing table for passing the internal system messages to different components. By modifying this routing table, the messages can be passed to test components instead of production components. In addition to basic test stubs, which provide messaging functionality, this has also been used to implement more complex functionality to gain control over deeply embedded functionality. This is discussed in more detail in section 7.

5.2 Company 2

As a basis in their products, C2 uses a generic open source software (OSS) OS, which can be used equally well in both a host test environment and on embedded target HW. The enabler for using the test environment is the custom middleware on top of which the whole system runs. This middleware contains a communication translator component, which handles the addressing of component communication. When the system components are composed together to form the system, each SW component publishes over the communication translator their communication id values and subscribes to other components using their id values. The counterparts are mapped together by the communication translator. As soon as the required components are available, they are subscribed and connected together, and messages can be passed.

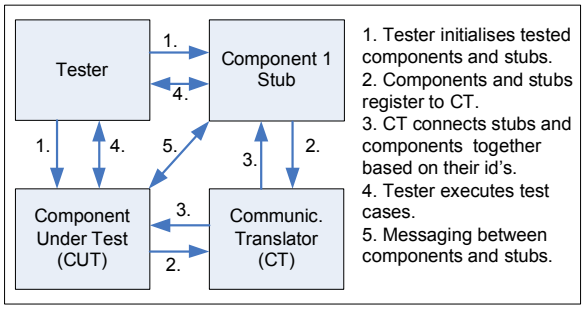

Figure 2. C2 message flow control and testing.

Using this functionality, the components are wired together to provide the test composition needed to effectively isolate and test the production components as illustrated in Figure 2. To create system compositions for testing, test components are published with the connection id of the matching production components. Thus, the system under test (SUT) sees this as a normal operating environment. To enable the creation of different system compositions for testing, the database describing the system components can also be controlled. The system configuration is set through the database, and the required components are then implemented in the test environment. These test components can then, depending on the configuration, provide test functionality such as answering with certain messages and verifying the contents and order of received messages. High reuse factor has been achieved in using parts of the test components over different test cases.

5.3 Discussion

As both C1 and C2 both work in the same domain and develop products based on the same specifications, they have many commonalities in their testing. The systems are based on a set of standard interfaces and system components that define the external structure and interfaces of the system. At the level of integration and system testing, both have mainly focused their testing on the messages between the components. Although there are differences in test implementation and system architecture, there are also similarities in the approaches taken to address the testability requirements in the internal design of the systems.

To control the system execution for testing, both take a similar approach of controlling the routing of control- and data-flow via the system messages. C1 uses the configurability of their OS message routing to enable this. C2 uses the similar functionality of their custom middleware. Both use these to enable stubbing of interfaces in their test environment, which in turn enables customized test configurations of the SUT. Both of these approaches have their own advantages. Using the services provided by the OS, there is less need to develop a custom, self-made middleware. However, using a custom-made middleware that can be used on both host and target system provides more flexibility and better possibilities for host testing, as described in section 6.

Overall, it can be concluded that for the type of testing described here, it is necessary to be able to control the messaging of the system for effective test implementation. The possibilities for this are constrained by the used SW platform.

6. Simulation Strategies

In all SW testing it is important to be able to simulate parts of the SUT for effective test implementation. As described in section 5, it must be possible to use simulated versions of components (test stubs) providing test functionality as a replacement for actual production components. However, especially in the context of embedded systems, it is also important in whole to be able to run tests in a simulated host test environment, without the need to always run on target HW [6]. This section describes the different approaches and related constraints with the interviewed companies in this regard.

6.1 Company 1

As described in section 5.1, C1 uses a COTS OS, targeted especially at embedded systems. From the simulation viewpoint, this is problematic as the OS is tied to the target HW, and cannot be used as such in a host test environment. While it does provide a separate simulation environment that enables some testing, this

simulation environment does not equal the use of actual full OS. Although C1 sees it important to effectively simulate different parts for testing, they have done only very limited host testing. Instead, they have aimed at running as much as possible of testing on target HW, where the tested component(s) and the simulated stubs are all loaded on to actual target HW. They see this to have the benefit of getting more accurate timing as well as fully matching any parameters of the target system that may affect the functionality.

In C1, the communication interface uses a common messaging protocol, built on top of standard network protocols, for all parts of the system. This is seen as an especially important enabler for building effective test automation systems, as it enables building reusable test components and services. With this type of a communication mechanism, C1 has found it possible to build multipurpose test components that can be used in testing of different parts of the system.

6.2 Company 2

As described in section 5.2, C2 uses a generic OSS OS, which can be used equally well in both a host environment and on embedded target HW. As both the target HW and the host simulation environment use the same full OS, they are a very close match to each other. Also, the same SW can be deployed both on target and in host environment without changes or visibility to the deployed SW. Only the HW interface part needs to be specific to the target system, and is typically only needed in later phases of integration. It is also possible to run parts of the system, such as HW specific startup code or databases on a different blade on a target HW, while running the rest of the system in a host environment. While the goal with testing at this level is to test as much as possible in the host environment, some of the testing such as redundancy and failover needs to be done with actual target HW, where multiple HW blades are available.

6.3 Discussion

The main difference with regards to simulation environments in C1 and C2 is in the type of OS used and the environment in which most of the testing is performed. While this is partly defined by the possibilities of the used OS, it also reflects the deeper views of the two organizations. Running as much as possible of integration testing in a host test environment is a view shared by all interviewed people in C2, whereas the opposite view is shared by the people in C1. The main reason mentioned for C1 to prefer running as much of testing as possible on target is that in this case all parameters of the actual environment are correct according to the target HW. While the optimal choice depends on many properties, both C1 and C2 recognize that testing on target HW is expensive as it requires having a large number of custom made target systems and specialized test equipment available for testing. A host testing environment also enables better control over the test environment, and the target HW issues as a whole can best be addressed in the system integration test phase, when the whole system is composed.

For C2, an effective host simulation environment is available by running their middleware and SW on top of it in a (desktop) host test environment in the same OS. As the middleware is in-house and the OS is OSS, the system supports a wide range of customization possibilities. The C1 OS based solution is tightly coupled with the internals of a third party COTS OS, which makes effective customization more difficult. Their OS is also specifically for embedded systems, and cannot be used in a host test environment as such. Instead, a specialized simulation environment is needed, which is more complex to match to target than running the actual OS. As C1 has not made much effort to use host testing, it is unclear how well this could be done.

7. Test Functionality

Enabling effective test implementation typically requires certain properties and functionality from the SUT. How SW components can support the testing process has received a lot of attention in the recent years [11]. However, these techniques are mostly considered from the viewpoint of third-party black-box components, where support is built into a single component. At a higher level, this section reviews how C1 and C2 have addressed supporting testing at a level where several components are integrated.

While the techniques described earlier for controlling messaging and using simulation are basic enablers, also more advanced functionality is needed. To effectively build automated test cases, it must be possible to observe and control different parts of the system, which when integrated can be difficult to access. This is especially true when there is a need to support testing and diagnosis of deployed products, as is the case with both C1 and C2. Finally, even when this observation and control support exists, data still needs to be made available. In embedded systems this provides its own challenges. As opposed to SW running in a desktop environment, there is often no direct visibility to the SW running on target HW, and even reading printouts from application code requires custom solutions, such as use of network protocols as used by C1 and C2. To support analysis of long-running systems, run-

time test support is also needed. In this section the functionality to support test implementation is reviewed.

7.1 Company 1

When data from an internal part of the system needs to be collected in C1, the data flow can be changed. One applied solution is using the functionality described in section 5.1; message routing is configured to go through an extra test component that collects data. A basic setting for this is illustrated in Figure 3. Similarly, by connecting various component input and output ports together, data flow in the system can be looped to come back to the sender. Using this technique requires special consideration, as the data will likely be of a different format than expected in the "abused" output path.

For direct access to deeply embedded features in C1, a technique called embedded tests is used. These are test components that are integrated into the system to provide specific test functionality. To enable this, the message routing techniques described in section 5.1 are once again used, as illustrated in Figure 4. The functionality of these components includes feeding test input data, doing comparison and transferring test result data out from internal interfaces. The embedded tests also help address other constraints such as real-time requirements and limited communication buses, by providing fast data input and processing near the interface, limiting the need for external communication. These tests are typically integrated ad-hoc where needed and not left in the system, as also the need where they are used varies. Other techniques to address performance constraints include running test data processing and transfer tasks when system load is low, by using low task priorities.

In C1 the viewpoint has been that it is difficult to provide generic observation points between different components of a system. This is both because of differences in actual implementations of components and due to difficulty to get management support for adding extra test code into the system. The priority of DFT SW development and resourcing is always put much lower than that of production SW. As most tracing then needs to be implemented on ad-hoc basis, common functionality that is used in tracing a system has been implemented in a test point library. This functionality can be integrated in different parts of the system through the library. Separate integration is needed for each test point in the system, but from thereon the library provides the functionality. The functionality of this test point library includes different functionality needed in testing, such as storing test data, moving it out of the system, doing comparisons, reporting results and monitoring system resources.

The goal with the test point library implementation is to provide functionality that makes it possible to implement embedded test and other test functionality into the system with minimal effort. This includes implementation of such functionality over different parts of a system, but also across different products in a product family. To this end, the library has been made HW platform independent. The library functionality is built on top of a HW abstraction layer (HW API), and porting the library to a new system typically needs porting the HW API code between the SW and HW, while the HW API interface stays the same.

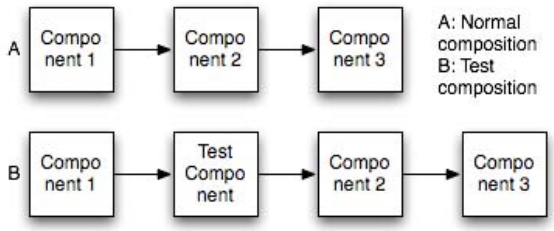

Figure 3. Embedded test component.

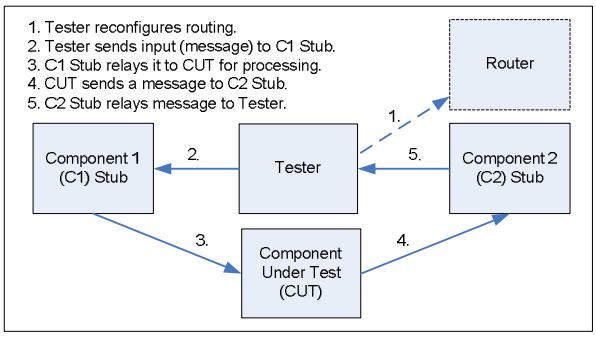

Figure 4. Embedded test.

7.2 Company 2

In C2, features to support testing and more detailed analysis of system behavior have been included in the system as first-class features. This includes both functionality to observe system behavior and built-in test functionality. While their middleware would enable using techniques such as embedded tests described in section 7.1, these have not been used extensively due to wanting to not change the SUT during testing. Instead, the goal has been to always test the system as it will be finally deployed, with no test features added only for testing and removed after testing is finished.

For systematic tracing support, development guidelines define what can and needs to be traced at different levels. Tracing is divided to different levels, and different parts of the system are defined to belong to a certain trace module. Each module contains what is

called a trace notebook to store trace data. At the finest trace levels, developers are free to put almost any trace they wish, as these levels will be compiled away when actual SW releases are made. At the higher levels, which are included in deployed products and where less trace can be produced, guidelines include always putting a certain trace level on input and output of external and internal interfaces.

The goal with the trace functionality is to always have some systematic information to work with, even if it is only possible to gather very limited amount of data due to performance constraints of deployed systems. Quality assurance policies are applied and inspections are done with experiences developers to see that all required important trace points are included in the system. In addition to storing information about system execution, all error symptoms are also logged in the notebooks. Techniques such as shared memory and stored to disk storage are used to enable access to data also in case of application crash.

The trace functionality can be configured statically through a configuration file or dynamically through a configuration interface during run-time. Trace functionality can also be set to activate or configure based on various system event triggers. Some of the basic functionality of tracing is also supported similar to the test library of C1. Mainly this is related to inserting trace statements, moving the data out of the system, data post-processing and analysis. Monitoring of system resource usage such as CPU and memory is supported and stored by the trace system.

As the more abstract levels of trace are always included in the system and can be configured dynamically during run-time, they can also be used during long testing sessions and with deployed systems in the field. Additionally, this is supported by built-in test functionality called audit tests. These tests run inside the system, when the system load is low and there are resources available to run the tests. They check the system consistency for properties such as resource leaks. Without this, it is difficult to see what has happened in the system if the system goes down after weeks of running and there is no sign of how the symptoms developed over time.

Another application of audit type tests has been in testing for errors in redundant HW blades. In this case, the system contains two or more blades that are redundant and provide fail-over functionality. During system runtime, these can be tested with known input and output data to see if they provide correct functionality. If the results are different, an error notification can be raised and the failing blade can be replaced with a correct one while the system is running.

7.3 Discussion

The approaches with the two companies with regards to including test functionality in a system have been the opposite. In C1, the viewpoint has been that no functionality to support testing are to be included in actual products, as these are not something sold to customers. As DFT support in C1 has in general been a low priority, their solutions to support this have been limited. On the other hand, the goal in C2 has been that the SUT remains unchanged during testing. From their viewpoint, it is seen that unless test functionality is a part of the actual product, the tests do not test the actual system, since the test functionality is removed in the end. Thus their supporting DFT features have been made first-class features for the system.

Due to not being able to include test functionality as a part of the actual product, C1 has developed their test point library to support testing and debugging with ad-hoc solutions. This is seen especially problematic in diagnosing deployed products and in long testing sessions. Integrating separate test support functionality requires loading a new SW version and resetting the system, which also makes the fault state disappear and impossible to debug. In general, supporting field-testing of deployed products is identified as an important area of improvement in C1. Currently, only a number of basic properties can be observed, such as number of resets inside a HW block.

As stated earlier, C2 has taken the opposite approach to fully include all test functionality in the product. This along with their audit tests provides C2 with much better support for testing and diagnosing problems of deployed products and long testing sessions. Also, as systematic tracing is included in all parts of the C2 SUT, they also share a common data format. This enables use of same tools for all parts. In C1, this has caused some problems, due to fragmentation and incompatibilities caused by the different tools and data formats taken by different organizational units.

One of the reasons for difficulties in getting support included for testing purposes into the system in C1 is often cited as people not wanting to add any (test) features that are not sold to customer, into the product. In this regard, testing has not been valued high enough to be given systematic support, but has rather been viewed as something extra to be put up with. Thus lack of management support and not valuing testing high in the company culture are some of the main reasons. On the other hand, it is not clear if better support would have been received if someone had suggested similar solutions, and argued with extended support for field testing and other cost savings. Field test support in C1 is one of the identified areas needing improvement.

8. Conclusions

This paper discussed the DFT solutions to support test automation from two companies in the European telecommunications domain, working on similar large-scale component-based embedded systems. Their techniques to support effective test automation were discussed. While the approaches taken have a lot in common, there are also a number of differences. While it is not possible to generalize from this data to all SW development and testing, a number of observations can be made that provide interesting insight into these topics. These are summed in the following:

- Testability needs to be taken into account early in the design, in the SW platform. Control over system messaging provides support for control over system execution paths and efficient implementation of test environments and configurations. A common communication protocol further provides support for implementing reusable test components.
- Especially in the case of embedded systems, a good host test environment enables efficient SW testing. When this environment matches the target system as much as possible, efficient host testing is possible. One enabler for this is using an OS that is supported on both the target HW and in a (simulated desktop) host-testing environment.
- Including supporting test functionality in the system as first-class features allows for more effective analysis of the system, including analysis of long running tests and deployed systems, and enables efficient field-testing. Effectively implementing this requires possibilities for dynamic configuration of test functionality during system run-time.
- In addition to systematic test support functionality, ad-hoc requirements are likely to arise in different points of testing and analysis lifecycle, and in this case it is useful to have support for this functionality provided in the form of a reusable library.
- Abstracting test cases from the implementation minimizes the effects of internal system changes to the test cases. This mostly applies at the system testing level, as in earlier testing phases it is often necessary to observe more detailed properties of the system.
- To make it possible to get the desired test support functionality included into the system design and to create advanced tools, management support is crucial. This requires valuing testing and system analysis high in the company culture.

9. References

[1] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, 2nd edition, Addision-Wesley, 2003.

[2] B. Baudry, Y.L. Traon, Measuring Design Testability of a UML class diagram., Information & Software Technology, vol. 47, no. 13, pp. 859-879, 2005.

[3] S. Berner., R. Weber, R.K. Keller, Observations and Lessons Learned from Automated Testing, Proc. 27th Int'l. Conf. on Software Eng. (ICSE 2005), p. 571-579, 2005.

[4] A. Bertolino, Software Testing Research: Achievements, Challenges, Dreams, Proc. Future of Software Engineering (FOSE2007), pp. 85-103, 2007.

[5] R.V. Binder, Design for Testability in Object-Oriented Systems, Communications of the ACM, vol. 37, no. 9, pp. 87-101, September 1994.

[6] B. Broekman, E. Notenboom, Testing Embedded Software, Addison Wesley, 2002.

[7] M. Bruntik, A. Deursen, An Empirical study into class testability, Journal of Systems and Software, vol. 79, no. 9, September 2006.

[8] S. Jungmayr, Identifying Test-Critical Dependencies, Proc. IEEE Int'l. Conf. on Software Maintenance, Montréal, Canada, 2002.

[9] R. Kolb, D. Muthig, Making Testing Product Lines More Efficient by Improving the Testability of Product Line Architectures, Proc. of the ISSTA 2006 workshop on Role of Software Architecture for Testing and Analysis (ROSETEA2006), pp. 22-27, 2006, Portland, Maine.

[10] B. Pettichord, Design for Testability, Proc. Pacific Northwest Software Quality Conference (PNSQC2002), Oct. 2002.

[11] M.J. Rehman, F. Jabeen, A. Bertolino, A. Polini, Testing Software Components for Integration: A Survey of Issues and Techniques, Software Testing, Verification and Reliability, vol. 17, 2007, pp. 95-133.

[12] M. Utting, B. Legeard, Practical Model-Based Testing: A Tools Approach, Morgan Kaufmann, 2006.