Verification and Validation of Software Components and Component Based Software Systems

Christina Wallin

Industrial Information Technology
Software Engineering Processes
ABB Corporate Research
christina.wallin@mdh.se

Abstract: One premise with component based software technology is that the software component consumers, that is the software system builders, can decrease their effort needed for, among other things, verification of their component based software systems compared to traditional custom developed systems. But to reach this goal the component producers instead has to ensure a high, documented and trusted quality of their commercial components; else the effect will be the opposite. It should be easier to assemble pre-made software component into systems than to develop everything from scratch, but if a used software component is of poor quality the system verification task can be very difficult, time consuming and thereby expensive.

The traditional techniques for verification and validation are still essential to ensure software quality, especially for the software component producers. Detailed specifications of the software, and thorough verifications against these specifications are ways to certify the component functionality and quality in a way that can be trusted by the component consumers. Also a defined software development process and appropriate development methods and tools ensure a high software quality. For the component consumers software verification is not like traditional verification. Software faults cannot be found and corrected by traditional debugging techniques, as most of the source code is not available. Instead a more research like technique has to be used to locate faults and workarounds has to be implemented to avoid them.

1 Introduction

The emerging technology of Component Based Software Development put new demands on how to ensure the needed functionality and quality of the software. In traditional, or custom developed software, verification and validation could be done in close cooperation with the customer to meet the specific requirements from the customer [4]. When developing software components, or component based software systems, verification and validation becomes a bit more complicated. Components developed for reuse, and especially components developed for the open market, has to be more thoroughly specified and verified then most custom software for many reasons. Others than those that have developed them will use them. They will be used in many different situations and configurations, many of them not foreseen at the time of development. They will be not be delivered together with its source code, but as black boxes [7].

On the other hand, software systems developed by means of components should require less verification than custom software systems; at least this is one of the premises with component based software technology. But if the quality of the used components is not enough, identifying defects in a component-based system can be both difficult and expensive [2]. Instead of
systematically searching through the software code for faults, a more research like approach has to be used to locate defects [10]. Instead of correcting a defect, a work around has to be developed to avoid it.

There are two ways to ensure the quality of a software component or software system. One way is to identify (and then remove) defects introduced; this is done by means of different verification and validation activities. Another way is to prevent the defect introduction; this is done by means of an appropriate development process. But ensuring the functionality and quality is not enough, for commercial components functionality and quality also has to be certified, that is proofed in a trusted way.

Section 2 describes shortly some traditional and some new verification and validation techniques, section 3 discusses some aspects of defect prevention, section 4 describes shortly some issues around software certification and section 5 discusses about which verification and validation techniques that are useful in the different component based software development processes.

2 Verification & Validation Techniques

The purpose of verification is to assure that the software components or software systems meet their specified requirements [3]. Specified requirements are not only found in requirements specification documents but also in functional specifications, architecture and design models, test cases and so on. Specified requirements can also be found in regulations, standards, instructions, and guidelines and similar that puts constraints on the software. The purpose of validation is to demonstrate that a software component or system fulfills its intended use when placed in its intended environment [3]. There are two main techniques, inspections and tests, which may be used for software verification. Tests are also used for software validation [4].

The purpose with both inspections and tests is to identify defects in the software. Inspections, or static verification, are used to check and analyze representations of the software component or system such as specifications, models and source code. Testing, or dynamic verification and validation, involves executing and examining an implementation of the software component or system [4]. Verification and validation should be performed continuously during the development of a software component or system.

Specifications of the software are crucial to support verification and validation [4] [6]. Specifications define the correct structure and behavior of the software so that incorrectness can be identified. Incorrectness is software faults (also called defects or bugs) and can cause software failures. Software can also fail due to environmental constraints not caused by software faults.

2.1 Inspections

Systematic software testing requires a large number of tests to be developed, executed and examined and this means that testing is both time-consuming and expensive. Inspections have proved to be an inexpensive and effective technique to identify defects for two reasons [4]. One reason is that many defects can be found during one inspection session while a test normally only identifies one or a few defect at the time. The other reason is that inspections reuse domain and implementation knowledge among the reviewers in a more efficient way than tests.
Inspections can be done on different levels of formality and can either be done manually or by means of some analysis tools.

2.1.1 Manual Inspections Techniques

Audits and assessments are formal inspection techniques and are based on interviews and reviews of work products such as documents, models and source code. Stakeholders or experts or quality responsible or similar external to the software development responsible normally does audits and assessments and the also decide what and how to inspect. Audits and assessments findings are documented and corrections based on these findings should be verified again.

Formal reviews are normally done internally by the software development team itself. Relevant work products are sent out the review team members and each reviewer report his/hers findings. The final result is summoned on a review meeting. Findings are documented and corrections based on these findings should be reported [4].

Peer reviews are more informal inspections performed between colleagues. It is an exchange of services. Findings are normally not documented and corrections are not reported. The most extreme usage of peer reviews can be found in eXtreme Programming (XP) [9], that describes a Pair Programming strategy where two colleagues, or more, work in pair or groups cooperating in analysis, design, coding and test all the time.

2.1.2 Tool Based Inspections

The most common software inspection tool is the source code compiler. A lot of software fault are identified in a very early stage of development by the compiler. Static source code analyzers are software tools that scan the source code and detect possible faults and anomalies by analyzing the control flow, data usage, interfaces, and also information flow and execution paths if applicable [4].

Static program analysis of executable software is done with tools that can analyze software contents, structure, and logic by examining the executable file image. Binary viewer and editor tools show the binary code of the software and the corresponding ASCII representation and allows minor modifications (patches) of the code. Dis-assemblers can create pseudo code representation of the binary code and de-compilers can even reconstruct the original source code [10].

2.2 Tests

Testing of software components (and component based software systems) is possibly the single most demanding aspect of component technology [1]. When a software component is developed all future usage of it cannot be known and thereby not tested, and when the software component is integrated into a system it should already be tested and ready for use. The possibility to reuse pre-made and tested components is the core of the component technology.

Software testing is the activity of executing software to determine whether it matches its specifications and executes in its intended environment [6]. The fact that the software is
executed distinguishes test from code inspections in which the un-compiled source code is read and analyzed. Testing requires a running executable.

Testing can be grouped depending of the focus and level of details of the test. White box tests verify the details of the software, how it is designed and implemented. Black box tests verify the functionality and quality of the software without consideration of its implementation. Live tests validate the software behavior under operational conditions. White and black box tests are primarily done to identify faults while live tests are primarily done to prove that the software works as intended during operation.

2.2.1 White Box Tests

White box testing (also called code based testing or structure testing [6]) is done to verify that a piece of software is implemented and works as intended. During structure testing all source code should be verified, that means that every statement should be executed at least once and all paths through the code should be followed [5]. Structure testing can be done with a debugging tool, which allows evaluation and manipulation of the code during execution, and with trace logs.

2.2.2 Black Box Tests

Black box testing (also called specification based testing or functional testing [6]) is done to verify a software component or system’s behavior without consideration to how the software is implemented. The black-box test looks at what output the software returns when given a certain input and starting in a particular state [5]. For most software the combination of all possible inputs, starting states and expected outputs is almost infinite which makes it more or less impossible to test every combination. One good approach to design test cases for black-box testing is to identify all equivalence partitions that has to be tested and then pick at least one set of test data from each partition [4,5]. Another good approach is to identify the operational profile of the software. The operational profile of the software reflects how it will be used in practice [4,8] or put in other words the operational profile is the distribution of use cases when the system is in normal operation [7]. The operational profile is a specification of types of inputs and the probability of their occurrence and it is found by tracing the usage of the software during normal operation. This means that it is quite difficult to identify the operational profile for a completely new software, at least one version of the software, including trace loggers, has to be put in operation and be used for a while before enough data is collected to identify the operational profile [8]. The operational profile can be used to design test cases for statistical testing.

Probing can be used to observe the state (parent-child, threads, stacks, resources, signals, events, queues, system calls and files) of individual software components within a component-based system during execution, or postmortem. Probing is done by means of tools that are usually specific to, and distributed with the actual operating system [10].

Snooping can be used to observe the communication between two or more components within a component-based system during execution. Also snooping is done by means of tools that are usually specific to the operating system [10].
Spoofing can also be used to observe the communication between components but in this case by means of interposing an observer between the components. Some spoofing tools exist and they are not so dependent on the operating system as probing and snooping tools [10]. Spoofing can also be done by developing specific test components and interpose them between the components that should be observed.

System-level fault injectors test the system by creating errors in it [7]. One fault injection technique is the so-called interface propagation analysis (IPA). IPA corrupts the states propagated through the component interfaces by means of for example a specific test component interposed between two system components, the same technique as spoofing. The test component replaces the original state propagated between the system components with a corrupt state during software execution.

2.2.3 Live Tests

Statistical testing is used to check how the software works under operational conditions. The test cases are designed to, based on the operational profile, simulate normal software usage and the goal is to get data to estimate the software reliability and performance during normal system operation.

Operational testing of a software system and its integrated components is performed during system operation. Components should be deployed in such a way that any faults leave logged traces so failures in the software systems in operation can be traced [1]. The system should be deployed in such a way that the usage of the system is traced and the operational profile can be identified.

3 Defect Prevention

To decrease the effort needed to identify software faults, the faults should not be introduced in the first place.

One way to prevent defects to be introduced into the source code is to use implementation languages and development tools fit for the purpose. A safe and expressive language rules out some of the possibilities to introduce defects, for example memory leaks, and also allows the compiler and source code analysis tools to catch faults early during implementation [1].

Another way to prevent the introduction of defects is to have a mature software development culture and use a defined software development process. With defined software development process means rules, instructions, procedures, guidelines, good practices, checklists, patterns, templates, examples, methods and tools and so on that guides the software development. With mature software development culture means that the software development process evolves as experience is gained. During for example software inspections knowledge and experience is captured that should be transferred back to the defined software development process [4]. This will prevent even more defects to be introduced in the future.

One example of example of a defined software development process is the Clean-room process [4]. The Clean-room process uses incremental development practice combined with a formal requirements change request procedure. All software is formally specified with for example state
transition models, and then verified with formal inspections against these specifications. The source code is written in a structured way exactly as stated in the specifications with only a few allowed constructions. Only statistical tests of the software, based on the operational profile, are performed to certify the system, no structure or specification testing is performed.

4 Software Certification

Software certification is nothing new; it has a long history in the software industry. The common use of certification is that some authoritative organization attests to the fact that some software satisfies some criteria or conforms to some specification [2]. Certification has two purposes, one is to establish facts about the software being certified, and the other is to establish trust in the validity of these facts. Certification of a custom developed software systems is not that difficult but certification of software component and component-based systems is a little bit more complicated [2]. Software components have to be certified before they are integrated into a system and they have to be certified for usage in many different configurations. A component certificate says (hopefully) something about the quality of the component, but (probably) nothing about the quality of the component-based system it will be a part of. Even if a component vendor certifies a component, the system builder still has to certify it for their specific usage [7].

Certification of a software component or system can be done in two ways. One is to let some trusted independent organization or institute perform an assessment of the software according to some established standard. The result from the assessment is then provided with the software. This way is probably both time-consuming and expensive, especially for components. Another way to certify the software itself is to do as in the process industry. Verification and validation of the software could be done according to some standard and the final verification and validation results are provided with the software. But no common practice is established for this yet.

A way to indirect certify software from the trust perspective, is to certify the software development process. ISO 9000 and TickIt assessments are commonly known certification methods. CMM (Capability Maturity Model) assessment [3] includes another method to assess the development process, but it is normally not regarded as a certificate in the same way as ISO9000. Whether certifying the development process will increase the trust in the software itself depends on how dependent the software quality is of the development process quality. If the dependency is high certifying the development process may prove to be an economical alternative to software certification [2]. For example, if a software component is developed by an organization that is certified to follow the Clean-Room Process completely the component will be implemented exactly as specified, that can be trusted, and the specifications can be read so also the facts about the component is available.

5 Verification & Validation and Component Based Software

All of the traditional verification and validation techniques are still valid in Component Based Software Development, but the main focus differs depending on if the task is to develop (produce) (re)usable components or if the task is to use (consume) components.
5.1 Verification & Validation from the Component Producer perspective

Producing reusable software components for the component market requires rigid verification and validation of the component. Implicitly then also rigid specifications is required. This gives that more effort will probably be invested in the component software than if custom software is developed. Component software development activities does not differ much from custom software development activities. Requirements have to be analyzed, the functionality and quality has to be specified and the specifications have to be verified with document and model inspections. The architecture and detailed design has to be specified although architectural design may be limited for “small” components. The design models should also be verified by means of inspections. The code has to be written, inspected and white box tested to verify its compliance with the design. The functionality and quality of the component has to be black box tested to verify the compliance with the corresponding specifications. Finally the component should be certified and deployed together with its certificate and other externally interesting specifications.

5.2 Verification & Validation from the Component Consumer perspective

OTS (Off The Shelf) components are often delivered in “black-boxes” as executables with licenses that forbid de-compilation back to source code [7]. Due to this they cannot be verified by means of white box testing, as the source code is not available. If an OTS-component is not delivered together with a component certificate, or the certificate is not trustable, different black-box tests have to be performed to provide information useful for the evaluation and selection of components and to certify it for usage. Quality problems using OTS-components is not just a matter of the quality of each component but also a matter of component integration compatibility.

The OTS Component Certification Process [7] suggests a combination of three verification and validation techniques to certify an OTS-component for usage; black box testing based on the systems operational profile, system-level fault injection and operational testing. After a component with the required functionality is selected and evaluated black box testing is performed to verify the quality of the component. If the identified defects are acceptable the component is integrated into a system that is then tested with system-level fault injections to validate system behavior due to component failure. Finally, if the component seems to be acceptable to the system, an operational testing is performed to validate that the component (and the system) is behaving correctly during normal system operation.

5.3 Verification & Validation of Component Based Software Systems

Component Based Software Systems development activities are basically the same as custom software development activities. Requirements have to be analyzed, the functionality and quality has to be specified and the specifications should be verified with document and model inspections. The addition is that requirements on functionality and quality of needed software components also have to be specified as input to the component selection and evaluation activities. The architecture and some detailed design also have to be specified. Especially the architecture design model should be verified by means of thorough inspections. Some code probably has to be written to integrate the selected components, the code should be inspected and white box tested to verify its compliance with the architecture and detailed design.
functionality and quality of the integrated system has to be black box tested to verify the compliance with the corresponding specifications. Finally the system should be certified, or tested for acceptance as the more common wording is in this case, and deployed.

In custom developed software systems faults can be found by performing white box testing but in component based software systems fault identification can be much more difficult [10] especially when OTS (Off The Shelf) components are used. Instead of finding defects in the source code, component behavior has to be observed and conclusions have to be made from the observations.

The approach for identifying faults in component based software systems is essentially the classic scientific method of observation, hypothesis, prediction, experiment and result [10]. Based on the observed result a conclusion is made if the hypothesis is correct or not. To be able to make observations when the source code not is available other means to get visibility must be used. Four common techniques can be used to gain visibility into OTS-components namely probing, snooping, spoofing and static program analysis [10]

6 Summary

The success of component based software technology is dependent on the fact that the effort needed to build component based software systems can be decreased significantly compared to traditional custom software development. Verification and validation has always been major activities in software development and if at least the verification effort can be decreased, a lot is gained. But that requires, among other things, that the components used has the right functionality and quality and are well documented.

Developing software components that should be reused, and maybe also commercial, requires a high focus on software quality. In mature software development organizations high quality can be ensured by means of an appropriate defined development process and appropriate methods and tools, that is defect prevention. High quality can also be ensured by means of thorough verification and validation, that is defect identification. The quality of the component should be certified in a, for the component purchasers, trusted way.

There are two main techniques to identify defects in software, inspection and test. Inspections can be done, on all software work products such as documents, models and source code. Tests can only be done on executable code. Inspections can be more efficient and effective then tests as more faults can be found per session and the faults can be identified earlier in the development process. Specifications, defining the correct structure and behavior of the software, are needed to support verification and validation so that defects can be identified.

The only thing new regarding verification and validation when developing software components, compared to custom software development, is the issue of proofing, or certifying, the software functionality and quality. The verification and validation activities when developing component based software systems differ more, especially if the components used are of poor quality, where the verification of the software can be a very difficult task.

Christina Wallin: Verification and Validation of Software Components and Component Based Software Systems

Chapter 5: Component-Based Development Process, B. Christiansson, L. Jakobsson and I. Crnkovic
7 References


Christina Wallin: Verification and Validation of Software Components and Component Based Software Systems


Chapter 5: *Component-Based Development Process*, B. Christiansson, L. Jakobsson and I. Crnkovic