Software Component Testing and Test Automation in Embedded Systems.

Author:

Mahesh Pande
Designation: Senior Software Specialist
Email: mahesh.pande@patni.com

Company Address :
IT1/IT2 Patni Knowledge Park (PKP)
Block C, 2nd Floor, Wing A (C2A)
Thane Belapur Road, Airoli
Navi Mumbai 400 708

Abstract

The intent of this paper is to impart an understanding on the need for doing software component testing in embedded systems and also give a viable solution for testing and test automation of the software components involved in the embedded systems.

Verification and Validation (V&V) activities focus on both the quality of the software product and the engineering process involved in building the product. These V&V activities can be sub-classified as preventive, detective, or corrective measures of quality. While testing is most often regarded as a detective measure of quality, it is closely related to preventive as well as corrective measures. In practice, software developers usually find it more productive to enact testing and debugging together, usually as an interactive process.

This paper addresses the need for the software component testing in embedded systems because software now makes up 90 percent of the value of the embedded system devices. More so, the software architecture of the embedded system devices, in most cases, is componentized in nature. Further to add, at the system level, embedded system devices are extremely hard to test and the defects detected are further more complicated to debug and fix at the system level. Hence, if the software part is tested first by implementing component testing, then, the software components are ensured to be with minimal defects or with no critical defects when the system testing phase begins.

Table of Contents

1. Introduction

In order to settle on a common set of concepts, let's start with some definitions:

Embedded System: It's difficult, and highly controversial, to give a precise definition of embedded system. So, here are some examples. Embedded systems are in every "intelligent" device that is infiltrating our daily lives: the cell phone in your pocket, and the entire wireless infrastructure behind it; the Palm Pilot on your desk; the Internet router your e-mails are channeled through; your big-screen home theater system; the air traffic control station as well as the delayed aircraft it is monitoring.

Software Component: A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

Software Component Testing: Component testing is the act of subdividing an object-oriented software system into units of particular granularity, applying stimuli to the component's interface and validating the correct responses to those stimuli, in the form of either a state change or reaction in the component, or elsewhere in the system.

2. The relevance of Software Component Testing in Embedded Systems

Before discussing the relevance of software component testing in embedded systems, let us discuss the high level features the software component architectures offer.
The software component architectures are:

a. Organized
b. Loosely Coupled
c. Support Event Bus Model
d. Maintainable
e. Reusable
f. Extensible

All of the above features make software component architectures an ideal place for software development in embedded system devices.

Having said the above, it is imperative that the testing of such software that conforms to a componentized architecture should take the advantage of its architectural benefits to test early and test efficiently.

2.1 Illustration for the need of software component tests

To illustrate briefly the need for doing software component testing in embedded systems, let us consider the following scenario:

Let us assume that there is an embedded system device under development which follows a componentized architecture. The software development is in iterative fashion. The classes that make up a software component are unit tested while development during the same iteration and system tested at the end of the iteration. The system test defects of the earlier iteration are therefore fixed in the next adjacent iteration(s). This development model seems to work just as defined because it follows an iterative and incremental approach. However, there is one basic flaw with this approach.

The flaw is that the defects induced during iteration are fixed in the next adjacent iteration(s). This is termed as a flaw because, it is the number of defects found and the criticality of the defects detected in the earlier iteration that defines the amount of time it would take to fix those defects in the next adjacent iteration(s). This causes an undue delay in the software development. Such a delay is hard to predict and difficult to plan.

This flaw can be overcome by doing software component testing as and when the software components are developed. The software components are tested in isolation and also in conjunction with other components, as integration tests.

2.2 Where does the software component testing begin in the SDLC?

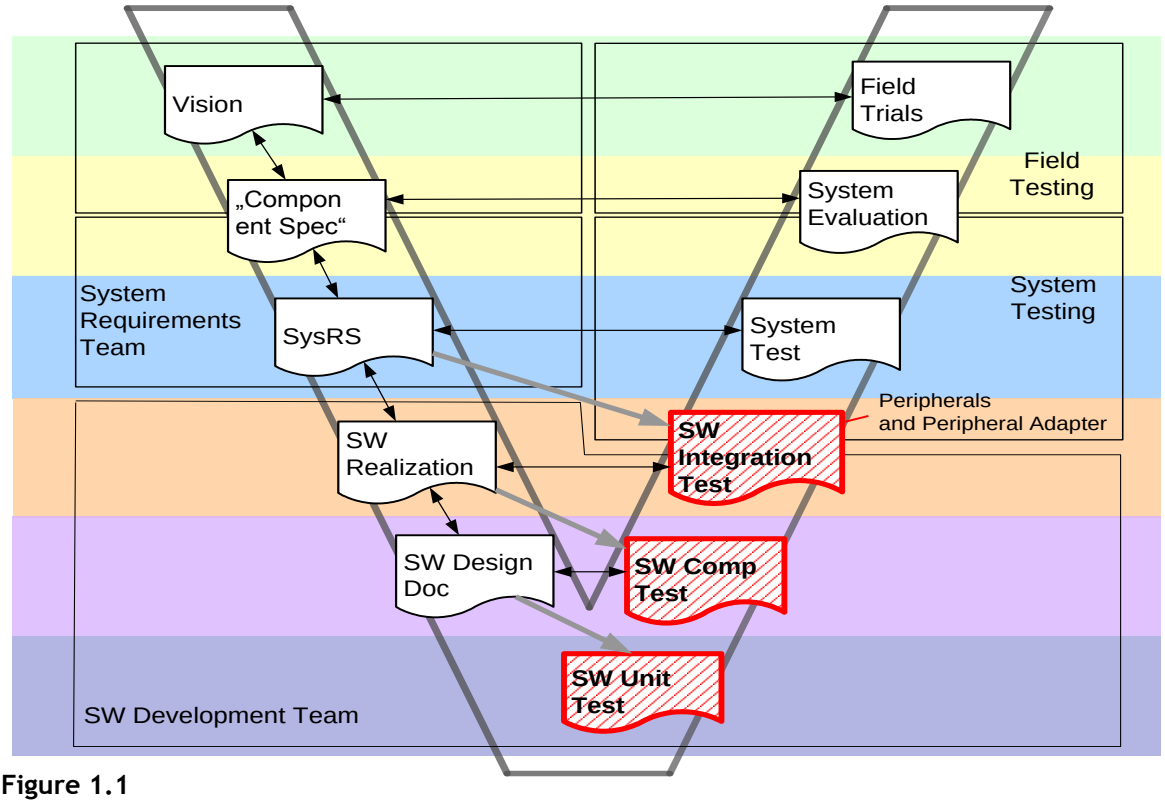


Figure 1.1

As seen from the above “V” development paradigm, unlike system tests, the software component tests does not just rely on the System Requirement Specification (SRS) and/or use cases as the input.
The inputs required for the software component tests are:
a. System Requirement Specification – To understand the basic functionality of the software components.
b. Software Design Document – To understand how many Software Components exists in the system and how they collaborate with each other as a system.
c. Software Realization – In terms of sequence diagrams to understand the sequence of messages and/or events through which the components collaborate with each other for a given functionality. Software component tests are just a replica of these sequence diagrams from the functionality verification perspective.

In most of the cases, SW Component Tests provide design level information to the software architect in terms of collaboration charts.

2.3 Test stages and Test Types

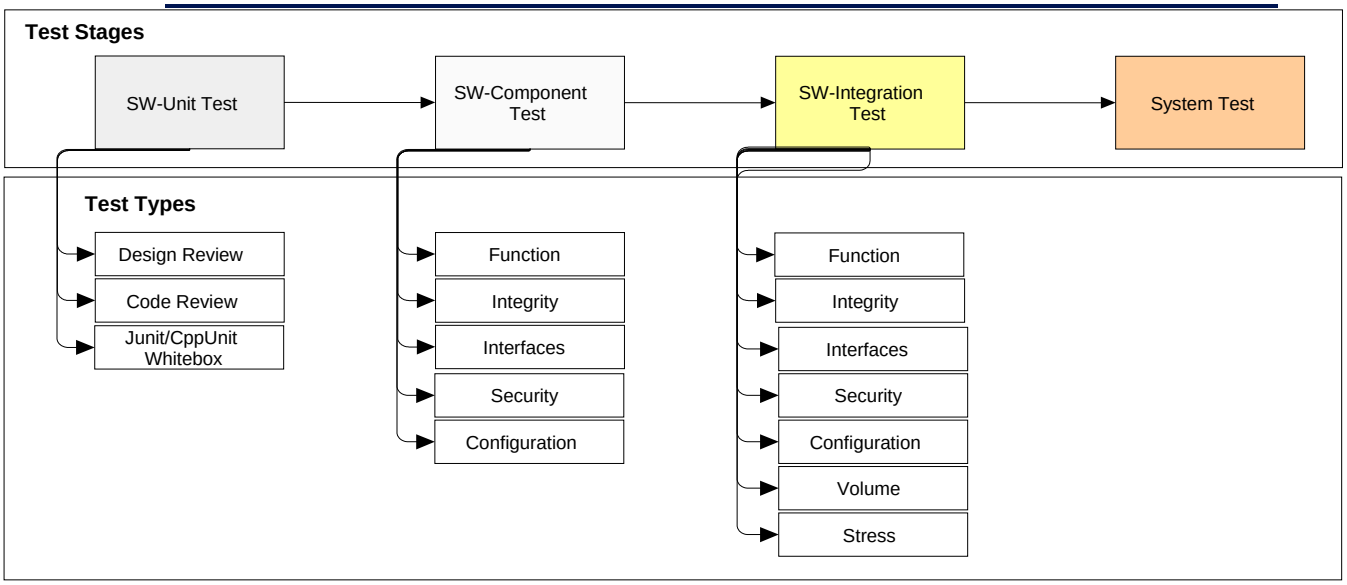


Figure 1.2

As seen from the above diagram, after the individual classes that make up a component are unit tested and after the unit tests are successful, would the SW Component test stage begin.

SW Component Tests verify the SW Component in isolation and also verify the collaboration of various SW components as an integration test for a given functionality. However, the scope of the software component tests is not just limited to the functionality tests.

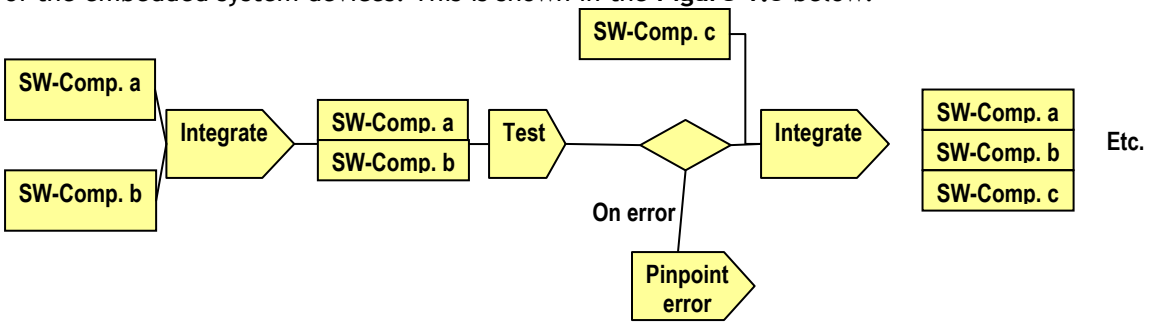
SW component tests may involve:

a. Functionality testing
b. Integrity testing
c. Testing the involved interfaces
d. Security testing
e. Configuration testing.

SW Component tests as an integration test may also involve:

f. Volume testing
g. Stress testing.

SW Component tests facilitate a step-by-step integration process in the software development of the embedded system devices. This is shown in the **Figure 1.3** below:



2.4 Overview of Software Component Test Design

There are several approaches to the SW Component Test Design enlisted in numerous informative websites and blogs. Some of the SW Component test Design techniques also use the component test design patterns.

The test design approach presented in this paper is to take advantage of the sequence diagrams in the SW Architecture Design Model of the embedded system device to create software component tests. In other words, the approach suggested in this paper for component test design is as per the "***Test Sequence***" component test design pattern.

Please refer the section 2.2 and figure 1.1 of this paper for the inputs of software component test design using this approach.

According to the information in the sequence diagrams a component test is created for each component involved within that functionality. Such a component test should have:

a. A test goal
b. Precondition(s)
c. Trigger
d. Actions to be performed
e. Responses to be verified
f. Post Condition

This test design approach at the SW component level, concentrates on verification of all the possible input stimuli for each and every component in isolation, for a given functionality under test.

This test design approach at the integration level, concentrates on verification of the collaboration of the involved components within a given functionality under test.

For example – For a given functionality, say, there are three SW components involved: Component 'A', 'B', and 'C'.
Component 'A' could take 'n' number of input parameters and give a response which is chained to 'B' and from 'B' to 'C'.

This approach suggests to first test Comp 'A', Comp 'B' and Comp 'C' in isolation as a component test and then as a SW Component Integration, test the collaboration of these three components for that particular functionality under test.

In such a test design approach, we employ both "top-down" and "bottom-up" test strategy.

For the software component tests to verify each component in isolation, using this approach, we employ "bottom-up" test strategy which means, the components are individually tested using specially written drivers that provide the necessary functions. This activity necessitates that, till the lower dependent components are ready for use they need to be simulated using program stubs.

For the software component tests during integration testing, using this approach we employ "top-down" test strategy which means, that the initial tests establish a basic system skeleton and each new module adds capability.

3. Overview of Software Component Test Automation

The basic thing required is a software simulation environment of the embedded device where the software components under test can be deployed and tested.
Assuming the above as a prerequisite, the challenge to test framework developers is to recognize the common problems and associated implementation activities, to the extent that scalable and easily maintainable automated test architecture can be applied as a foundation for implementing tests. Our experience has shown that for most types of component testing, the component test developer must eventually find answers to one or more of the following questions, and possibly the questions that their answers pose:

- How do I create a specifically configured instance of a component under test?
- How to I manage the application of a stimulus to the component under test?
- How do I organize stimuli into reusable groups of arbitrary granularity?
- How to I validate the state of the component and system under test after one or more stimuli?
    - How do I manage reference objects?
        - How do I compare references objects to target objects, so that just the states that are important are compared?
    - How do I intelligently traverse the state of my object under test
- How do I validate that expected exceptions are raised.
- How do I manage variations on test inputs in order to drive different paths with the same test stimuli?
- How do I map development activities to the tests that validate their correctness?

The activities for software component test automation include:

a. Preparation of the test environment
b. Evaluation of a possible test automation tool to satisfy the need for test automation.
c. Evaluation of a need of a library / framework to act as middleware between the component under test and the simulation environment.
d. Identification of Reusable steps that can be clubbed together in the form of reusable test procedure library
e. Creation of such a test procedure library
f. Creation of test scripts.
g. Creation of report generation libraries
h. Creation of utilities to save application logs for failed tests
i. Execution of test scripts
    a. Grouping together the test scripts as a module test driver.
    b. Command line execution of the test scripts
    c. Batch execution / selective execution of test scripts through a batch file
    d. Implementing auto generated emails for reporting purpose (ANT build scripts can be of use).

3.1 An approach to test automation using IBM Rational Test RealTime for an elevator system

The following block diagram explains the solution for testing / test automation:

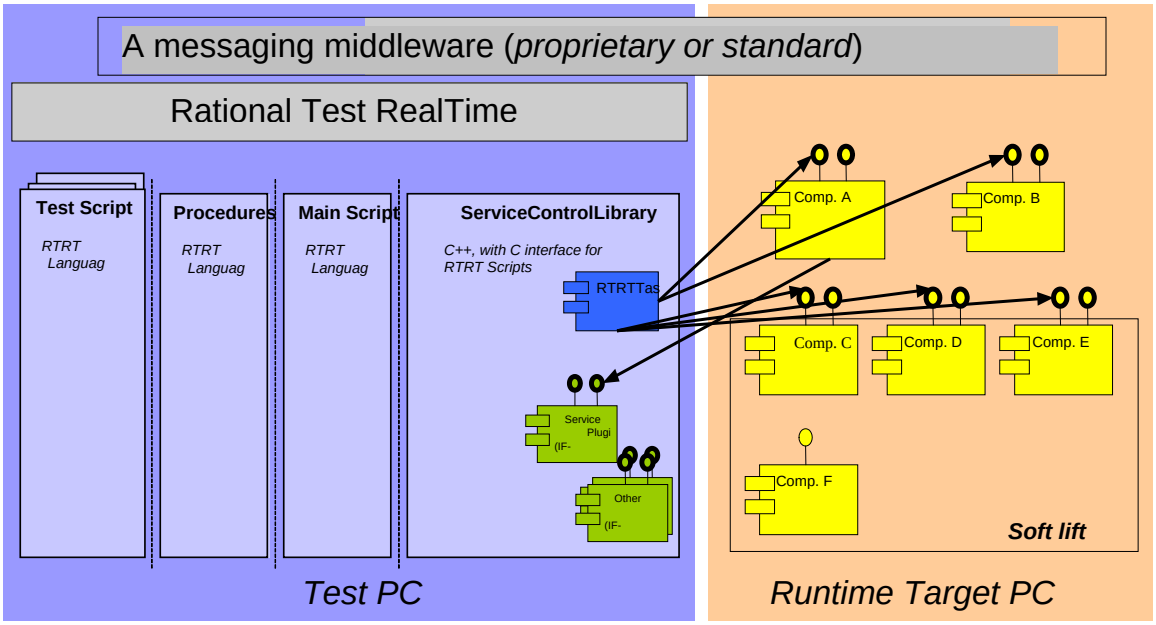


Figure 1.4

A brief explanation of the given solution:

1. **Test Script** – Test Case implementation which is to be created by the test engineer according to the test case definition using Rational Test Real Time (RTRT).
2. **Procedures** – Implementation of commonly used functions using RTRT to access embedded system device Interfaces. This is to be adapted by the test engineer according to the test case / test script definition.
3. **Main Script** – Implementation of generic routines using RTRT for:
   - Initialization
   - Central Send Routines
   - Central Callback Routines
   - Exception Handling

   A central main test script gives an advantage of creating a generic library of test procedures so that there are no adaptations needed in the test scripts when modifying test cases or embedded system device Interfaces

4. **ServiceControlLibrary** – A C/C++ Library to provide access to the Software Components within the embedded system device that:
   - Sends events through Remote Access Protocol (RAPs)
   - Subscribes to read (RAPs)
   - Simulate a publisher (Read RAP – often called as test stub)
   - Simulate a receiver (Even RAP – often called as test stub)

   This library auto-generates the RAP definitions by reading the interface definitions used by Rational Test RealTime (RTRT) test scripts.
   This library requires a recompile in case of modifications to the system interfaces.

Runtime Target – This is the system under test. This contains various components under test.

4. Software Component Testing / Test Automation Process

The following diagram represents the software component testing / test automation process as described by this paper:

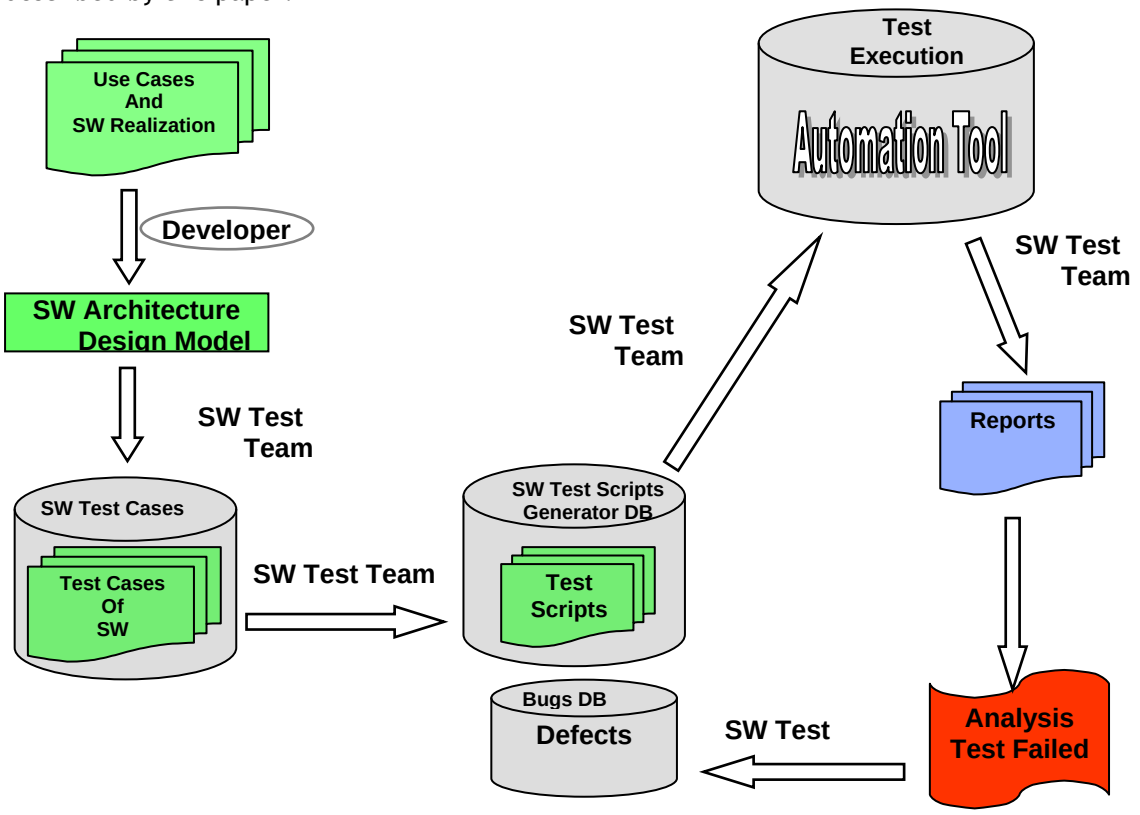


Figure 1.5

5. Conclusion

The software component testing and test automation is a means of exploring the “preventive” and “corrective” aspects of the Verification & Validation. This testing is of utmost importance when it comes to embedded system device development because through this testing, there is a timely and a necessary check on the embedded system software development, which in its absence, could lead to a software functionality defect found only in system testing thereby causing undue delay in the time to market proposition of the embedded systems device.

Also, unlike system testing, software component testing does not necessarily require the hardware infrastructure. A reliable software simulation environment is good enough for doing software component testing. (Any embedded system device like: elevators, automotives, flights, pacemakers, mobile phones, PDA’s have reliable software simulation environment in place). This point makes the task an ideal candidate to be executed from offshore, reaping the cost benefits of offshorization.

Having said the positive things about software component testing, there are still some concerns that theoretically and practically are yet to be addressed in Software Component Testing for embedded system devices. These are:

a. Effective testing strategies need to be made for testing domain specific component software and developed tests can be stored to be reused later
b. If the metadata is considered to be a potential solution to the problem of component testing, then Metadata standards creation will need a lot of cooperation and coordination among the various third party component producers of embedded system devices around the world.
c. Reliability of components can be improved by improving the languages used to implement them (like Java has popularized the use of a garbage collector).
d. The range of test scenarios should be more comprehensive for making the components cater to a wide range of usage patterns
e. Additional techniques like providing extensive Component User Manuals and Application Interface Specifications can be considered too.

6. References

[1] Clemens Szyperski, Component Software- Beyond Object Oriented Programming, *Addison Wesley*, 1997

[2] E.J.Weyuker, Testing Component-Based Software: A Cautionary Tale, *IEEE Software,* Vol. 15, No. 5, September/October 1998

[3] Jefferey M.Vaos, Certifying Off-the Shelf-Components, *IEEE Computer*, June 1998

[4] Jefferey M.Vaos, A Defensive Approach to Certifying COTS Software, *Technical Report, Reliable Software Technologies Corporation*, August 1997

[5] Allessandro Orso, Mary Jean Harrold, David Rosenblum, Component Metadata for Software Engineering Tasks, In *Proc. 2nd International Workshop on Engineering Distributed Objects*, Davis, CA, November 2000.

[6] Gary A. Bundell, Gareth Lee, John Morris, Kris Parker, A Software Component Verification Tool, *In the Proceedings of International Conference on Software Methods and Tools*, 2000. SMT, 2000

[7] Hoijin Yoon, Byoungju Choi, Jin-Ok Jeon, A UML Based Test Model for Component Integration Test, *Workshop on Software Architecture and component (WSAC), Japan*, 1999

[8] Wayne Liu and P. Dasiewicz, Formal Test Requirements for Component Interactions, *IEEE Canadian Conference on Electrical and Computer Engineering*, 1999

[9] Yingxu Wang, Graham King, Hakan Wickburg, A method for Built-in Tests in Component-based Software Maintenance, Proceedings of the Third European Conference on Software Maintenance and Reengineering, 1999

[10] John D. McGregor, Component Testing,JOOP Column,1997

[11] John D. McGregor and Anuradha Kare, Parallel Architecture for Component Testing, In *Proceedings of the Ninth International Quality Week*, 1996.

7. Author's Biography

Mahesh Pande is a senior software specialist at Patni having over 11 years of software experience.

This experience is spread across design, development and testing domains of Software Development Life Cycle. In testing and test automation of software applications, Mahesh has over 7 years of experience. He has an expertise in presentation layer testing and API level testing. He was also involved in testing swing applications at Sun Microsystems Inc for 2 years.