Optimization of Automation Environment for Set-Top Box

By

SUNIL MISHRA

10MCEC28

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

AHMEDABAD-382481

MAY 2012

Optimization of Automation Environment for Set-Top Box

Major Project

Submitted in partial fulfillment of the requirements

For the degree of

By

SUNIL MISHRA

10MCEC28

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

AHMEDABAD-382481

MAY 2012

Declaration

This is to certify that

i) The thesis comprises my original work towards the degree of Master of Technology in Computer Engineering at Nirma University and has not been submitted elsewhere for a degree.

ii) Due acknowledgement has been made in the text to all other material used.

SUNIL MISHRA

Certificate

This is to certify that the Major Project Part-II entitled "Optimization of Automation Environment for Set-Top Box" submitted by Sunil J. Mishra (10MCEC28), towards the partial fulfillment of the requirements for the degree of Master of Technology in Computer Science and Engineering of Nirma University of Science and Technology, Ahmedabad is the record of work carried out by him under my supervision and guidance. In my opinion, the submitted work has reached a level required for being accepted for examination. The results embodied in this major project, to the best of my knowledge, haven't been submitted to any other university or institution for award of any degree or diploma.

Prof. Madhuri Bhavasar
Internal Guide,
Department of Computer Engineering,
Institute of Technology,
Nirma University,Ahmedabad

Dr. S.N. Pradhan
Professor and Co-ordinator
Department of Computer Engineering,
Institute of Technology,
Nirma University,Ahmedabad

Prof. D. J. Patel
Professor and Head,
Department of Computer Engineering,
Institute of Technology,
Nirma University, Ahmedabad

Dr K Kotecha
Director,
Department of Computer Engineering,
Institute of Technology,
Nirma University, Ahmedabad

Abstract

Testing is required for every product or s/w, so that the product delivered to the customer is free from defects, highly stable and reliable. Automated Testing is an alternative to Manual testing that can reduce the time and cost spent over testing. Also Regression testing, Stress testing and Stability testing become very easy with the help of automation.

In the current environment software applications built to do the automated testing are highly dependent on the machine on which they run and the devices they interact with. Hence failure in either of them can have a high impact on the time and cost of testing.

The focus of the project is to create an environment where testing applications can run without being affected by system and network failures. Hence to provide a stable environment for applications to run without failure, an independent framework has to be developed which can aid the applications running on top of it, to be reliable and stable. The framework will handle all the additional tasks like process recovery, error handling, backup and scheduling, so that the applications can focus on the main task i.e. of testing.

Acknowledgements

With immense pleasure, I would like to present this report on the dissertation work related to "Optimization of Automation Environment for Set-Top Box". I am very thankful to all those who helped me for the successful completion of the first phase of the dissertation and for providing valuable guidance throughout the project work.

I would first of all like to offer thanks to **Dr. S. N. Pradhan**, Programme Co-ordinator M.Tech CSE, Institute of Technology, Nirma University, Ahmedabad, **Prof. Madhuri Bhavasar**, Guide, Institure of Technology, Nirma University,Ahmedabad and **Mrs. Anitha Pushpanathan**, Team-Lead, ATG-SIT, Motorola Mobility India Pvt. Ltd., Bangalore, whose keen interest and excellent knowledge base helped me to finalize the topic of the dissertation work.Their constant support and interest in the subject equipped me with a great understanding of different aspects of the required architecture for the project work. They have shown keen interest in this dissertation work right from beginning and have been a great motivating factor in outlining the flow of my work.

My sincere thanks and gratitude to **Mr. Manoj John Gerald**, Project Manager, ATG-SIT, Motorola Mobility India Pvt. Ltd., Bangalore for his continual kind words of encouragement and motivation throughout the Dissertation work.

I am thankful to Motorola Mobility for providing all kind of required resources. I would like to thank The Almight and my family, for supporting and encouraging me in all possible ways. I would also like to thank all my friends who have directly or indirectly helped in making this dissertation work successful.

- Sunil Mishra

10MCEC28

Abbreviation Notation and Nomenclature

AUT ..... Application Under Test
DAC ..... Digital Addressable Controller
GUI ..... Graphical User Interface
HDD ..... Hard Disk Drive
NIC ..... Network Interface Card
SIT ..... System Integration and Test
STB ..... Set Top Box

Contents

Declaration iii

Certificate iv

Abstract v

Acknowledgements vi

Abbreviation Notation and Nomenclature vii

List of Figures x

1 Introduction 1
- 1.1 Testing . . . 1
- 1.2 SIT Testing and Benefits . . . 2
- 1.3 Benefits of Automation . . . 3
- 1.4 General Approaches to Test Automation . . . 6
- 1.5 Code-driven testing . . . 6
- 1.6 Graphical User Interface (GUI) testing . . . 7
- 1.7 Framework approach in Automation . . . 8

2 Current Environment 9
- 2.1 Current Architecture . . . 9
- 2.2 Automation Framework . . . 11

3 Problem in the current environment 14
- 3.1 Types of Failures . . . 14
- 3.2 Statistics . . . 16
- 3.3 Previous Work . . . 17

4 Proposal 18
- 4.1 A Software Framework . . . 18

5 Challenges 22

6 Implementation **24**

7 Enhancement **28**
7.1 Exploratory Testing 28
7.2 Motivation 29
7.3 Pros and Cons 31
7.4 Research Questions 33
7.5 Rule Based Exploratory Testing - A Custom Approach 34
7.6 Research Questions - Answers 41

8 Conclusion **43**

9 Future Implementation **47**

List of Figures

2.1 Architecture of Automation System with Other External Devices . . 10
2.2 Current Automation Framework 13

3.1 Statistics of Manual vs Automation 16

4.1 Proposed architecture 19
4.2 Description of Monitors 20

6.1 Local System View of Monitors 25
6.2 Remote System View of Monitors 26

7.1 Architecture of Custom Rule Based Exploratory Testing 38
7.2 Sample Structure of a Rule 38
7.3 Sample Manual Exploratory Scenario 39
7.4 Sample Manual Exploratory Scenario Execution 39
7.5 Sample Manual Exploratory Scenario Results Screen 40
7.6 Sample Manual Exploratory Scenario Summary 40

8.1 Comparison Between Existing And New Environment 43
8.2 Comparison Between Existing And New Environment 43
8.3 Cumulative Time Saving in the New Environment 44
8.4 Cycle Time Saving in the New Environment 45
8.5 Staff Time Saving in the New Environment 45
8.6 Efficiency of Framework 46
8.7 Performance of Framework in the New Environment 46

Chapter 1

Introduction

1.1 Testing

Testing is the process of executing a program or system with the intent of finding errors. Or, it involves any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results.

Testing is an integral part in software or product development. It is broadly deployed in every phase in the development cycle. Typically, more than 50 percent of the development time is spent in testing.

Testing is an investigation conducted to provide stakeholders with information about the quality of the product or service under test. Testing can also provide an objective, independent view of the software to allow the business to appreciate and understand the risks of software implementation. Test techniques include, but are not limited to, the process of executing a program or application with the intent of finding software bugs (errors or other defects).

Testing can be stated as the process of validating and verifying that a software

program/application/product:

- meets the requirements that guided its design and development;
- works as expected; and
- can be implemented with the same characteristics.

Testing, depending on the testing method employed, can be implemented at any time in the development process. However, most of the test effort occurs after the requirements have been defined and the coding process has been completed. As such, the methodology of the test is governed by the software/product development methodology adopted.

1.2 SIT Testing and Benefits

SIT is part of the software testing life cycle for collaborative projects. Usually, round of SIT precedes the user acceptance test (UAT) round. And software providers usually run a pre-SIT round before consumers run their SIT test cases.

As an example, if integrator (company) is providing an enhancement to customer's existing solution, then they integrate the new application layer and the new database layer with existing customer's application and existing database layers. After the integration completes, users use the new part (extended part) of the integrated application to update data. Along they use old part (pre-existing part) of the integrated application. A process should exist to exchange data imports and exports between the two data layers. This data exchange process should keep both systems up-to-date. Purpose of the system integration testing is to make sure whether these systems are successfully integrated and been up-to-date by exchanging data with each other.

Testing is usually performed for the following purposes:

- To improve quality.
  - Quality of the product can be increased when the defects in the product can be identified before the product gets deployed.
- For Verification and Validation
  - Testing ensures that the product or the system is in accordance to the requirements specified and functions exactly the way it needs to.
- For reliability estimation
  - The products reliability can be estimated, which indicates the measure of the stability of the product.

1.3 Benefits of Automation

Testing can be very costly. Automation is a good way to cut down time and cost. Testing tools and techniques usually suffer from a lack of generic applicability and scalability. The reason is straight-forward. In order to automate the process, we have to have some ways to generate oracles from the specification, and generate test cases to test the target system against the oracles to decide their correctness. Today we still don't have a full-scale system that has achieved this goal. In general, significant amount of human intervention is still needed in testing. The degree of automation remains at the automated test script level.

Every software development group tests its products, yet delivered software always has defects. Test engineers strive to catch them before the product is released but they always creep in and they often reappear, even with the best manual testing processes. Automated software testing is the best way to increase the effectiveness,

efficiency and coverage of your software testing. Manual software testing is performed by a human sitting in front of a computer carefully going through application screens, trying various usage and input combinations, comparing the results to the expected behavior and recording their observations. Manual tests are repeated often during development cycles for source code changes and other situations like multiple operating environments and hardware configurations. An automated software testing tool is able to playback pre-recorded and predefined actions, compare the results to the expected behavior and report the success or failure of these manual tests to a test engineer.

Once automated tests are created they can easily be repeated and they can be extended to perform tasks impossible with manual testing. Because of this, savvy managers have found that automated software testing is an essential component of successful development projects. The benefits of automation are:

Automated Software Testing Saves Time and Money

Software tests have to be repeated often during development cycles to ensure quality. Every time source code is modified software tests should be repeated. For each release of the software it may be tested on all supported operating systems and hardware configurations. Manually repeating these tests is costly and time consuming. Once created, automated tests can be run over and over again at no additional cost and they are much faster than manual tests. Automated software testing can reduce the time to run repetitive tests from days to hours. A time savings that translates directly into cost savings.

Automated Software Testing Improves Accuracy

Even the most conscientious tester will make mistakes during monotonous manual

testing. Automated tests perform the same steps precisely every time they are executed and never forget to record detailed results.

Automated Software Testing Increases Test Coverage

Automated software testing can increase the depth and scope of tests to help improve software quality. Lengthy tests that are often avoided during manual testing can be run unattended. They can even be run on multiple computers with different configurations. Automated software testing can look inside an application and see memory contents, data tables, file contents, and internal program states to determine if the product is behaving as expected. Automated software tests can easily execute thousands of different complex test cases during every test run providing coverage that is impossible with manual tests. Testers freed from repetitive manual tests have more time to create new automated software tests and deal with complex features.

Automated Software Testing Does What Manual Testing Cannot

Even the largest software departments cannot perform a controlled web application test with thousands of users. Automated testing can simulate tens, hundreds or thousands of virtual users interacting with network or web software and applications.

Automated Software Testing Helps Developers and Testers

Shared automated tests can be used by developers to catch problems quickly before sending to QA. Tests can run automatically whenever source code changes are checked in and notify the team or the developer if they fail. Features like these save developers time and increase their confidence.

Automated Software Testing Improves Team Morale

This is hard to measure but weve experienced it first hand, automated software testing can improve team morale. Automating repetitive tasks with automated software testing gives your team time to spend on more challenging and rewarding projects. Team members improve their skill sets and confidence and, in turn, pass those gains on to their organization

1.4 General Approaches to Test Automation

Code-driven testing

The public (usually) interfaces to classes, modules or libraries are tested with a variety of input arguments to validate that the results that are returned are correct.

Graphical user interface testing

A testing framework generates user interface events such as keystrokes and mouse clicks, and observes the changes that result in the user interface, to validate that the observable behavior of the program is correct.

1.5 Code-driven testing

A growing trend in software development is the use of testing frameworks such as the xUnit frameworks (for example, JUnit and NUnit) that allow the execution of unit tests to determine whether various sections of the code are acting as expected under various circumstances. Test cases describe tests that need to be run on the program to verify that the program runs as expected. Code driven test automation is a key feature of Agile software development, where it is known as Testdriven development (TDD). Unit tests are written to define the functionality before the code is written.

Only when all tests pass is the code considered complete. Proponents argue that it produces software that is both more reliable and less costly than code that is tested by manual exploration. It is considered more reliable because the code coverage is better, and because it is run constantly during development rather than once at the end of a waterfall development cycle. The developer discovers defects immediately upon making a change, when it is least expensive to fix. Finally, code refactoring is safer; transforming the code into a simpler form with less code duplication, but equivalent behavior, is much less likely to introduce new defects.

1.6 Graphical User Interface (GUI) testing

Many test automation tools provide record and playback features that allow users to interactively record user actions and replay them back any number of times, comparing actual results to those expected. The advantage of this approach is that it requires little or no software development. This approach can be applied to any application that has a graphical user interface. However, reliance on these features poses major reliability and maintainability problems. Relabelling a button or moving it to another part of the window may require the test to be re-recorded. Record and playback also often adds irrelevant activities or incorrectly records some activities. A variation on this type of tool is for testing of web sites. Here, the "interface" is the web page. This type of tool also requires little or no software development. However, such a framework utilizes entirely different techniques because it is reading HTML instead of observing window events.

Another variation is scriptless test automation that does not use record and playback, but instead builds a model of the Application Under Test (AUT) and then enables the tester to create test cases by simply editing in test parameters and conditions. This requires no scripting skills, but has all the power and flexibility of a scripted approach. Test-case maintenance seems to be easy, as there is no code to

maintain and as the AUT changes the software objects can simply be re-learned or added. It can be applied to any GUI-based software application. The problem is the model of the AUT is actually implemented using test scripts, which have to be constantly maintained whenever there's change to the AUT.

1.7 Framework approach in Automation

A framework is an integrated system that sets the rules of Automation of a specific product. This system integrates the function libraries, test data sources, object details and various reusable modules. These components act as small building blocks which need to be assembled to represent a business process. The framework provides the basis of test automation and simplifies the automation effort.

Chapter 2

Current Environment

2.1 Current Architecture

The current existing automation architecture in the organization is given in the figure 2.1:

Description of the devices is given below:

System PC

The system used for automation, on which automation tool and other necessary drivers are installed.

DAC (Digital Addressable Controller)

A headend device which is used to stream videos to the set-top box and do other functions like initializing the box, resetting the box, etc.

Webcam/Video Capture Card

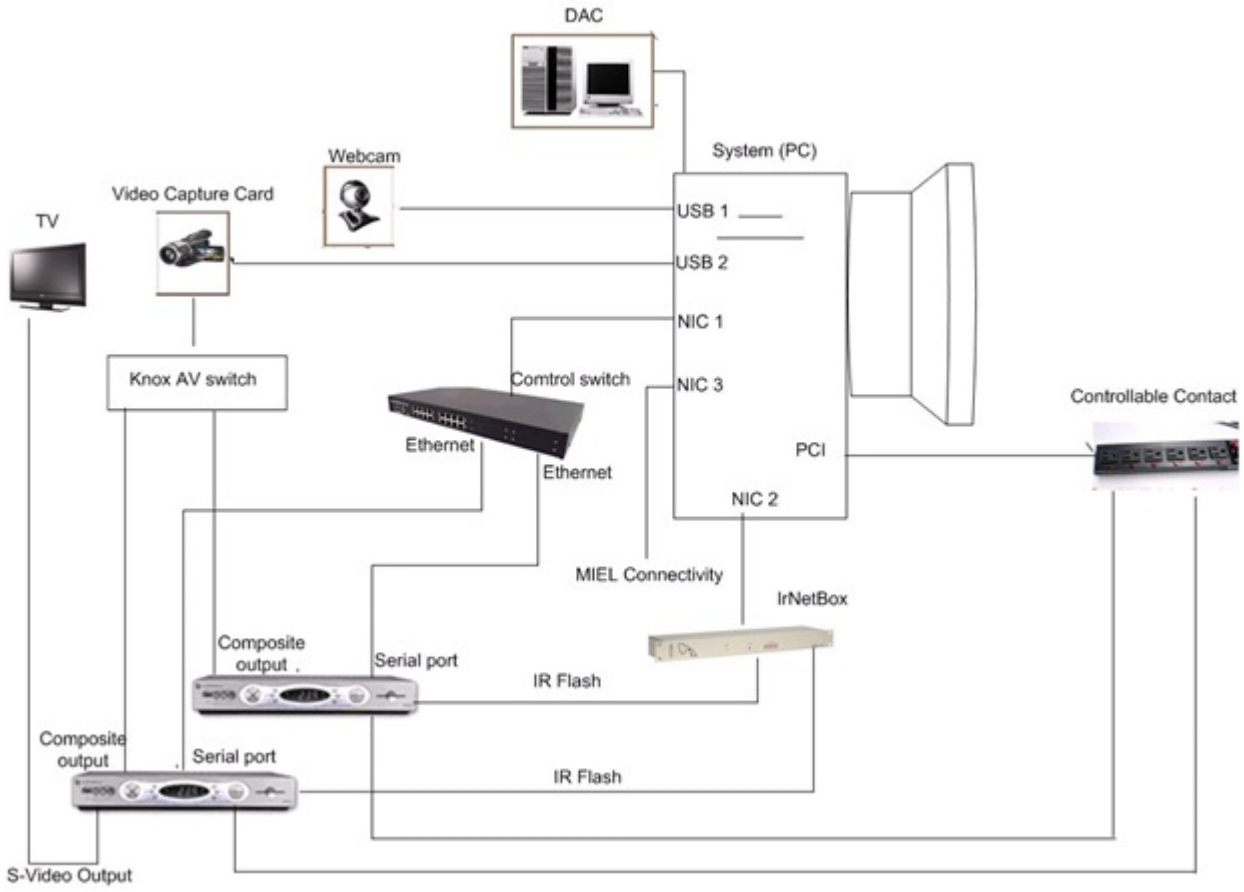

Figure 2.1: Architecture of Automation System with Other External Devices

Output device used to capture video and images from the STB.

STB (Set Top Box)

The device under test

Knox Switch

Switch that is used to connect multiple STB's to capture the composite output.

IrNetBox

A switch that is used for transferring IR commands from the System PC.

Comtrol Switch

A switch used to communicate between the STB using serial port and Ethernet port of System PC.

2.2 Automation Framework

The current automation framework in use is as shown in figure 2.2

It is a complicated test-framework, consisted of several components:

- **Configuration component** - is used for logical and physical devices setup
- **Test creator component** - is used for test designing and modification
- **Test session scheduler component** - is used for defining executing tests set, execution date and time, and set of testing devices

- **Background service component** - monitors the list of scheduled test-sessions and drives corresponding session from the queue to run
- **Driver engine monitor component** - allows user to observe the current status of all scheduled test-sessions in the real-time mode
- **Verification system** - is used for gathered data review and verification.

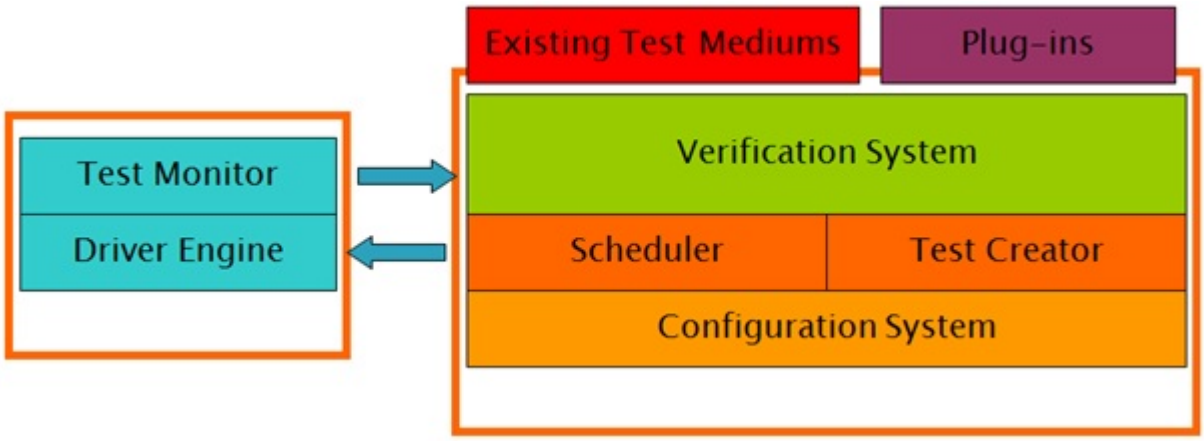

Figure 2.2: Current Automation Framework

Chapter 3

Problem in the current environment

The current applications used for automated testing mainly focus on simulating the test cases on the systems. They are not able to handle the errors arising due to system or network failures. This results into the failure of the applications and the testing time is increased along with the cost.

There are situations where the test cases are scheduled to be running on the weekend. In these cases if the pc crashes or due to any network or external device issue, the application fails, then the whole weekend is spoiled and the test engineers have to do the test cases again. This can heavily impact the organization as the deployment time will be delayed as the test cases have to be scheduled again.

3.1 Types of Failures

- **Application fails at night** It may happen due to any of the hardware or the newtork issues, that the application may fail at midnight. This will lead to the

scheduled test cases to stop. Hence the next day the test engineer comes and restarts the application. This may decrease the cycle time savings that should be acheived with Automation.

- **System Crash** The system may crash due to low memory or low virtual memory. This will also affect the cycle time as the test engineer has to restart the pc during such cases or may require reparing of the system.

- **Network Error** There may arise some issue in the network. All the scripts are stored in a database in a server, and loaded when they are used. If any network issue happens, the execution will fail, and the test engineer has to reschedule the remaining test cases again.

- **HDD Failure** During execution we are caputring video and images from the STB, which contains both SD and HD. Both the video types consume lot of memory. It may happen that during a overnight execution the HDD may get full and may lead to the failure of the application.

- **External Device Failure** Similarly it may happen that the external devices attached to the system might fail. Hence they need to be refreshed and intialized again before they could be used.

3.2 Statistics

Figure 3.1 shows the comparision of manual testing with automated testing with failures.

Testing Type	STB Type	Product Area	Number of Test Cases	Stall Time (Hours) / Manual Effort Required	Cycle Time (Hours)	Cumulative Time (Hours)
Manual	Delmar	iPG	105	42	50	92
		DVR	69	27.6	35	62.6
	Cable	VOD	83	26.56	35	61.56
		EPG	36	11.52	20	31.52
Automated With Failures	Delmar	iPG	105	24	40	64
		DVR	69	7.56	20	27.56
	Cable	VOD	83	10	18	28
		EPG	36	16	12	28

Figure 3.1: Statistics of Manual vs Automation

3.3 Previous Work

Application specific plug-in had been developed. The plug-in was attached to the application. It could monitor the test cases executing in the environment and was able to report the errors back to the user. No corrective measures could be taken by the plug-in.

The plug-in developed was tightly coupled with the application, hence if the application failed the plug-in also failed.

Other monitoring softwares Softwares available in market are only for monitoring purposes. They cannot handle the failures of the automation application and the automation environment.

Chapter 4

Proposal

4.1 A Software Framework

A software framework is a universal, reusable software platform used to develop applications, products and solutions. Software Frameworks include support programs, compilers, code libraries, an application programming interface (API) and tool sets that bring together all the different components to enable development of a project or solution.

Software Frameworks are designed to facilitate the development process by allowing designers and programmers to spend more time on meeting software requirements rather than dealing with the more tedious details of providing a working system. Software frameworks allow developers to spend less time coding, less time developing and debugging and more time on value-added development and concentrating on the business-specific problem at hand rather than on the plumbing code behind it resulting, faster time to market.

Software frameworks are used to develop device-to-enterprise applications, Internet-enabled products and automation system solutions.

An application framework needs to be developed which can help the applications running on top of it, without worrying about the system or network errors.
The architecture of the framework is shown below

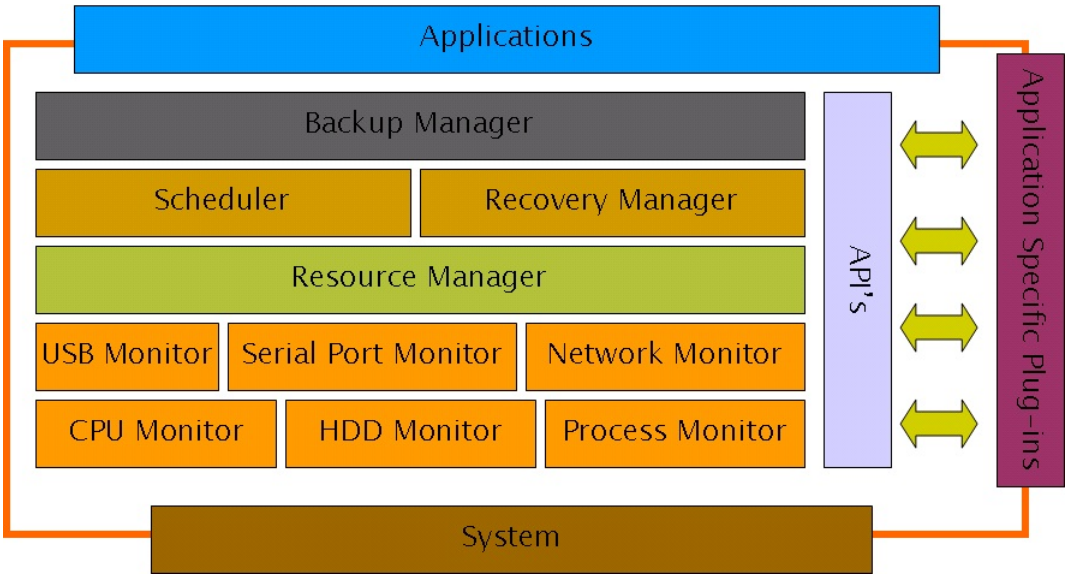

Figure 4.1: Proposed architecture

System - PC on which automated testing applications run
Monitors - Independent Modules

Resource Manager - Resource Manager Keeps a track of all the available resources and which process is currently using it. It also helps in proper allocation of resources to every running process.

Recovery Manager - It gets the data from the monitors and keeps on track on all the active processes. When any process fails, it takes the corrective action like restarting the process and then schedules it with the help of the scheduler. It also restarts the execution of the automated test cases from where it stopped.

Scheduler - It assists the recovery manager by scheduling the processes on the

Feature	Description
CPU Monitor	This module monitors the current CPU utilization and returns % of CPU utilized at any given time.
RAM Monitor	This module monitors the current RAM utilization and returns the used and the available memory at any given time.
Process Monitor	This module monitors the processes currently running on the system. It also takes the corrective action on the process (like pause, restart and kill) so that the process does not hinder the other processes.
HDD Monitor	This module monitors the HDD and returns the space available and the free space.
Ethernet Monitor	This module monitors the Ethernet links and returns the status of each Ethernet port.
Serial Port Monitor	This module monitors all the serial ports attached to the system.
USB Monitor	This module monitors the USB devices attached to the system.

Figure 4.2: Description of Monitors

system. It is capable of starting, stopping and restarting the processes.

Backup Manager - It periodically takes the backup of the executed test cases so that in case of failures the execution can be rolled up to the last consistent state.

Applications - Independent automated test solutions developed to run over the framework.

API's - These are the application interfaces exposed to the developers to interact with the framework.

Application Specific Plug-in - These use the exposed API's and are developed to perform application specific tasks.

Chapter 5

Challenges

- Determining the frequency at which the monitor should get the information about the system and the devices.
- Determining the threshold values for each monitor, i.e. at which point the monitor should report to the upper layer, so that corrective actions can be taken.
- Controlling the threads - Each monitor is working on a separate thread; hence the threads must be running without affecting the work of the other thread.
- Prevent Starvation - The framework monitors the system, processes and the external devices. The framework should be such that it does not use much of the CPU cycles, hindering other important processes.
- Recover and Restart - Once the application failed, the application needs to be restarted and recovered to the consistent state before which it failed.
- Rescheduling - Once we have recognized that the application has failed, the next question arises how do you schedule the failed test cases. Either stop the current on going execution and re run it or re-schedule the failed test cases once all the test cases have completed

- Monitoring the external devices attached and resetting them when required.
- Backup - We need to take the backup of the test cases so that upon failure, they could be re-run from the same place where the application failed.
- Generic - The framework should not be restricted for only kind of automation environment, it should be generic enough to handle other automation environment with ease.
- API's - Exposing well defined API's so that new programmers can build their own logic and add it to the framework.
- Remote configuration - Provide a mechanism of monitoring and recovering the failure of the application from remote system.
- Retrying - How many times to retry, after a failure has happened is a challenging task.

Chapter 6

Implementation

- **All monitors implemented for remote as well as local pc**
  The monitors are interconnected to the system and the devices as shown below in figure 7.1 and figure 7.2.

- **API related to monitors are exposed**
  All the necessary API for the monitoring purpose like get the information from CPU, HDD status, Process Status have been exposed for the programmers to use and implement according to their need. The API information can also be found in the Help file shipped with the software.

- **Recovery Manager, Resource Manager and Scheduler**

  - Common issues like HDD failure due to lack of space in the PC have been handled by transferring the stored content to some other location on the fly. Also if necessary the application will be paused for proper recovery to take place.
  - Power failure issues have been handled by scheduling the complete test suite again.
  - The monitor threads are made such that they do not starve any other processes, using proper sleep and wake-up mechanisms.

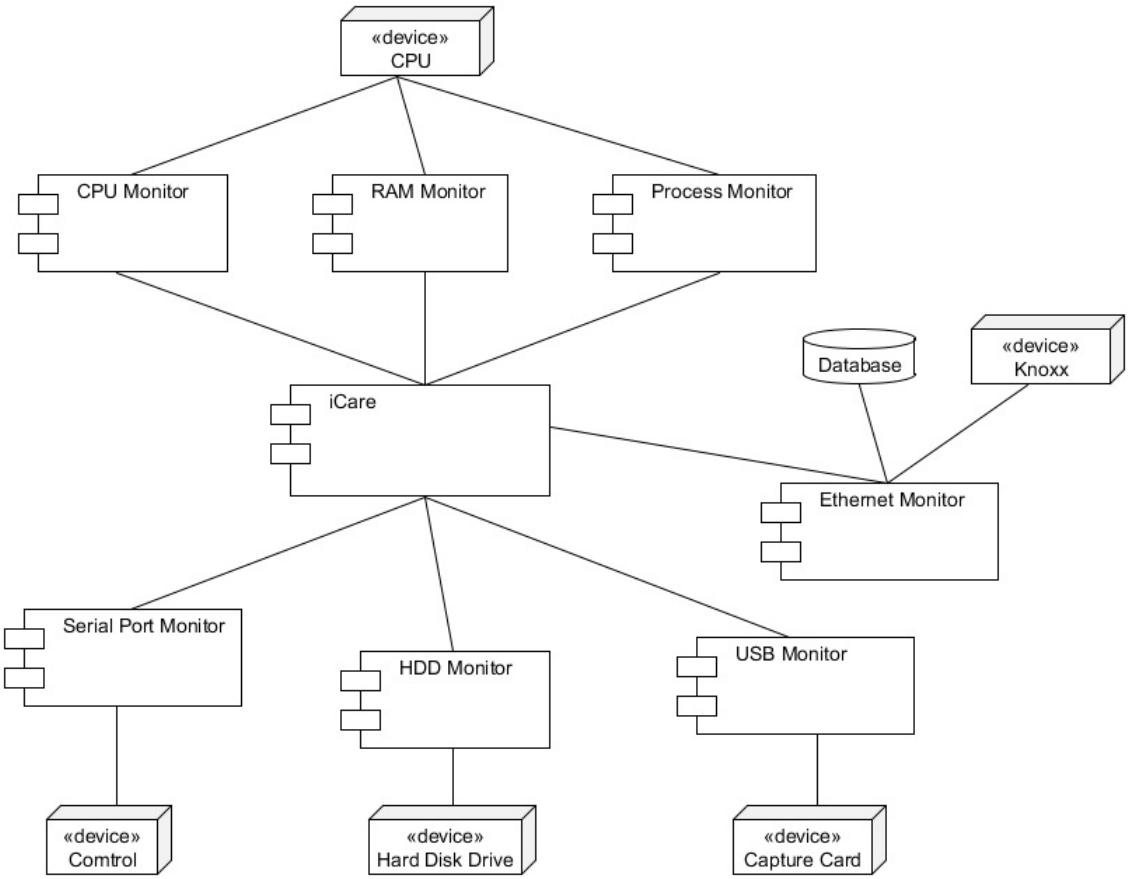

Figure 6.1: Local System View of Monitors

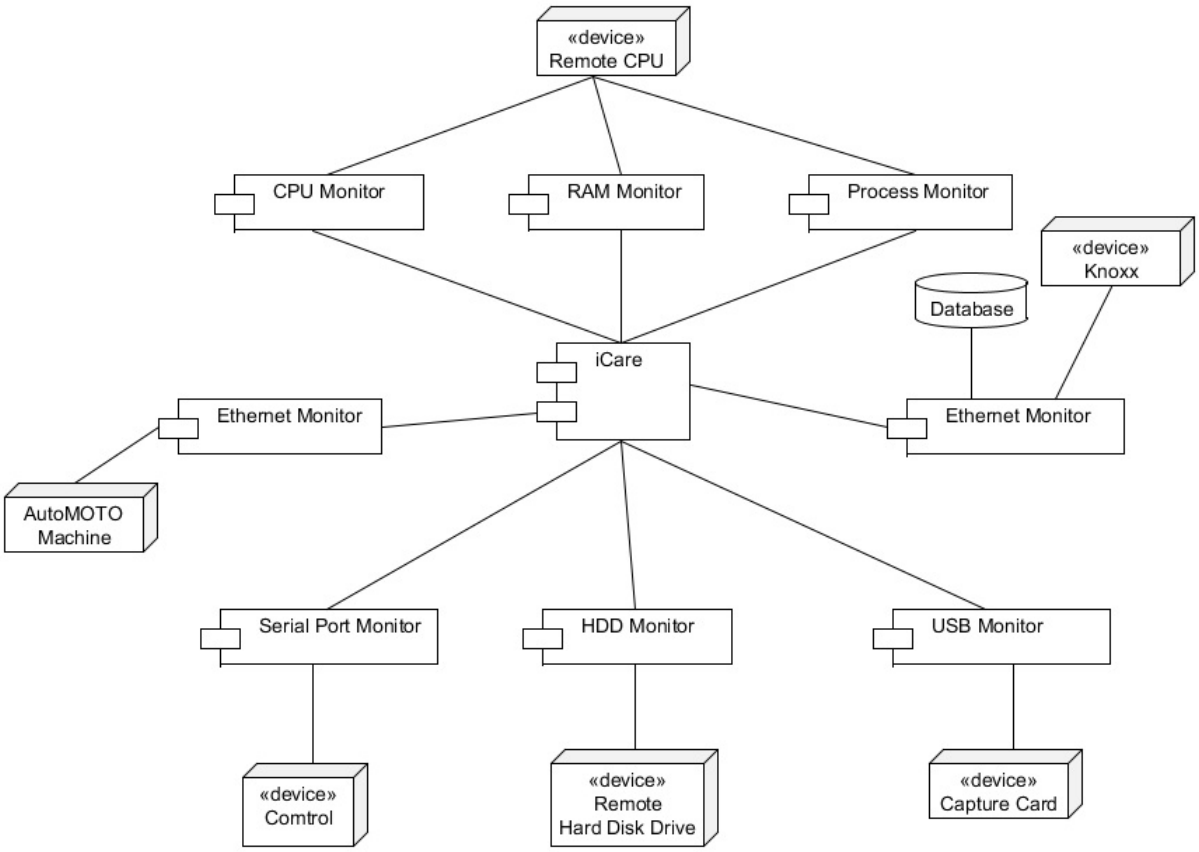

Figure 6.2: Remote System View of Monitors

- The threshold values of the monitors are calculated relatively to the system's resources during the run time. Hence the monitors can be used on any PC without any modifications.
- Scheduler is able to schedule the failed test cases successfully.

Chapter 7

Enhancement

Automated Exploratory Testing is being currently implemented as an enhancement over the existing software.

7.1 Exploratory Testing

Exploratory testing is an approach that does not rely on the documentation of test cases prior to test execution. This approach has been acknowledged in software testing books since the 1970's [7]. However, authors have usually not presented actual techniques or methods for performing exploratory testing; instead treating it as an 'ad hoc' or error guessing method.

Exploratory testing is an approach to software testing that is concisely described as simultaneous learning, test design and test execution. Cem Kaner, who coined the term in 1983, now defines exploratory testing as "a style of software testing that emphasizes the personal freedom and responsibility of the individual tester to continually optimize the quality of his/her work by treating test-related learning, test design, test execution, and test result interpretation as mutually supportive activities that run in parallel throughout the project."

While the software is being tested, the tester learns things that together with experience and creativity generates new good tests to run. Exploratory testing is often thought of as a black box testing technique. Instead, those who have studied it consider it a test approach that can be applied to any test technique, at any stage in the development process.

7.2 Motivation

Exploratory testing lacks scientific research [9]. While test case design techniques set the theoretical principles for testing, it is too straightforward to ignore all the factors that can affect testing activities during test execution work.

In the context of verifying executable specifications Houdek et al. [8] have performed a student experiment comparing reviews, systematic testing techniques and the exploratory (ad-hoc) testing approach. The results showed that the exploratory approach required less effort, and there was no difference between the techniques with respect to defect detection effectiveness. None of the studied techniques alone revealed a majority of the defects and only 44 percent of the defects were such that the same defect was found by more than one technique.

Some research on exploratory testing can be found in end-user programming context. Rothemel et al. [11] reported benefits of supporting exploratory testing tasks by a tool that is based on formal test adequacy criteria. Phalgune et al. have found that oracle mistakes are common and should be taken into account in tools supporting end-user programmer testing [10]. Oracle mistakes, meaning that a tester judges incorrect behavior correct or vice versa, could be an important factor affecting the effectiveness of exploratory testing and should be studied also in the professional software development context.

Even though the efficiency and applicability of exploratory testing lacks reliable research, there are anecdotal reports listing many benefits of this type of testing. The claimed benefits, summarized in [9] include, e.g., effectiveness, the ability to utilize tester's creativity and non-reliance on documentation [12,13]. Considering the claimed benefits of exploratory testing and its popularity in industry, the approach seems to deserve more research. The exploratory approach lets the tester freely explore without being restricted by pre-designed test cases. The aspects that are proposed to make exploratory testing so effective are the experience, creativity, and personal skills of the tester. These aspects affect the results, and some amount of exploratory searching and learning exists, in all manual testing; perhaps excluding the most rigorous and controlled laboratory settings. Since the effects of exploratory approach and the strength of those effects have not been studied and are not known, it is hard to draw strong conclusions on the performance of manual testing techniques.

We recognize that planning and designing test cases can provide many other benefits besides defect detection effectiveness. These include, e.g., benefits for test planning, test coverage, repeatability, and tracking.

Exploratory Testing Can be used when

- You need to provide rapid feedback on a new product or feature.
- You need to learn the product quickly.
- You have already tested using scripts, and seek to diversify the testing.
- You want to find the single most important bug in the shortest time.
- You want to check the work of another tester by doing a brief independent investigation.
- You want to investigate and isolate a particular defect.

- You want to investigate the status of a particular risk, in order to evaluate the need for scripted tests in that area.
- Improvising on scripted tests.
- Interpreting vague test instructions.
- Product analysis and test planning.
- Improving existing tests.
- Writing new test scripts.
- Regression testing based on old bug reports.
- Testing based on reading the user manual and checking each assertion.

7.3 Pros and Cons

Pros

- Checks application usability
  - This of course depends on how the tester is sensitive to usability. Usually, however, something that is understandable for the developer and the robot, it is not obvious to the tester. Because exploratory tests are treated as a whole, it is easier to see the wrong assumptions in cross-section of multiple modules or functions.
- Helpful with a lack of documentation, requirements, test cases, etc
  - There are situations when we get application to test without any documentation, test cases or automation scripts. Natural way is, of course, writing the test scenarios first, however, I recommend to pre-order exploratory tests. Soon we will have the information about the quality level of the

product - our benchmark. Testers will have a good basis to start writing the missing parts. So this kind of testing allows you to get temporary salvation and able to maintain the expected quality in short term.

- Find holes in requirements
  - Exploratory tester usually report many errors caused by wrong requirements or documentation. What is interesting is such errors are usually reported as critical. As mentioned earlier, deduction across whole applications have no small importance here. Exploratory testing can upset upside down some of the assumptions.

Cons

- We do not need test scenarios, unit tests, test automation, etc.! We have skilled testers!
  - The situation is often encountered in smaller firms where there is no dedicated quality assurance teams. Tester is a programmer, a tester is a business owner, a secretary, all strive to perform exploratory tests. At the end of their testing, it becomes a set of performed routinely activities. For larger companies, we also deal with the syndrome, "Our application is free of errors!".
- Poor detection of minor issues
  - Exploratory tester is focused on finding gaps in mainstream business process covered by tested application. But of course there are also deviations in the other side, in example : focusing on the type of error - whether it is possible to enter a negative value in the field, which should has only positive values. Fields validation is a role of automated or unit testing.
- Exploratory testers can get into a routine

  - Described methodology is based on the deduction, which degrades when is constantly exposed to the same experience. Tester which is extensively used in this way, often becomes an automation robot that uses a memorized test script. As team leaders we have seen this phenomenon in advance. The easiest way to counteract this situation is to apply the testers and test applications rotation.

Thus exploratory testing is a "must have" methodology in testing process. In larger teams it may be a group of dedicated testers, the smaller one or two people with extensive experience.

7.4 Research Questions

We study the effects of using predesigned test cases in manual functional testing at the system level. Due to the scarce existing knowledge, we focus on one research problem: What is the effect of using predesigned and documented test cases in manual functional testing with respect to defect detection performance?

Based on existing knowledge we can pose two alternative hypotheses. First, because almost all research is focused on test case design issues we could hypothesise that the results are better when using predesigned test cases. Second, from the practitioner reports and case studies on exploratory testing we could draw a hypothesis that results are better when testing without predesigned test cases. The research questions and the hypotheses of this study are presented below.

- Research question 1: How does using predesigned test cases affect the number of detected defects?
  - Hypothesis H10: There is no difference in the number of detected defects between testing with and without predesigned test cases.
  - Hypothesis H11: More defects are detected with predesigned test cases than without predesigned test cases.

  - Hypothesis H12: More defects are detected without predesigned test cases than with predesigned test cases.

- Research question 2: How does using predesigned test cases affect the type of defects found?

  - Hypothesis H20: There is no difference in the type of the detected defects between testing with and without predesigned test cases.

- Research question 3: How does using predesigned test cases affect the number of false defect reports?

  - Hypothesis H30: There is no difference in the number of produced false defect reports between testing with and without predesigned test cases.

False defect reports refer to reported defects that cannot be understood, are duplicates, or report nonexisting defects. This metric is used to analyze how using test cases affects the quality of test results.

7.5 Rule Based Exploratory Testing - A Custom Approach

The first step to combining these two methods is to record the interactions performed by a human tester during an exploratory test session as a replay able script. This can be accomplished using a capture/replay tool (CRT) - a tool that records interactions with an application as a script and can later replay these actions as an automated test. Next, a human tester defines a set of rules that can be used to define the expected (or forbidden) behavior of the application. The rules and script are then combined into an automated regression test which increases the state space of the system that is tested. This approach allows a human tester to use exploratory tests to identify regions of the state space of the system that need to be subjected to more rigorous

rule-based testing, which, in effect, identifies an important subset of the system under test on which testing should focus. At the same time, this subset is tested thoroughly using automated rules in order to verify this subset more thoroughly than would be possible with exploratory testing alone.

It has been suggested that manual exploratory testing could benefit from the addition of automated support. In light of this, we propose that manual exploratory test sessions be recorded in a replay able format, and then enhanced with short, automated rules that increase the amount of verification performed when that test is replayed. In this way, only a subset of the state space of the application under test in which a human has expressed interest will receive additional automated scrutiny. The additional verifications provided by these rules will increase the parts of the state space that are tested, but only in that same subset identified by the human tester. In this way, we aim to create strong, relevant tests by relying on the repeatability and verification ability of rule-based testing as well as the intelligence of human testers. The fact that rules contain preconditions also makes it less likely that rule-based tests will falsely report failures when the STB environment changes - instead, they will simply not fire when they are not applicable.

We enhanced the tool, to enable us to test out the concept of R-BET. The overall structure of enhanced feature is shown in Figure 8.1.

Next, the tool can be used to create rules - short verifications that will interact with a system to ensure that a specific property is always (or never) true. Each rule takes the form of an "if try catch" statement. The "if," or precondition, of a rule makes sure that it will only fire under certain circumstances.

For instance, if a rule should only ensure that the box is in standby mode, a precondition for this rule might be "if the STB is in standby." If a rule has multiple preconditions, then all of these must be met before a rule will fire, because the pre-

conditions are connected with a logical AND. The same precondition can be used by more than one rule. The "try," or action, represents the main body of the rule, and will be executed when all preconditions are met. In the previous example, the action might be "assert that the STB is in standby mode." An action can be as simple as a verification of a single property or as complex as a series of interactions with the application under test. The "catch," or consequence, determines what should happen if the action fails or throws an exception. This allows test authors to distinguish between failures that indicate bugs and warnings that represent that a coding standard has not been met. In the previous example, it might not be necessary to fail the entire test if the STB was in on state, but it might be helpful to log this warning so that developers will be made aware of its existence.

The rules are combined with recorded exploratory tests as a TestRunner - an program that combines a recorded exploratory test with a set of rules. A TestRunner runs a test script one step at a time and checks each rule against the system under test after each of step. In this way, testers are able to define rules that will help explore the state space of the application under test more thoroughly. In the example used in the previous paragraph, a rule can be defined that will check that each STB in the rack is not both enabled and in standby mode. This could be checked manually by a human tester, but it would be a tedious task. Creating a rule to test for this behavior will reduce the amount of work that must be done by human testers at the same time as increasing the number of different states from which this verification can be performed. Additionally, rules can be defined to test for typical errors that a human tester may overlook, may not have time to test for, may not be experienced enough to know about. Automated, rule-based verification not only allows a system to be tested more thoroughly than would otherwise be possible within a given timeframe, but also frees up human testers to perform more interesting testing.

Structure of a rule designed to detect standby mode of STB that should not respond

to simulated user input is shown below in figure 8.2

The below figure shows the implementation of generating manual exploratory scenarios. However, these generated scenarios will run in automated way. We have favorites tab on the top, which has scripts to perform one set of complete action. We have the remote control image map on the right side, which allows user to perform all the actions, which he can do using a STB remote. The test engineer can create his exploratory scenarios, run it and save it for further use. Upon successful execution of the scenario, the tool will give graphical results, showing which steps failed/passed, along with a personal mail to the test engineer.

The below figure shows the automation framework screen, which shows that the exploratory scenario is being run for the particular STB. User can pause, cancel and stop the ongoing execution if needed.

Once the execution is finished, the detailed description of the exploratory scenario can be found in the automation framework as shown below in figure 8.5
And the summary of the results is shown in the exploratory framework, as shown in figure 8.6

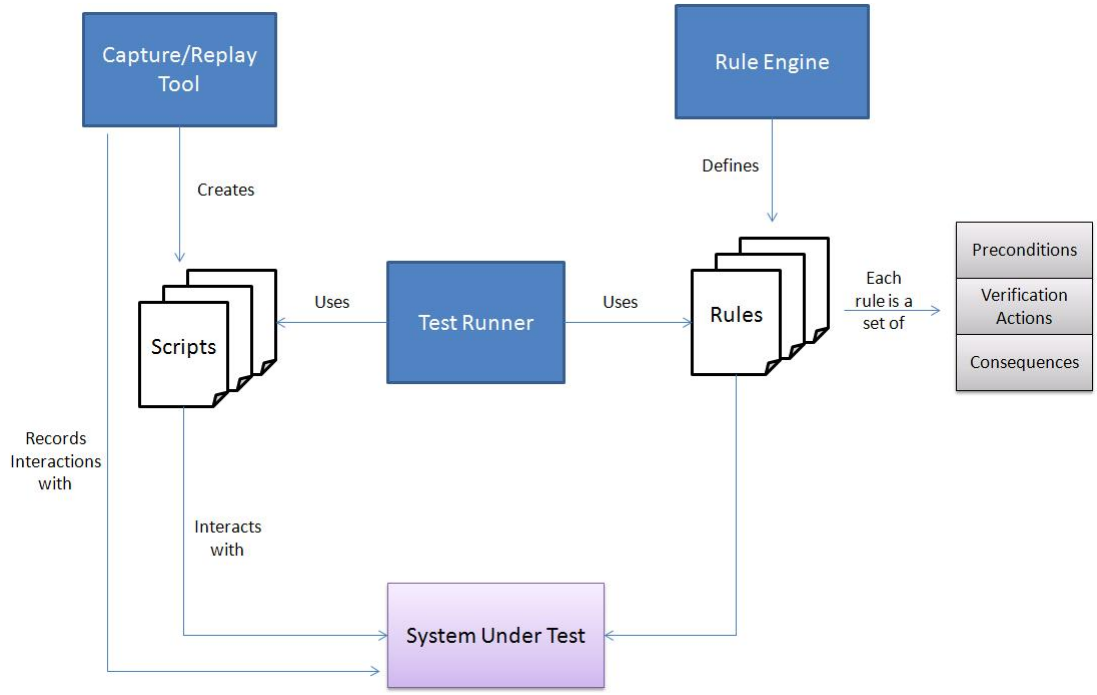

Figure 7.1: Architecture of Custom Rule Based Exploratory Testing

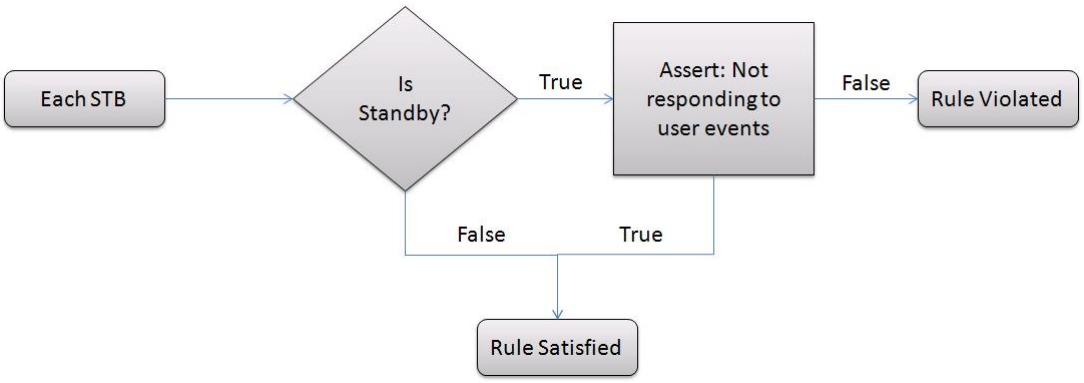

Figure 7.2: Sample Structure of a Rule

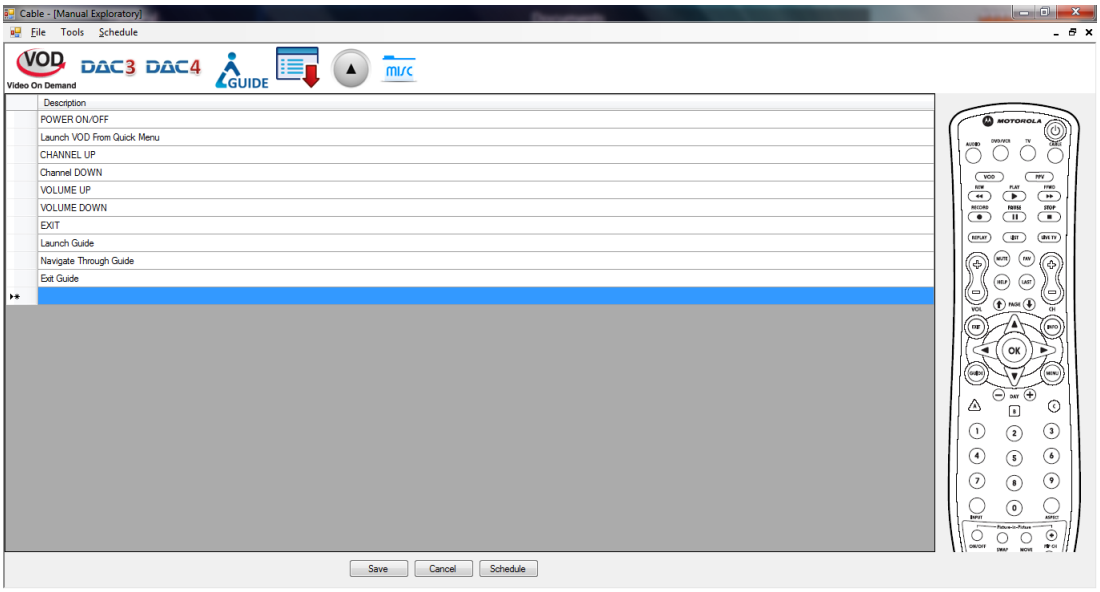

Figure 7.3: Sample Manual Exploratory Scenario

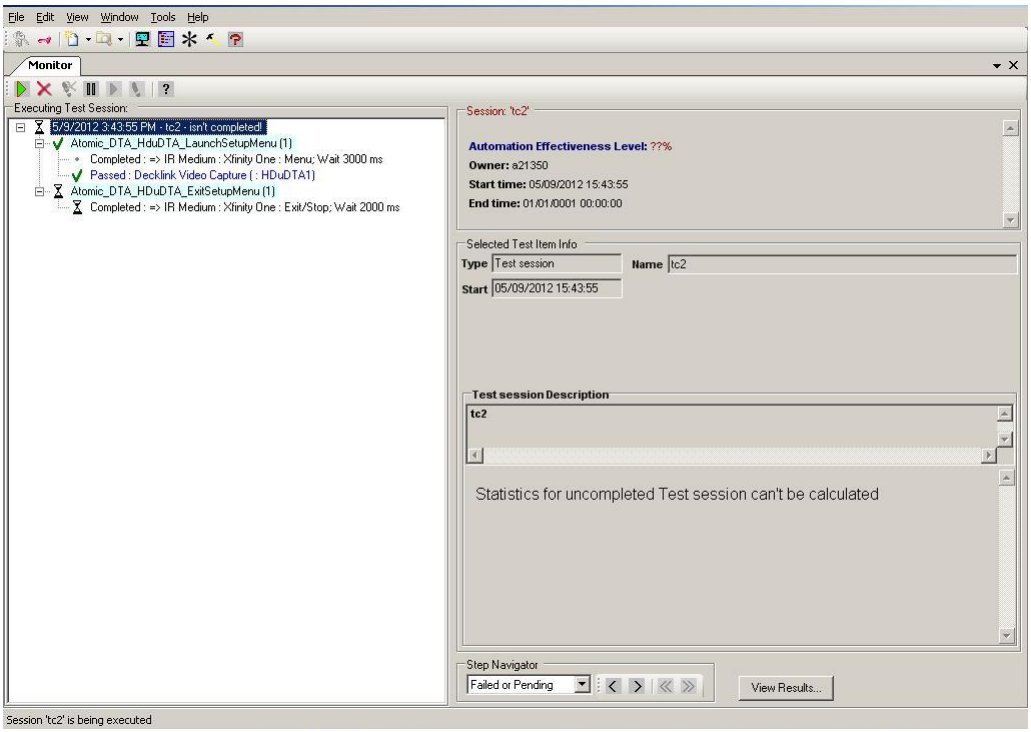

Figure 7.4: Sample Manual Exploratory Scenario Execution

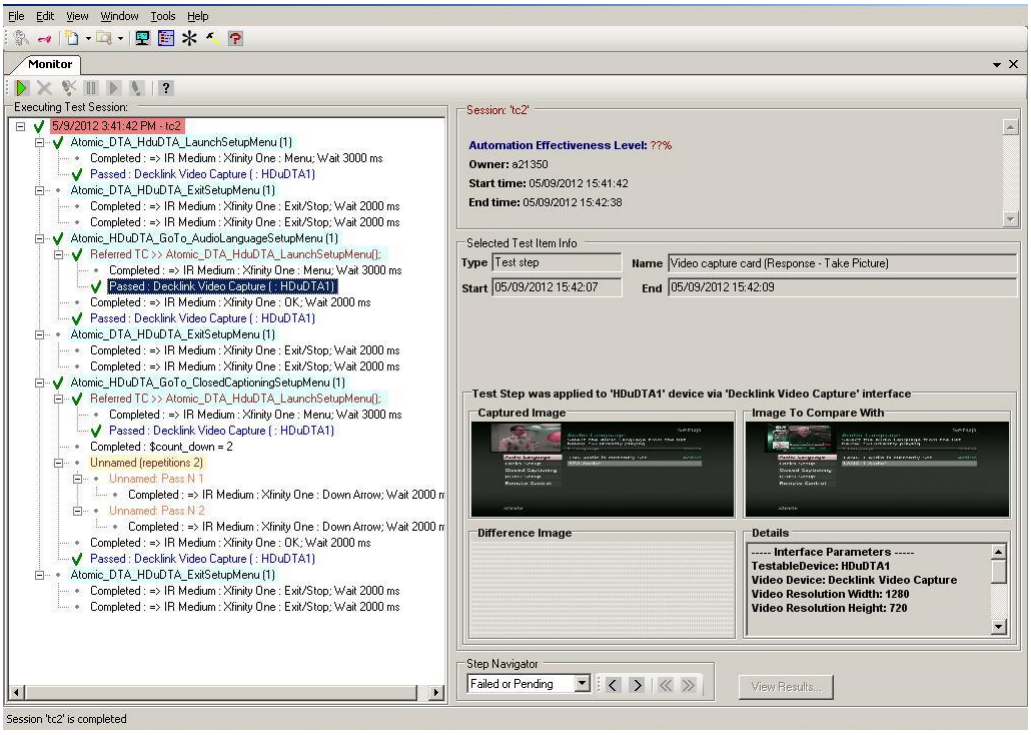

Figure 7.5: Sample Manual Exploratory Scenario Results Screen

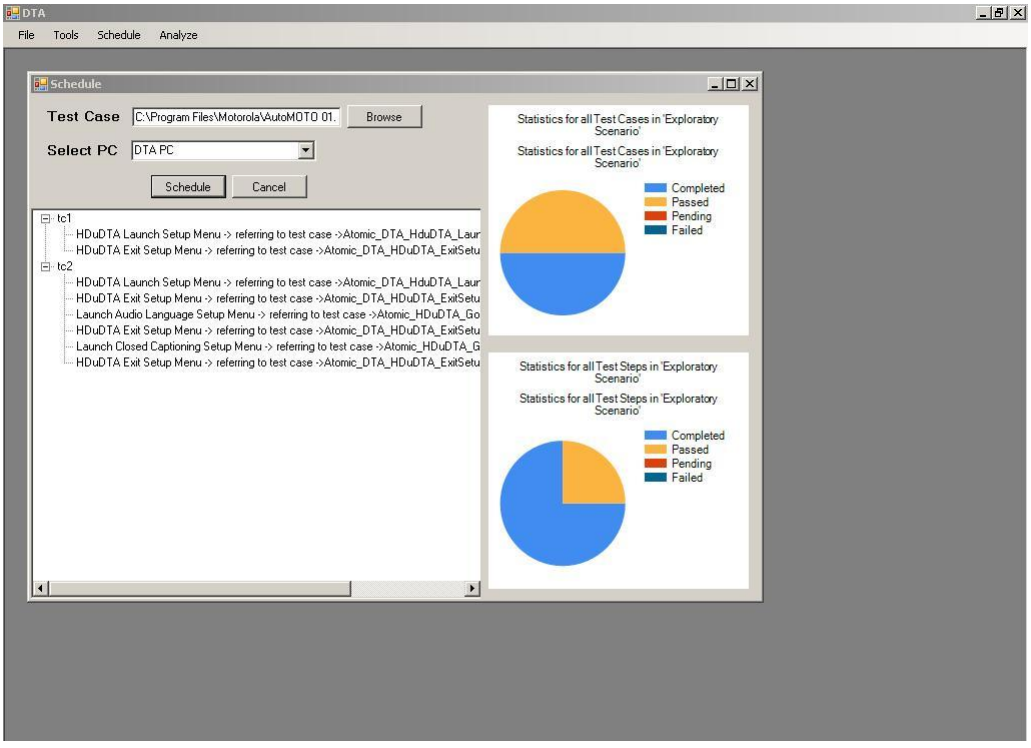

Figure 7.6: Sample Manual Exploratory Scenario Summary

7.6 Research Questions - Answers

- Research question 1. How does using predesigned test cases affect the number of detected defects?
  - In this experiment, the subjects found less defects when using predesigned test cases. The difference between the two approaches was not statistically significant, and does not allow rejecting the null hypothesis that assumes there is no difference in the number of detected defects when testing with or without test cases. Although we cannot reject the null hypothesis, the results strengthen the hypotheses of the possible benefits of exploratory testing.
- Research question 2. How does using predesigned test cases affect the type of found defects?
  - We analyzed the differences in the types of the detected defects from three viewpoints; severity, type, and detection difficulty. Based on the data, we can conclude that testers seem to find more of both the most obvious defects, as well as the ones most difficult to detect when testing without test cases. In the terms of defect type, the testers found more user interface defects and usability problems without test cases. More technical defects were found using test cases. When considering defect severity, the data shows that more low severity defects were found without test cases. The statistical significance of the differences in all these defect characterizations is low. We must be cautious of drawing strong conclusions based on the defect classification data even though the results show a significant difference in the numbers of usability and minor defects detected between the two approaches.
  - The differences in the defect types and severities suggest that testing without test cases tend to produce larger amounts of defects that are obvious to

detect and related to user interface and usability issues. These differences could be explained by the fact that test cases are typically not written to test obvious features and writing good test cases for testing many details of a graphical user interface is very laborious and challenging. On the other hand, subjects testing without test cases found more defects that were difficult to detect, which supports the claims that exploratory testing makes better use of tester's creativity and skills during test execution. The higher amount of low severity defects detected without test cases suggests that predesigned test cases guide the tester to pay attention on more focused areas and thus lead to ignoring some of the minor issues.

- Research question 3. How does using predesigned test cases affect the number of false defect reports?
  - The purpose of this research question was to provide an understanding on the effects of the two approaches from the test reporting quality viewpoint. The data shows that testers reported around twice as many, false defect reports when testing with test cases than when testing without test cases. This difference is statistically significant. This issue raises the more general question of the consequences of following predesigned test cases in manual test execution. Test cases are used to guide the work of the tester and more studies are needed to better understand how different ways of documenting tests and guiding testers' work affect their behavior in performing the tests and the results of testing efforts.

Chapter 8

Conclusion

The below figure, shows that using the above mentioned mechanism, the current automation environment has been optimized to perform in a better way. The data was captured by executing the same test cases in all the three forms, i.e. Manually, Automated Without Our Tool in Use and Automated With Our Tool in Use.

Testing Type	STB Type	Product Area	Number of Test Cases	Staff Time (Hours) / Manual Effort Required	Cycle Time (Hours)	Cumulative Time (Hours)
Manual	Delmar	IPG	105	42	50	92
		DVR	69	27.6	35	62.6
	Cable	VOD	83	26.56	35	61.56
		EPG	36	11.52	20	31.52
Automated With Failures	Delmar	IPG	105	24	40	64
		DVR	69	7.56	20	27.56
	Cable	VOD	83	10	18	28
		EPG	36	16	12	28
Automated w/o Failures	Delmar	IPG	105	12	20	32
		DVR	69	3.78	10	13.78
	Cable	VOD	83	5	9	14
		EPG	36	8	6	14

Figure 8.1: Comparison Between Existing And New Environment

Testing Type	Product	No. of Test Cases	Staff Time(Hours)/ Manual Effort Required	Cycle Time(Hours)	Cumulative Time(Hours)
Manual	DVR	80	64	42	106
	MRDVR	60	52	36	88
Automated with Old Environment	DVR	80	36	40	76
	MRDVR	60	32	36	68
Automated with New Environment	DVR	80	24	28	52
	MRDVR	60	22	24	46

Figure 8.2: Comparison Between Existing And New Environment

Performance in the form of Time Savings of the new environment vs the existing envrionment is as shown below.

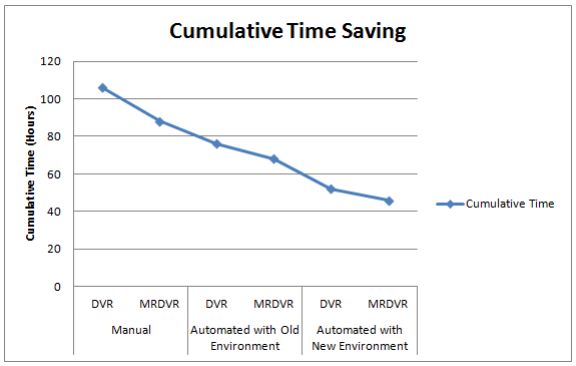

Figure 8.3: Cumulative Time Saving in the New Environment

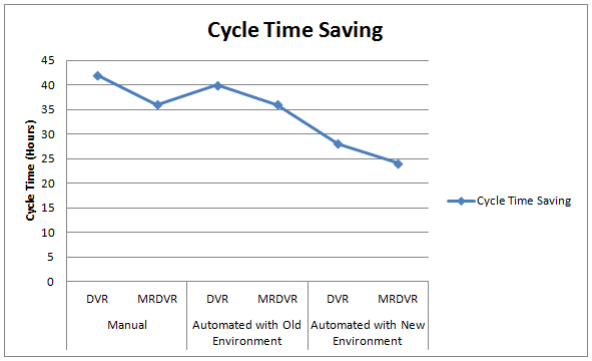

Figure 8.4: Cycle Time Saving in the New Environment

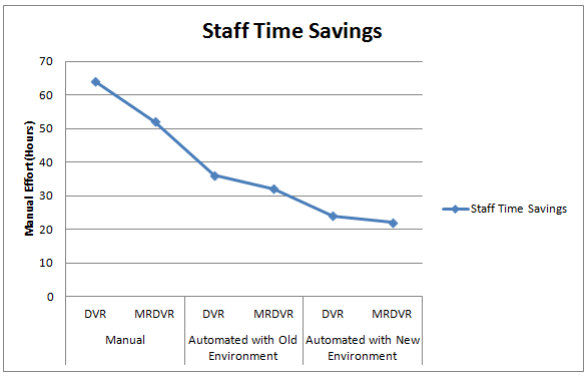

Figure 8.5: Staff Time Saving in the New Environment

The below figure shows that the efficiency of the framework is more than 80 percent.

Box Type	No of times Issue Observerd (Per Cycle)	No of time issue recovered (Per Cycle)	Gain in terms of Hours (Per Cycle)
DTA	5	4	2
HuDTA	7	5	2
Cable	12	10	6
Irvine	9	6	5
Glennover	13	11	6

Figure 8.6: Efficiency of Framework

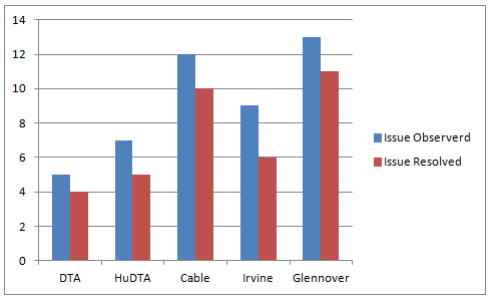

Figure 8.7: Performance of Framework in the New Environment

Thus, the implemented solution is able to reduce the cycle time, and thus increasing the productivity along with increasing the stability of the automation environment for STB's.

Chapter 9

Future Implementation

- Automatic validation of the test cases which are generated using Automatic Generation Feature.
  - In the implemented solution, once the exploratory scenario is generated, the test engineer has to verify the generated scenario. In special cases like, stability and stress testing, this verification is also not required.
  - Currently there is no way to deduce that whether the automatically generated exploratory scenarios are valid or not. This can be implemented using a machine learning mechanism, where the tool can understand the pattern in which test engineer creates manual exploratory scenario's and then use the learning in creating its own valid exploratory scenario.

References

[1] Software Testing by Jiantao Pan, Carnegie Mellon University

[2] Exploratory Testing Explained, James Bach

[3] Black, Rex; (2002). Managing the Testing Process (2nd ed.). Wiley Publishing. ISBN 0-471-22398-0

[4] Digital CM. http://digitalcm.corp.mot.com/default.asp

[5] Compass. http://compass.mot-mobility.com

[6] Myers, G. J., The Art of Software Testing, John Wiley and Sons, New York, 1979.

[7] Houdek, F., T. Schwinn, and D. Ernst, "Defect Detection for Executable Specifications - An Experiment", IJSEKE, vol. 12(6), 2002, pp. 637-655.

[8] Itkonen, J. and K. Rautiainen, "Exploratory Testing: A Multiple Case Study", in Proceedings of ISESE, 2005, pp.84-93.

[9] Phalgune, A., C. Kissinger, M. M. Burnett, C. R. Cook, L. Beckwith, and J. R. Ruthruff, "Garbage In, Garbage Out? An Empirical Look at Oracle Mistakes by End-User Programmers", in IEEE Symposium on Visual Languages and Human-Centric Computing, 2005, pp. 45-52.

[10] Rothermel, K., C. R. Cook, M. M. Burnett, J. Schonfeld, Green, Thomas R. G., and G. Rothermel, "WYSIWYT Testing in the Spreadsheet Paradigm: An Empirical Evaluation", in Proceedings of ICSE, 2000, pp. 230-239.

[11] Bach,J., "Exploratory Testing", in The Testing Practitioner, Second ed., E. van Veenendaal Ed., Den Bosch: UTN Publishers, 2004, pp. 253-265.

[12] Bach, J., "Session-Based Test Management", STQE, vol.2, no. 6, 2000,

[13] Kaner, C., J. Bach and B. Pettichord, Lessons Learned in Software Testing, John Wiley and Sons, Inc., New York, 2002.

[14] Lyndsay, J. and N. van Eeden, "Adventures in Session- Based Testing", 2003, Accessed 2007 06/26, http://www.workroom-productions.com/papers/AiSBTv1.2.pdf.

[15] Vga,J. and S. Amland, "Managing High-Speed Web Testing", in Software Quality and Software Testing in Internet Times, D. Meyerhoff, B. Laibarra, van der Pouw Kraan,Rob and A. Wallet Eds., Berlin: Springer-Verlag, 2002, pp. 23-30.