Combinatorial Testing: Learnings from our Experience

R Krishnan, S Murali Krishna, P Siva Nandhan
Motorola Software Group-India, Motorola
krishnanr@motorola.com, a19032@motorola.com, a16581@motorola.com

Abstract

Combinatorial testing methods address generation of test cases for problems involving multiple parameters and combinations. The Orthogonal Array Based Testing Strategy (OATS) is one such combinatorial testing method, a systematic, statistical way of testing pair-wise interactions. It provides representative (uniformly distributed) coverage of all variable pair combinations. This makes the technique particularly useful for testing of software, wherever there is combinatorial explosion: a. In system testing for handling feature interactions b. In integration testing components c. It is also quite useful for testing products with a large number of configuration possibilities.

One of the fundamental assumptions behind OATS aprroach is that a subset covering all pair-wise combinations will be more effective than a randomly selected subset. OATS provides a means to select a minimal test set that guarantees testing the pair-wise combinations of all the selected variables. Covering pair-wise combinations has been reported to be very effective in the literature. Successful use of this technique, with 50% effort saving and improved testing with a factor of 2.6 is reported in the literature.

In this paper, we report on the in-house web-based application that we designed and implemented to support customized version of OATS and our experience in piloting and getting this method used in projects. In the in-house tool we have introduced a number of additional features, that help in generation and post processing of testcases. We have also designed a supporting process for using this method, and we discuss the steps in this process in the paper. We share details on application in feature testing of a mobile phone application. This method has also been successfully used in designing feature interaction test cases and for augmenting the regression suite to increase coverage.

Keywords: Orthogonal arrays, testcase generation, combinatorial testing

1. Introduction

In many test problems, the test space can be characterized as combinations of the values that different parameters take. Combinatorial testing methods can be used for addressing these test problems. The Orthogonal Array Based Testing Strategy (OATS) is one such combinatorial testing method, a systematic, statistical way of testing pair-wise interactions. It provides representative (uniformly distributed) coverage of all variable pair combinations. This makes the technique particularly useful for testing of software, wherever there is combinatorial explosion: a. In system testing for handling feature interactions b. In integration testing components c. It is also quite useful for testing products with a large number of configuration possibilities. Refer [4] for a collection of published orthogonal arrays

Interactions and integrations are a major source of defects. Based on our experience in testing mobile phone applications, defects by and large are not associated with complex interactions such as "*When the background is blue and the font is Arial and the layout has menus on the right and the images are large and it's a Thursday then the tables don't line up properly.*" Most of these defects arise from simple pair-wise interactions such as "*When the font is Arial and the menus are on the right the tables don't line up properly.*" With so many possible combinations of components or settings, it is easy to miss one. Randomly selecting values is bound to create inefficient test sets and test sets with arbitrary distribution of values and incomplete coverage. Ensuring coverage of pair-wise combinations (as in the OATS) is effective in bringing out all single-mode (where a value of a factor causes the error) and double-mode faults (where a combination of a factor-value pair causes the error).

Following is a sample application of this technique to handle combinations resulting from different configuration settings. taken from [5]. There are four factors/parameters namely: Web Browser, Web server OS, Type of Operation and Back-end Database. The different possible levels/values for these factors are listed below in Table-1. 100% coverage of pairs can be achieved in 9 testcases generated by OATS as listed in Table 2.

In section 2 we cover some reported use of this technique in the industry. In section 3 we describe our approach, and report details on a case study in section 4. In section 5 we share our learnings and in section 6 we conclude and share extensions that we may be trying out in future.

2. Reported Use of this Technique

Following is how one of the papers by Phadke [1] projects the quantified benefits of using OATS.

"In all, only 422 tests were needed to test the software. These tests identified 41 faults in the software, which were fixed, and the software was released. Two years of operation in the field generated no faults within the scope of testing, which indicated that relative to the scenarios

encountered, the test plan was 100 percent effective in identifying faults.

Web Browser	Web Server OS	Type of Operation	Back-end Database
IE 5.0™	Sun Os 7.0 ™	Store Data	Oracle 8i ™
Netscape 5.0 ™	Windows XP Pro ™	Retrieve Data	SQL Server 7.0 ™
Mozilla 1.3.1 ™		Delete Data	

Table 1: Sample Application

Web Browser	Web Server OS	Type of Operation	Back-end Database
IE 5.0 ™	Sun Os 7.0 ™	Retrieve Data	Oracle 8i ™
IE 5.0 ™	Windows XP Pro ™	Store Data	SQL Server 7.0 ™
IE 5.0 ™	Windows XP Pro ™	Delete Data	Oracle 8i ™
Netscape 5.0 ™	Windows XP Pro ™	Retrieve Data	SQL Server 7.0 ™
Netscape 5.0 ™	Sun Os 7.0 ™	Store Data	Oracle 8i ™
Netscape 5.0 ™	Sun Os 7.0 ™	Delete Data	SQL Server 7.0 ™
Mozilla 1.3.1 ™	Sun Os 7.0 ™	Retrieve Data	SQL Server 7.0 ™
Mozilla 1.3.1 ™	Windows XP Pro ™	Store Data	Oracle 8i ™
Mozilla 1.3.1 ™	Windows XP Pro ™	Delete Data	Oracle 8i ™

Table 2: Testcases covering all the pairs

Had AT&T run the alternate test plan that involved 1,000 tests, the lead tester estimated they may have found only 32 of the 41 faults. Compared to the original testing plan, Robust Testing required 50 percent less testing effort, identified 28 percent more faults, and was more productive (number of faults detected per tester week) by a factor of 2.6."

One of the major concerns expressed by teams in adopting this approach is the defect detection capability of this method. Establishing theoretical bounds for the defect detection capability of this method may not be possible. However, empirical studies have been done and reported [3]. Reported bugs from Open Source Mozilla Web browser and Apache web server have been analyzed and found that roughly 70% of the reported bugs could have been covered through testing pair-wise combinations. The study also showed that roughly 90% of the bugs get covered through testing three way combinations.

3. Our Approach

Orthogonal Arrays found out till date can be found in [4], but finding an appropriate orthogonal array for a given test problem may not be possible since an orthogonal array might not exist for that particular combination of factors (or parameters) and their levels (or values). One solution to this problem is to find an array that closely fits the requirements of the test problem. In this method, we were arriving at a pretty large number of test cases, since we had to choose an array, with factor-level pairs greater than or equal to the factor-level pairs of the problem.

This led us to develop an in-house tool OATSGen, to generate arrays such that all pairs were covered, relaxing the criterion that the pairs should be uniformly distributed. The tool takes the factors and their levels as input through an excel sheet. Additionally, constraints on the inputs can be fed in as well. The tool generates the test cases covering all the pairs in an additional work sheet as part of the excel sheet submitted as input.

The following greedy algorithm was used in the in-house tool:

- Sort the N factors in descending order of the number of levels
- Populate all possible combinations of the top 2 factors with m and n levels, creating a matrix (mn X N), with the first two columns filled and the rest empty. Each row in this matrix represents a test case and columns 1..N represent the factors sorted in descending order of the number of levels
- Empty cells are filled with an appropriate value column by column (from left to right), such that maximum number of valid pairs are covered in the resulting test case
- After the (mn X N) matrix is completely filled, there might be pairs that may not be covered. These are listed out and additional rows are added to accommodate these pairs. A greedy algorithm is used in adding every additional row, with the additional pairs getting covered as the objective function. Other values are added so that levels are uniformly distributed in a column.
- The termination condition is that all pairs are covered

The number of testcases generated to cover all pairs has been slightly better than the available commercial tools. The method and tool were used in testing one of the small features in a mobile phone application. In one of the regression rounds, the existing test cases as well as the Oats based test cases were run and the test suite generated

by the internal tool, performed better than the existing test suite, in terms of number of test cases, test design effort and number of defects found. Following table (Table 3) shows the comparison:

	Existing Suite	OatsGen Generated
Number of testcases	776	30
Number of Defects found	1	8
Test Efficiency (Defects/test case)	0.026	0.267
Test execution effort required	4 staff days	1.5 staff days
Test Design Effort Required	4.5 staff days	1 staff day

Table 3: Early Pilot Results

Encouraged by the early results and feedback, the tool got enhanced with additional features required by the practitioners. Our experience is that this method is applicable to all forms of testing – Unit Testing, Integration Testing, Feature Testing, System Testing, etc. where combinations are involved. It gives a structured way of designing the test cases, thereby saving effort during the test-design phase and saves effort during the test execution phase, as it generates a minimal set of highly effective testcases.

The learnings from the early pilots also helped us come up with the process steps for using this method. These steps are explained in the context of a real-life pilot described in the following section.

4. Case Study – Mobile Application Testing

This case study explains the process steps we designed to use OATS methodology for mobile application testing. The recently introduced slider functionality in our phones has been tested using the OATS approach and will be explained here. The functioning of a slider is significantly different from that of a flip in a phone. Combinations of different settings and phone states determine the slider phone behavior, which is not uniform across the applications. This makes it a fit case to use OATS approach for test designing.

Step 1: Partitioning the requirements in to logical groups:

When you attempt to test a feature or an application as a single entity using OATS, you will find it difficult to do so. This is because; an application's requirements are made up of distinct functional areas and the factors controlling each functional area may not be the same.

So the first step before attempting OATS based test design is to divide the application's requirements in to logical groups. For example, the slider feature's requirements can be divided as follows:

- Continue Task / End task functionality
- Slider Tone functionality
- Voice Call / PTX/PTT/VR functionality
- Camera / Video view finder functionality
- Key pad lock / unlock functionality

Step 2: Scope and approach of testing:

Based on the phase and type of testing you are doing, your scope of testing for different requirement groups could differ. If you are a component tester, you might want to test only the requirements your feature team is responsible for. If you are a system tester, you might want to test from the feature interaction perspective. If you are a UI tester, you might just want to test for the UI issues.

Based on the scope of testing, you may need to test a group of requirements for a significant number of factor combinations and some requirement groups would just need traditional testing (using equivalence partition & boundary value analysis).

In the slider example, if we consider UI testing as the scope, Key pad lock/unlock functionality or slider tone functionality testing is not as complex as Voice call / PTT functionality and traditional test design can be used to detect any UI issues. Where as Voice call /PTT functionality is complex enough to warrant OATS approach as we will see in the next step.

Step 3: Identifying, organizing & validating the factors

A simple approach to identifying the factors is to go through the group of requirements you plan to test and scribble down the variables you come across. Every noun you see in a requirement could possibly be a variable.

For example, in the requirement " This screen shall be displayed upon receiving an incoming call with the slider closed, speaker phone off , and no headset connected" , the variables are "incoming call" , "slider (state)" , "Speaker phone (setting)", "head set connection (status)".

Once you have scanned through all the requirements and collected the list of all variables, separate them in to two categories –

- Passive variables, that do not affect the requirements to be tested irrespective of the value you assign to them
- Active variables that can take multiple values and influence the functionality.

Brain storming on the end to end use cases in case of new features and analyzing the defects fixed on the features, in case of old features will help a lot in identifying all the applicable variables.

Once you have the list of all the active variables (factors), organize them in to different parts of your test case flow.

A typical test case has the following parts:

- Initial conditions & settings
- The phone state at which the test is executed
- Actions performed by the user
- Events received
- Observations and comparison with expected results

In the slider example, the factors affecting the Voice call /PTT/ VR functionality can be organized as follows:

Factors representing initial conditions/settings:

- Slide Closed setting (Continue Task / End Task)
- PTT Answer mode (Auto / Manual)

Factors representing the phone state at which the test is executed:

- Active dialog on the screen (Idle screen, Call Connected dialog etc.)
- Slider status (Open / Closed)
- Speaker Phone Status (On/Off)
- Head phone connection status (connected / not connected)

Factors representing the user actions & events:

- User Action (Accept incoming call, Initiate a PTT Call /Invite etc.)

Once you complete the selection of factors, you need to validate the factors to see if they are mutually independent or not.

- Take every pair of factors and see if you can assign the values (Levels) to them independently.
- If the value of a factor can be uniquely determined with the knowledge of the values of other factors, then the factor under consideration is wrongly chosen.

Step 4: Assigning the Levels

- Assigning the right number of levels to each factor is very important to get the optimum number of tests with out compromising on the effectiveness.
- Equivalence partitioning technique, Boundary value analysis should be used to determine the levels
- Selecting the levels, where possibilities are many, may not be simple enough. As you can see in Table 4, determining the levels for the factor "Active dialog" in the slider example, is not straight forward. The levels have been determined after analyzing different use cases and the likelihood of detecting UI issues based on previous defect history.

In Table 4, you will see that Head set and Speaker phone status is clubbed together. This is because; speaker phone functionality won't be available when the head set is connected. If the levels of a factor are fully dependent on the levels of the other factor, they should be merged in to single factor. If there is a partial dependency, then constraints can be used, as you will see in the next step.

Step 5: Identification of Constraints

- It is often possible that, when you assign certain level to a factor, it constrains (limits) the levels other factors can take.
  - For example, when the "Active dialog" is "Idle screen with key pad locked", then user can not "initiate a (normal) voice call", with out unlocking the phone.
- The constraints like above need to be generated by carefully going through the levels of every factor pairs.
- Missing constraints result in the generation of invalid tests.
- Invalid tests significantly reduce the overall effectiveness of the test cases generated.
- Some of the constraints identified for the Slider example are listed in Table 5.

	Initial Settings		State of the Phone			Test Step
Factors	**Slider Close Action**	**PTT Answer mode**	**Head set & Speaker phone status**	**Slider State**	**Active Dialog**	**User Action**
Level 1	Continue Task	Auto	BT Head set connected	Open	Idle - Keypad locked	Initiate a voice call
Level 2	End Task	Manual	Wired Headset connected	Closed	Idle - Background MP3 on	Initiate a call with VR key
Level 3			Speaker phone on		Call Connected	Accept voice call
Level 4			Speaker phone off		Browser Download on	Initiate PTT call/Invite
Level 5					Running Midlet	Answer a PTT call
Level 6					Phone book Summary view	PTX-Send new video/Picture
Level 7					Full Screen Picture viewer	Answer a PTV call request
Level 8					Video capture on	Answer Delayed voice call
Level 9					Blue tooth transfer on	

Table 4: Test Inputs

Step 6: Generation of OATS tests

Once you have the factors, levels and constraints ready, OATS tests can be generated using the tool.

The number of test runs generated is 54, which is far less than ~2300 test cases, if we were to test all the combinations. These 54 tests cover 23 requirements

specified in the requirements document. The number may sound more compared to the number of tests you would have developed to cover the same requirements in a traditional approach. But the pair wise coverage provided by the OATS is 100% here, which is very important. In the areas where not more than 2 factors influence most of the requirements, traditional design approach can yield less number of tests compared to OATS, with the same test effectiveness. But there are ways to optimize the number of test runs generated by OATS tool, as the next step shows:

Step 7: Optimization of OATS tests

After the first attempt, it is likely that you will see significantly more number of tests generated by the tool, than you normally expect to develop. If this is the case, you should optimize in the following order:

- Go through the tests (runs) generated to identify any invalid runs. Invalid runs indicate missing constraints or wrong factor & level combinations.
    - Make necessary corrections to the factors/ levels and constraints and regenerate the tests
- If the number of tests generated is still more than your expectation, look at the factor with the maximum levels to see if it needs all the levels or if some optimization is possible. Reducing the levels of a factor with maximum levels can significantly reduce he test runs generated.
- After removing couple of levels from the "Active Dialog" factor, which has the maximum levels, the test runs reduced from 54 to 42. (22% reduction).
- Optimization need not necessarily mean the reduction of tests. You can add any 3–level or 4-level combinations that are critical for testing, but missed by OATS.

Step 8: Adding Observations & Expected Results

Once OATS test runs are generated, you need to add observations/ expected results for each run, to make it a complete test case.

Each run may involve multiple observations to be made. Adding observations need in-depth usecase/requirements analysis, understanding of the existing phone behavior and awareness of undocumented requirements. While adding observations, you may discover invalid test runs caused by missing constraints or impractical use cases. This will need adding necessary constraints and regenerating of OATS runs.

It is important to check the traceability of the test cases to the requirements.

- In case, you find the tests that do not correspond to any requirement, then either the test case is invalid or there is a gap in the requirements.
- In case, you find the requirements that are not covered by any test, analyze the reasons. It could be due to missing factors / levels or simply because observations/ expected result field is not updated properly.
- In certain cases, a requirement may be an odd-man-out in the requirement group you are testing and you may have to add separate test cases to cover it.

Table 6 shows a test run with observations added:

If Factor ...	is at level ...	then Factor ...	can't be ...
Active Dialog	Call Connected	User Action	Initiate PTT call/Invite
Active Dialog	Call Connected	User Action	PTX-Send new video/Picture
Active Dialog	Full Screen Picture viewer	User Action	Initiate a call with VR key
Active Dialog	Full Screen Picture viewer	User Action	Initiate PTT call/Invite
Active Dialog	Full Screen Picture viewer	User Action	Initiate voice call
Active Dialog	Full Screen Picture viewer	User Action	PTX-Send new video/Picture
Active Dialog	Idle - Keypad locked	User Action	Initiate a call with VR key
Active Dialog	Idle - Keypad locked	User Action	Initiate PTT call/Invite
Active Dialog	Idle - Keypad locked	User Action	Initiate voice call
Active Dialog	Idle - Keypad locked	User Action	PTX-Send new video/Picture

Table 5: Feeding Constraints

5. Learnings

Following are our learnings from the use of this approach.

- **Invalid Pairs**

  When combinations are generated, some of the pairs could be Invalid. For example, if factor A takes level

'a' factor 'B' cannot take value 'b'. Example: When the view finder is 'on', Back light / Display time out can not be 'off'. This has been addressed through giving constraints as additional input to the tool. The tool masks invalid pairs from being generated.

- **Higher order Constraints**

If factor A takes level 'a' and factor B takes level 'b', then factor C cannot take level 'c'. Since such constraints were not frequent and the implementation may be complex, it was not planned to be implemented in the tool. Instead a work-around has been used. In this case a new compound factor A x B x C is introduced and the incompatible ones removed.

	Initial Condition		State of the phone			Test Step	
Test Case Id	Slider Close Action	PTT Answer mode	Head set & Speaker phone status	Slider State	Active Dialog	User Action	Observations & Expected Results
1	Continue Task	Auto	BT Head set connected	Open	Idle - Keypad locked	Accept voice call	1. Verify that BT icon on the status area is not blinking. 2. After accepting the incoming call, observe that the audio is routed to BT headset 3. Closing the slider should not disconnect the call 4. Pressing the End key should disconnect the call. 5. Long press of BT head set key should connect to the last dialed call.

Table 6: Test Case (with expected results)

- **Sub attributes**

Let us say a factor 'A' has sub attributes X, Y, Z which have their own levels. There are at least three possible approaches.

1. Approach-1: Treat A as compound factor and consider all possible combinations of X, Y, Z [X x Y x Z] (where x is the cross product) as the levels
2. Approach-2: Treat A as compound factor and consider Oats generated combinations of X, Y, Z as the levels of this compound factor A., (i.e) Input X, Y, Z as the three factors with their corresponding levels and generate the levels to be input to A.
3. Approach-3: Split A into three simple factors A.X, A.Y and A.Z. Levels of X are the levels of A.X, levels of Y are the levels of A.Y and levels of Z are the levels of A.Z.

In our experience, we found option 3 to be giving less number of testcases, however we do not have a theoretical proof for the same.

- **Treating error conditions**

Typically we did not find listing out invalid values for a factor as its levels useful. Typically the application under test would terminate in between because of the invalid value, preventing further testing with the other factors, making such test cases less effective from the pair-wise coverage perspective. We found it better to generate the test cases leaving out these erroneous values and then adding few test cases to address these erroneous values. By this approach we avoid bloat up in the number of test cases, because of the increased number of levels and the resulting additional testcases that may be required to cover the additional pairs involving erroneous values.

- **Priority between the Levels**

In our interactions with project teams in getting this approach adopted, another input we came across was that some of the levels under a factor were more important than others. This has been accommodated by marking the values in the generated test cases as replaceable, when they can be replaced without affecting the pair-wise coverage criteria. The tester can replace these values which they perceive to be of higher priority. The generation algorithm also uses the list of values sequentially. When any value can appear for a particular factor, it chooses the top most value subject to it satisfying the constraints with the values of the other factors in the test case. Testers are advised to list out the values of a factor in the descending order of priority.

- **Peer Reviews of the OATS tests**

The OATS tests must be peer reviewed with the developers to make sure that the understanding of the requirements, especially with regard to the interplay of factors is same.

- All the requirement ambiguities should be clearly identified and the requirement documents should get updated.
- The observations and expected results added to the OATS Runs should be updated based the review comments.

- The test cases generated using the steps above can be easily uploaded in to the test repository/test management tools, by just adding test case IDs and header data.

- **Execution of OATS Tests**

  - Ordering the execution of test cases generated by this method is a key precaution. This is because, it may be very inefficient, if the test cases are ordered in such a way that you need to change most of the factors to different levels. This may not be straight forward.
    - You may have to change flex settings, use a different accessory, change the language pack, use different SIM or populate phone memory with a different contact list. All this takes significant amount of time.
  - It will help to reduce the test execution effort, if you choose the execution sequence, in such a way that, most time consuming level changes occur less often.
    - For example, you may execute all tests related to one language pack at one go and move to a different language pack, if changing language pack is a time consuming step.
  - Test automation is a good option to the above problem, but it should be attempted only after ensuring that tests generated are the optimum ones and will not undergo major changes.
    - It is advisable to do first few rounds of OATS testing manually so that the test effectiveness can be determined and then automation can be attempted after fine tuning the test runs.
  - After completing the execution, you may find that, certain combination of levels can not be practically executed. These runs may be removed from the test runs after necessary risk assessment is made.
- Analyzing the faults founds by OATS test execution

One should analyze the faults found by OATS to understand what factors and what levels are involved.

- If you see that a group of faults has a single factor changing and rest of the factors remaining constant, then all those faults probably belong to same area of the code.
  - You should discuss with the developer to see if all these faults belong to same root cause and if so raise a single CR (Change Request). Else, the result will be duplication of your CRs.
- In the same way, you can identify and analyze double mode faults and triple mode faults as well.
- And finally, you should identify which levels & factors are part of the faults identified and which are not part of any fault found. This will help in optimizing the factors and levels.

- **Gaps in requirements can surface**

When the testers are not able to fill in the expected outcome, it could point to gaps in requirements which may have to be filled in.

- **The common pitfalls to avoid:**

  - Selecting an area which is simple to test will not yield any additional benefit by using OATS. If number of factors is less than 4, each with 2 to 3 levels, then traditional approach of testing all combinations still is used with out much of additional effort.
    - The power of OATS lies in dealing with complex areas, where factors are way too many. It generates manageable number of tests and still provides 100% coverage of all pairs.
  - It's good to select an area, which is complex & critical and where base code defects keep surfacing every now and then, despite your best effort to contain them.
  - If a module to be tested uses hash tables or look up tables to generate outputs, using OATS based test approach may not be right thing.
- Theoretical bounds on the errors detected/missed out may not be possible. Studies indicate that roughly 70% of the reported bugs could have been covered through testing pair-wise combinations [3]. This makes the review of generated testcases a very critical step. To assist in this, the tool has a feature by which it lists out all the missed out three way pairs, so that the important ones can be identified through domain experience and added to the test suite.

6. Conclusion

Our overall experience in inducting the use of this method in our organization has been very positive. It has brought a fresh perspective to testing. We are currently exploring the possible use of factors, levels and combinations in the requirements and design phases. Some of our teams are also exploring the possibility of automating the execution of OATS generated testcases in their test environment.

References

1. Madhav S Phadke, Phadke Associates Inc: Planning Efficient Software Tests. In Crosstalk, October 97, Vol. 10, No. 10, pp.11-15
2. David M. Cohen (1997) :The AETG System: An approach to Testing based on Combinatorial Desig. In IEEE Transactions on Software Engineering, Vol 23, No 7, 1997.

3. D. Richard Kuhn et.al (2002): An Investigation of the Applicability of Design of Experiments to Software Testing. In Proceedings, 27th NASA Goddard Space Flight Center, 4-6 December, 2002.
4. Library of Orthogonal Arrays. In http://www.research.att.com/~njas/oadir/
5. Myra B. Cohen et.al (2003): Constructing test suites for interaction testing. In Proc. of the Intl. Conf. on Software Engineering (ICSE 2003), Portland, Oregon, May 2003, pp. 38-48 .

ABOUT THE AUTHORS

R. Krishnan is a principal staff engineer with Motorola Software Group-India, where he leads the software technology function. He holds a PhD in computer science. He has successfully introduced several software engineering tools & practices in the organization. Krishnan has worked on a range of software engineering areas: software reuse, architecture, product quality and testing, and is a Six Sigma Green Belt.

Murali Krishna is a senior software engineer with Motorola. He is an alumnus of IIT Kharagpur and has been with Motorola for the past 4 years. Murali has worked on AOP, static analysis of software, unit test automation, combinatorial testing & software security

P. Siva Nandhan is an Engineering Manager with GSG-India, where he leads the feature testing group of User Interaction Services. He holds a B.Tech in Electronics Engg. He has worked on a wide range of test projects including stack and applications testing. He was actively involved in the evolution of the in-house 3G Stack Test Framework. Siva Nandhan is an active proponent of DSS based process improvements and is a Green Belt Candidate.