IJCSE International Journal of Computer Sciences and Engineering Open Access
Review Paper Vol.-7, Issue-1, Jan 2019 E-ISSN: 2347-2693

A Review of Hybrid Exploratory Testing Techniques

Manas kumar Yogi[1*], Y. Jnapika[2], Bhanuprakash Peddireddy[3]

[1]Computer Science & Engineering,Pragati Engineering College,Surampalem,Kakinada, India
[2] Computer Science & Engineering,Pragati Engineering College,Surampalem,Kakinada, India
[3] Computer Science & Engineering,Pragati Engineering College,Surampalem,Kakinada, India

[*]*Corresponding Author: manas.yogi@gmail.com, Tel.: 09966979279*

Available online at: www.ijcseonline.org

Accepted: 14/Nov/2018, Published: 31/Jan/2019

***Abstract*—**Wildcat testing contains a mess of strategy related to it. It is a decent combination of structured thinking and race exploration that may be terribly powerful for locating bugs and substantiate correctness. This paper shows however the wildcat testing mentality is often combined with additional ancient scenario-based and scripted testing. This hybrid technique relaxes a lot of the rigidity unremarkably related to scripting and makes smart use of the wildcat testing steering bestowed. It additionally permits groups that square measure heavily unconditional in existing scripts to feature wildcat testing to their arsenal. Ancient state of affairs testing is incredibly seemingly to be a well-known idea for the reader. Several testers write or follow some type of script or end-to-end state of affairs once they perform manual testing. State of affairs testing is well-liked as a result of it lends confidence that the merchandise can faithfully perform the state of affairs for actual users. The additional the state of affairs reflects expected usage, the additional such confidence is gained. The additional part that wildcat testing lends to the current method is to inject variation into the state of affairs in order that a wider swath of the merchandise gets tested. Users can't be unnatural to merely execute the software package the manner we have a tendency to intend, therefore our testing ought to expand to hide these extra state of affairs variants.

***Keywords*—** Hybrid, exploratory, scenarios, testing, tour

I. INTRODUCTION

Scenario-based exploration can cowl cases that straightforward state of affairs testing won't and additional accurately mimics real users, UN agency typically stray from the most scenarios[1]: finally, the merchandise permits several attainable variations. We must always not solely expect that they get used; we must always check that they're going to work.

The idea behind scenario-based wildcat testing is to use existing situations (we remark wherever to induce situations during this paper) very much like real explorers use a map to guide themselves through a geographical region or different unfamiliar piece of land. Scenarios, like maps, square measure a general guide regarding what to try and do throughout testing, that inputs to pick, and that code methods to traverse, however they're not absolutes. Maps could describe the situation of your destination however supply multiple ways that to induce there[2]. Likewise, the wildcat tester is obtainable alternate routes and even inspired to think about a large vary of attainable methods once death penalty a state of affairs. In fact, that's the precise purpose of this manner of wildcat testing: to check the practicality delineate by the state of affairs, adding the maximum amount variation as attainable. Our "map" isn't supposed to spot the shortest route; it's supposed to seek out several routes. The additional we will check, the better; this results in additional confidence that the software package can perform the state of affairs robustly once it's within the hands of users UN agency will and can deviate from our expectations.

There is no formal definition of eventualities that we do know of that very helps testers. Some scenarios are like maps, providing only general guidance, and others are more like printed driving directions with step-by-step instructions for every turn and intersection. In general, scenarios are written prose that follow no fixed format but describe how the features and functionality of the software under test work to solve user problems.

A state of affairs will describe inputs, information sources, setting conditions (things like register settings, obtainable memory, file sizes, then forth) further as UI components,

outputs, and specific information about how the software under test is supposed to react when it is used. The scenarios themselves often originate from outside the tester's domain. They can be gleaned from artifacts inherited from design and development. Requirements documents and specifications typically describe their purpose in the form of scenarios. Some forms of agile development require the creation of user stories; requirements are often documented with example scenarios of expected usage. In many cases, testers don't need to write the scenarios as much as gather them. In fact, recordings made (using capture/replay tools, keystroke recorders, and so forth) during testing are also legitimate scenarios, and thus the tours of the previous paper can be the source of a great number of high-quality scripts and scenarios[3]. Any and all such scenarios can be used as the starting point for exploration.

In general, a useful scenario will do one or more of the following:

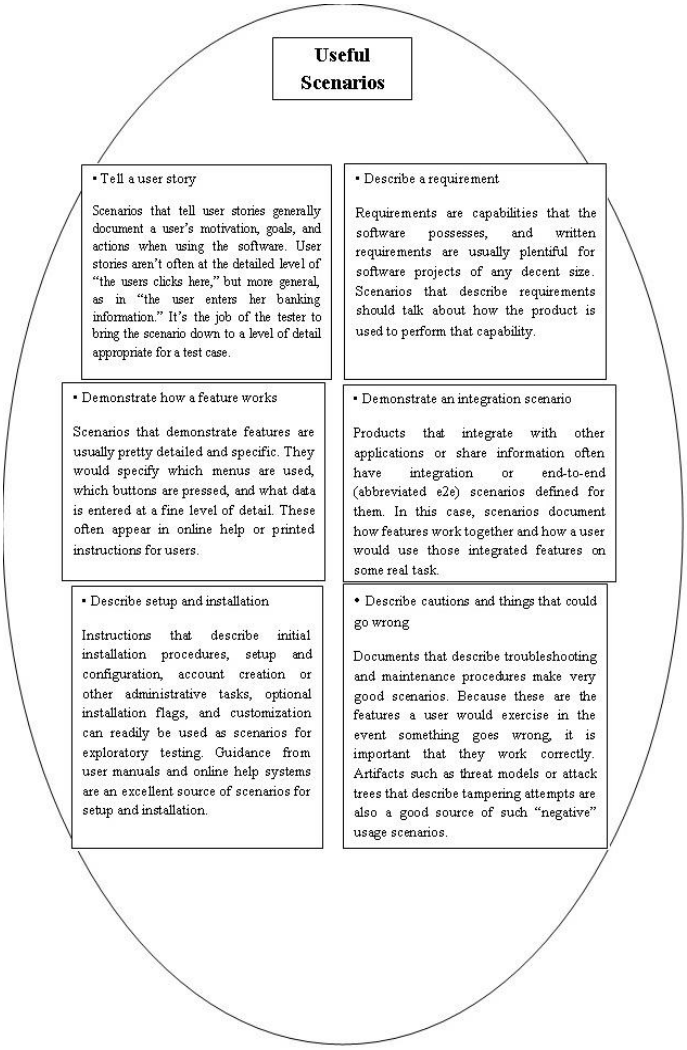

Figure 1.Allocation of jobs to processors

Exploratory testers ought to push to make sure they gather as several eventualities as doable from all of those classes. It is then our task to follow the eventualities and inject variation as we have a tendency to see work. It is however we decide to inject this variation that produces this task exploratory in nature which is that the subject we have a tendency to intercommunicate next[4].

II. PRINCIPLE

1. Applying Scenario-Based Exploratory Testing

State of affairs testing works as a result of it mimics the means a true user would behave and therefore it finds bugs that, if they survived testing, would plague actual users. However rarely do real users confine themselves to usage of the package as represented by the state of affairs. User's area unit liberated to vary from the state of affairs by adding steps or taking them away, and those they do therefore consistent with their own schedules and timetables. it's our task to second-guess such variation and guarantee they get tested as a result of they represent a number of the foremost possible ways in which during which the package are used when it's been free[5].

Injecting variation into eventualities is what this manner of exploratory testing all is concerning. One written state of affairs may be became several individual check cases by methodically considering selections in input choice, data usage, and environmental conditions. 2 main techniques area unit won't to accomplish this: state of affairs operators and tours.

2. Introducing Variation through Scenario Operators

Exploratory testing may be combined with state of affairs testing to assist a tester explore minor and even major state of affairs[6]. Wherever a state of affairs describes specific actions for a tester to require, the techniques represented next may be wont to transpose those actions and make deviations from the scenario that will test different states and code paths. Where a scenario describes general activity, these techniques can be used to select among the possible choices and allow a tester to consider alternate paths in a more methodical manner.

We introduce the concept of scenario operators to achieve this goal. State of affairs operators are constructs that treat steps at intervals a state of affairs to inject variation into the state of affairs. After we apply a state of affairs operator to associate existing state of affairs, we have a tendency to get a replacement state of affairs that we have a tendency to decision a derived state of affairs.

A tester can apply one or more scenario operators to a given scenario and even apply operators to derived scenarios. The amount and number of such operators is, in true exploratory

fashion, up to the individual tester and can be performed in advance of testing or, my preference, on-the-fly[7].
The scenarios state of affairs operators within the following subsections are those most testers can realize helpful.

i) Inserting Steps
Adding extra steps to a state of affairs will create them a lot of numerous and permit them to check a lot of practicality. Inserting one or more steps into a scenario creates more opportunity for the software to fail[8]. Code paths may be executed with different data, and the state of the software will be varied in ways that are different from what the original scenario allowed. The additional steps can be
• Adding a lot of data: once the state of affairs asks for, say, ten records to be additional to a information, the tester ought to increase that to twenty or thirty records or even more if it makes sense to do so. If the scenario requires an item to be added to the shopping cart, add that item and then some additional items on top of that. It is useful also to add related data so that if the scenario calls for a new account to be created, we may also add information to that account over and above what the scenario calls for.

The tester ought to raise herself, "What information is employed during this situation and the way would it not add up to extend the quantity of information I enter?"
• Victimization further inputs: once the situation involves a series of inputs to be entered, realize a lot of inputs that may be side. If the situation asks that the tester produce a product review for a few on-line searching web sites, the tester will prefer to add ratings for different client reviews, too. The concept is to know what further options square measure associated with the options within the situation and add inputs to check those new options similarly.
The tester ought to raise herself, "What different inputs square measure associated with the inputs utilized in the prevailing scenario?"

• Visiting a replacement a part of the UI: once the situation involves specific screens and dialog boxes to be used, the tester ought to establish different screens or dialogs and add those to the situation. If the situation involves a tester to pay a bill on a monetary services web site, the tester may prefer to conjointly visit the pages to examine account balances before submitting the payment.

The tester ought to raise herself, "What different elements of the UI square measure associated with the elements utilized in the prevailing scenario?" Eventually, the steps got to loop back to the original scenario. It helps to keep in mind that the idea is to enhance the scenario, not to change it from its fundamental purpose. If the scenario was meant to add records to the database, which should still be its primary purpose and that goal should not change[9]. What the tester is doing in this scenario operator is adding inputs, data, or variation that makes the scenario longer but does not alter its core purpose.

ii) Removing Steps
Redundant and optional steps can also be removed with the idea being to reduce the scenario to its shortest possible length. The derived scenario may then be missing steps that set preconditions for other steps, testing the application's ability to recognize missing information and dependent functionality.

A tester can apply this scenario operator in an iterative fashion, removing one step at a time. In this case, the scenario actually gets executed against the software under test each time a step is removed until the mini-mal test case ends the cycle. For example, a scenario that requires a tester to log on to a shopping site, search for items, add them to a shopping cart, enter account info, complete the purchase, and finally log off would be eventually reduced to just logging on and logging off (an interesting and important case to test!) with a single step being removed each time the test case is run.

iii) Replacing Steps
If there is more than one way to accomplish some specific step in a scenario, this scenario operator is the way to modify the scenario to accomplish that. It's really a combination of the preceding two operators in that replacement is the same thing as removing and then adding[10].

The tester must research alternate ways of performing each of the steps or actions in a scenario. For example, instead of searching for an item to purchase, we might simply use its item number to look it up directly.

Because the software under test provides both of these as options, we can create a derived scenario to test the alternative. Similarly, we might use key- board shortcuts instead of the mouse or choose to bypass creating an account and just purchase an item without registering on the site. Testers need to be aware of all the different options and functionality that exists within their application to be truly effective at applying this scenario operator.

iv) Repeating Steps
Scenarios often contain very specific sequences of actions. This operator modifies such a sequence by repeating steps individually or in groups to create additional variation. By repeating and reordering steps, we are testing new code paths and potentially finding bugs related to data initialization. If one feature initializes a data value that is used by another feature, the order in which the two features is executed matters, and reordering them may cause a failure.

Often, certain actions make sense to repeat. For example, if we are testing a financial services website for the general

scenario of log in to an account, check the balance, pay bills, make a deposit, and then log out, we may repeat the "check the balance" action after we pay the bills, and then again after making the deposit. The general scenario is the same, but we have repeated an action that a user is also likely to do. The same can be said of actions such as "view the shopping cart," which could happen over and over during a scenario for an online shopping site. Repetition can also occur with multiple actions, so that we pay one bill, check the balance, pay another bill, check the balance, and so forth. The tester's task is to understand the variability and create repetitive sequences as appropriate.

v) Data Substitution

It is often the case that a scenario will require a connection to some database, data file, or other local or remote data source. The scenario then specifies actions that the tester performs to cause that data to be read, modified, or manipulated in some way. Testers need to be aware of the data sources that the application under test interacts with and be able to offer variations.

Are there backup databases, alternate test databases, real customer databases, and so forth that are accessible to testers? If so, use those when testing the scenarios instead of the default. What if the data source is down or otherwise unavailable? Can we create or simulate that situation so that we can test how the system under test reacts? What if the data source holds ten times as many records? What if it only holds one record?

The idea here is to understand the data sources the application connects to or uses and to make sure that interaction is robust.

vi) Environment Substitution

Testing is necessarily dependent on the environment in which the software resides when we run our test cases. We can run billions of tests successfully when the software is in one environment only to have them all fail when the software is put into a different environment[11]. Therefore, this operator is used to ensure those alternate environments receive testing. The simple part of this operator is that the scenarios themselves don't actually change, only the system on which the software is running when the scenario is applied. Unfortunately, understanding which parts of the environment to change, and actually enacting that change, is very difficult. Here are some considerations:

• Substitute the hardware: The easiest part of the environment to vary is the hardware on which the application under test case runs. If we expect our users to have a range of hardware from fast and powerful to antiquated and slow, we need to acquire similar machines for our test lab and ensure that we have beta customers willing to help us with testing and pre-release validation. Of course, this is an excellent use of virtual machines as well.

• Substitute the container: If our application runs inside a so-called container application (like a browser), we need to ensure that our scenarios run in all the major containers we expect our user to have access to.

Browsers like Internet Explorer, Firefox, Opera, and Chrome or platforms like Java or .NET or even animation tools like Flash and Silver light will impact the way our applications run.

• Swap out the version: All the previous containers also have earlier versions that still enjoy market share. How does your app run in the earlier versions of Flash?

• Modify local settings: Does your application use cookies or write files to user machines? Does it use the local Registry? What happens when users modify their browser settings to limit these types of activity? What happens if they change your application's Registry settings directly (without going through your app)? If you don't test these things, your users likely will, and their doing so may bring a nasty post-release surprise to your engineering team. It's better to find out for yourself before the app ships how it will handle these things.

When using any of these operators to create derived scenarios, it is generally the case that we try to stay as true to the original scenario as possible.

Using too many operators or using operators in such a way as to make the origin of the derived scenarios unrecognizable is usually not useful. But don't take my word for it. If you try it and it finds good bugs, then it's a useful technique!

However, such broader based modification of tours is the job of the second technique to inject scenario.

3. Introducing Variation through Tours

At any point in the execution of a scenario, one can stop and inject variations that will create derived scenarios. The scenario operators described above are one way to do this, and using the tours is another. We like to think of this use of tours as side trips.

The idea is simple: A tester reviews the scripts looking for places where decisions can be made or places where it is possible to fork the logic within the. We like to use the analogy of a car tour or even a hike in the woods on foot. It's often that on such a trip there is some scenic overlook at which one can park the car and take a short walk to some monument or beautiful view before returning to the car and continuing the voyage. That short side trip represents the tour, and the longer car ride is the scenario. This is a useful technique for adding variation to scenarios[12]. The key difference between scenario operators and tours is that tours end up creating longer side trips, in general, than operators. Operators focus on small, incremental changes and optional steps in a scenario, and tours can actually create derived scenarios that are significantly longer and broader in scope.

Just as some side trips can turn into a destination all their own, it may be that the tours overwhelm the original scenario, and this can actually be a very desirable effect. It's good to always remember that exploratory testing is about variation, and when scenarios are combined with tours, the result can add significant variation. It's up to the tester to determine whether the variation is useful, and it is often the case that one has to build up some history to determine which tours are most effective for a given application.

Here is a list of tours that are effective as side trips during scenario- based exploratory testing. The tours a few times, you should be able to determine how to best follow this advice for your particular situation.

1. The Money Tour:
Can any major features not already used in the scenario be easily incorporated into the scenario? If so, modify the scenario to include the use of a new feature or features. Assuming that the original scenario already included some features, this will help test feature interaction in a scenario-driven way. If the scenario was a realistic user scenario, it's even better because we are mimicking the user including another feature into his existing work habits (as represented by the scenario). There are many users who will learn a feature, master it, and then move on to new features as their familiarity with the application grows. This technique mimics that usage pattern.

2.The Landmark Tour:
Start with a scenario and pick specific feature landmarks out of the scenario. Now randomize the order of the landmarks so that it is different than the original scenario. Run some tests with the new order of landmark features and repeat this process as often as you think is necessary. Obviously, that will depend on how many landmarks you are dealing with; use your own judgment. This combination of the Landmark tour within a structured scenario has been very valuable at Microsoft.

3.The Intellectual Tour:
Review the scenario and modify it so that it makes the software work harder. In other words, ask the software hard questions. If the scenario requires the software to open a file, what is the most complicated file you can give it? If the software asks for data, what is the data that will make it work the hardest? Would very long strings do the trick? What about input that breaks formatting rules (for example, Ctrl characters, Esc sequences, and special characters)?

4.The Back Alley Tour:
This is an interesting variation on the Money tour. Both tours suggest we inject new features into the scenario, but the Back Alley tour suggest the least likely or least useful features instead. Granted, this variation will find more obscure bugs, but if an application is widely used, there may be no such thing as least likely because every single feature will get used by someone, and all paying customers are important.

5.The Obsessive-Compulsive Tour:
This one is straightforward: Repeat every step of the scenario twice. Or three times. Be as obsessive as you like! Specifically, any step in a scenario that manipulates data is a good one to repeat because it will cause internal data to be manipulated and internal state to be set and then changed. Moving data around the software is always an effective way to test and to find important bugs.

6.The All-Nighter Tour:
This one is best when a scenario can be automated or even recorded and then played back. Just run the scenario over and over without ever exiting the application under test. If the scenario specifies that the software be shut down, remove that clause and keep the scenario running over and over again. Choose scenarios (or derived scenarios) that make the software work hard, use memory and the network, and otherwise consume resources that might over time cause problems.

7. The Saboteur:
Scenarios are a great start for sabotage. Review the scenario or derived scenario and make a note every time it uses some resource (another computer, the network, file system, or another local resource) that you have access to, and then when you execute the scenario, sabotage that resource when the scenario calls for it to be used.

For example, if a scenario causes data to be transmitted over a network, unplug the network cable (or disconnect it via the OS or turn off the radio switch for wireless connections) just before or while you are executing that particular step of the scenario. Document all such sabotage points and execute as many of them as sensible or prudent.

8.The Collector's Tour:
Document every output you see as you execute scenarios and derived scenarios. You can even score scenarios based on the number of such outputs they force. The more outputs, the higher the score for that scenario. Can you create (or derive) new scenarios that cause outputs that are not in any of the other scenarios? Can you create a super scenario that causes the absolute maximum number of outputs possible? Make a game out of it and let your testers compete to see who can generate the most outputs, and give prizes to the winners.

9.The Supermodel Tour:
Run the scenario but don't look past the interface. Make sure everything is where it is supposed to be, that the interface is sensible, and watch particularly for usability problems. Choose scenarios that manipulate data, and then cause it to

be displayed on the UI. Force the data to be displayed and redisplayed as often as possible and look for screen-refresh problems.

10. The Supporting Actor Tour:
I think of this as the Nearest-Neighbour tour, in that instead of exercising the features as described in the script, the testers find the nearest neighbouring feature instead. For example, if a scenario specifies an item on a drop-down menu, choose the item above or below the one specified. Whenever a choice is presented in the scenario, choose not the one suggested but one right next to it (either by proximity on the interface or close in semantic meaning). If the scenario specifies using italics, use boldface; if it wants you to highlight some text, highlight other text instead, always choosing that which is "nearest" in whatever way makes the most sense.

11.The Rained-Out Tour:
This is the tour that not only makes good use of the cancel button (press it whenever you see it while running the scenario) but also in starting and stopping execution. Review the scenarios for time-consuming tasks such as complicated searches, file transfers, and the like. Start those features, and then cancel them using provided cancel buttons, hitting the Escape key and so forth.

12.The Tour-Crasher Tour:
This tour is new for this paper and didn't appear earlier when the tourist metaphor was first described. Indeed, it is specific to scenario-based testing. The concept is based on those people who don't pay for the tour when it begins, but join it in progress by simply melting into the crowd and acting like they've been there all the time. They not only crash a tour, but they also may even hop from tour to tour as they encounter other groups (in a museum or some historical building where tours are continuous) of tourists. We're going to adopt this process for hopping from scenario to scenario as a way of combining two or more scenarios into a single scenario of mixed purpose. Review your scenarios and find ones that operate on common data, focus on common features, or have steps in common. Just like the guy who peels himself away from one tour and melts into the crowd of another. He's able to do it because for some small period of time, the two tour groups are sharing the same space on the museum floor. We're able to do it as testers because the scenarios both go through the same part of the application. We'll follow one scenario to that place but then follow the other when we leave it.

III. CONCLUSION

Static scenario testing and exploratory testing do not have to be at odds. Scenarios can represent an excellent starting point for exploration, and exploration can add valuable variation to otherwise limited scenarios. A wise tester can combine the two methods for better application coverage and variation of input sequences, code paths, and data usage.

REFERENCES

[1]. C. Agruss and B. Johnson. Ad hoc software testing, a perspective on exploration and improvisation. Technical report, Florida Institute of Technology, USA, April 2000.

[2]. J. J. Ahonen, T. Junttila, and M. Sakkinen. Impacts of the organizational model on testing: Three industrial cases. Empirical Software Engineering, 9(4):275–296, 2004.

[3]. P. Ammann and J. Offutt. Introduction to software testing. Cambridge University Press, Cambridge, 2008.

[4]. C. Andersson and P. Runeson. Verification and validation in industry – a qualitative survey on the state of practice. In International Symposium on Empirical Software Engineering (ISESE 2002), pages 37–47, 2002.

[5]. J. Bach. Session-based test management. Software Testing and Quality Engineering Magazine, 2, 2000.

[6]. J. Bach. Exploratory testing. In E. V. Veenendal, editor, The Testing Practitioner. UTN Publishers, 2005.

[7]. E. Barnett-Page and J. Thomas. Methods for the synthesis of qualitative research: a critical review. BMC medical research methodology, 9(1):59, 2009.

[8]. A. Bertolino. Software testing research: Achievements, challenges, dreams. In Proceedings of the Workshop on the Future of Software Engineering (FOSE 2007), pages 85–103, 2007.

[9]. P. Bourque and R. Dupuis. Guide to the software engineering body of knowledge (swebok). Technical report, IEEE Computer Society, Los Alamitos, California, 2004.

[10]. L. C. Briand, Y. Labiche, and Q. Lin. Improving the coverage criteria of uml state machines using data flow analysis. Softw. Test., Verif. Reliab., 20(3):177–207, 2010.

[11]. N. Britten, R. Campbell, C. Pope, J. Donovan, M. Morgan, and R. Pill. Using meta ethnography to synthesise qualitative research: a worked example. Journal of Health Services Research & Policy, 7(4):209–215, 2002.

[12]. L. Copeland. A practitioner's guide to software test design. Artech House, Boston, Mass., 2004.

Authors Profile

Mr. Manas Kumar Yogi pursued Bachelor of Technology from VR Siddhartha Engineering College,Vijayawada,A.P. in 2006 and Master of Technology From Malla Reddy College Of Engineering And Technology in year 2012. He is currently working as Assistant Professor in Department of Computer Science Engineering , Pragati Engineering College (Autonomous), Surampalem, East Godavari District, since 2014. He is a member of IEEE & ACM since 2014. He has published more than 40 review,research papers in reputed international journals ,conferences including IETE sponsered conferences. His main research work focuses on Software Engineering, Distributed Computing, Cloud Security and Privacy, Big Data Analytics, , IoT and Computational Intelligence based optimisations. He has 8 years of teaching experience and 2 years of software industry Experience.

Mrs.Y.Jnapika has received her B.Tech in Computer Science & Engineering from Godavari Institute of Engineering and Technology, JNTU, Hyderabad, Andhra Pradesh, India in 2007 and her M.Tech in Software Engineering from Godavari Institute of Engineering and Technology, JNTUK, Kakinada, Andhra Pradesh, India in 2011. She is currently working as Assistant Professor, Computer Science and Engineering, Pragati Engineering College, Surampalem, East Godavari District, Andhra Pradesh, India. She has 02 years of Industrial experience at WIRPO technologies and 8.6 years of experience in teaching undergraduate students. Her research interests are in the area of Network Security, Software Engineering, Big Data Analytics, Machine Learning.

