Integrated Intelligent Research (IIR)

International Journal of Web Technology
Volume: 05 Issue: 01, June 2016, Pages: 75-80
ISSN: 2278-2389

Comparison of Component Based Testing and Model based Functional Testing through Graphical User Interface

K.Jaganeshwari[1], S.Djodilatchoumy[2]
[1]Research Scholar, SCSVMV University, Kanchipuram
[2]Head of the Department of Computer Science, Pachaiyappa’s College, Chennai

Abstract - The birth of GUI is a milestone in the development of software. It is very popular and welcomed by the consumers because of its friendly user interface and easy straightforward operations. Graphical User Interface (GUI) design is currently shifting from designing GUIs composed of standard widgets to designing GUIs relying on more natural interactions and adhoc widgets. This shift is meant to support the advent of GUIs providing users with more adapted and natural interaction and the support of new input devices such as multi-touch screens. Standards widgets (eg. buttons) are more and more replaced by adhoc ones (eg. the drawing area of graphical editors) and interaction are shifting from mono-event (eg. button pressures) to multi event interactions (eg. Multi-touch and gesture-based interactions).Model based testing is a technique to design abstract tests from models that partially describe the system’s behavior. As a consequence, the current GUI model based testing approaches, which target event based systems, show their limits when applied to test such new advanced GUIs. Functional testing is also referred to as black box testing in which contents of the black box are not known. Functionality of the black box is understood on the basis of the inputs and outputs in software. Component based software development is used for making a new software product rapidly by using fewer resources. Different components are collected and integrated together to form new product therefore the quality of new software product depends upon these components. To ensure quality of overall product testing of each component is essential. But problems arise during testing when tester has limited access to the components. The use of a model to describe the behavior of a system is a proven and major advantage to test development teams. Models can be utilized in many ways throughout the product life-cycle, including improved quality of specifications, code generation, reliability analysis, and test generation. This paper discusses the comparison of component based testing and Model based testing through GUI of such software products. Some important strategies are also discussed in this paper and considering different testing issues and a new component and model based testing strategy is proposed. A case study of component and model based system is also used to validate the effectiveness of this strategy. Also it will focus on the testing benefits from Model based testing methods and component based testing to share our experiences from a long term industrial evaluation on automatically extracting models of GUI applications and utilizing the extracted models to automate and support GUI testing.

Keywords - Graphical User Interface, Component based Testing, Model based Testing, Functional testing, Industrial test environment

I. INTRODUCTION

The constant increase of system interactivity requires software testing to closely consider the testing of Graphical User Interface (GUI). The standard GUI model based testing process is depicted in Figure 2.the first step consists of obtaining models describing the structure and behavior of a GUI of the system under test using a User Interface description Language(UIDL). These models can be automatically extracted by reverse engineering from the SUT binaries. Such a model of existing GUIs is effective for detecting crashes and regressions. In this case, GUI models are designed manually from the requirements and the testing process targets mismatches between a system and its specifications. Once a model of the GUI is available, a test model is produced, to drive test generation. In GUI testing, test models are mainly event flow graphs (EFG).EFGs contain all the possible sequences of user interactions that can be performed within a GUI. Test scripts are automatically generated by traversing the EFG according to a specific test adequacy criterion. Test scripts are executed on the SUT manually or automatically.

Finally GUI oracles yield test verdicts by comparing effective results of test scripts with the expected outputs. Component based software engineering is used to develop new software application by reusing already build components, such as third party components and in-house built components. The reusable components are integrated to develop a new software product. So the integration process of the components is very important in component based development and the quality of new developed application depends upon the quality of the components that are used in it. Component based development technique is used to minimize the complexity in software development. It reduces the delivery time of the product and increases the software productivity. It also improves the quality and reduces the maintenance cost of the product [1].

II. COMPONENT BASED SOFTWARE TESTING

A. Component Testability

Software component testing refers to testing activity that examines component along with its design, generates component tests, identifies component faults and evaluates component reliability. So component testing plays very important role for the development of a quality component based software product [4].

B. Significance of Component Based Software Testing

If bugs are not found in software system by the development company then the customer will find them. At this stage it would cause more problems in the system. If these bugs are found during testing phase by the development team then it is possible to deliver quality software product to the customer. Therefore to develop nearly a bug free software product testing is one of the key phases in software development life cycle. Without testing it is impossible to deliver such quality product which is fulfilling customer requirements. Therefore to improve company's credibility in the market it is very important to deliver bug free product.

Therefore testing is very critical part for the development of quality product. This testing focuses on the prevention of the product from the bugs and also reduces the risks which cause the system from working efficiently [5]. Cost of fixing any bug in the software system is more if it is found in the later stages. Due to this reason it become more difficult to solve the problem and cost of fixing that bug in the product grows immensely. Therefore testing process supports the software development cycle a lot to reduce the development cost from the early stage and bugs can be removed with fewer loads on the software product.

The main purpose of software testing is to identify the hidden errors in the product. After finding and removing these bugs guarantee to develop a quality product. So testing provides confirmation whether the software product will actually work according to the specification and will be functioning with very less bugs. In Component based software development, testing of each component and overall final product is very important. Since final product is made of different software components therefore the quality of overall product depends upon each component that is used. The purpose of testing each component proves the quality of the component and the overall performance of the system. If any component used in the software product fails to perform its tasks then it can lead to the overall software product failure [6].

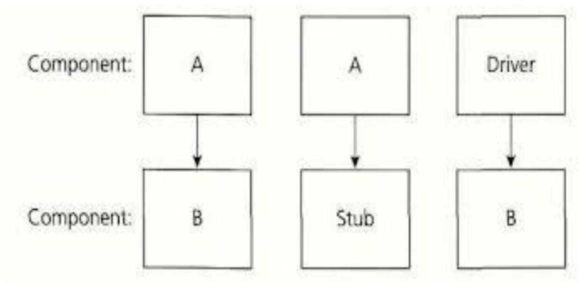

Figure 1: Component Testing

Component testing may be done in isolation from rest of the system depending on the development life cycle model chosen for that particular application. In such case the missing software is replaced by **Stubs** and **Drivers** and simulates the interface between the software components in a simple manner. Let's take an example to understand it in a better way. Suppose there is an application consisting of three modules say, module A, module B and module C. The developer has developed the module B and now wanted to test it. But in order to test the module B completely few of its functionalities are dependent on module A and few on module C. But the module A and module C has not been developed yet. In that case to test the module B completely we can replace the module A and module C by stub and drivers as required. A stub is called from the software component to be tested. As shown in the diagram below 'Stub' is called by 'component A'. A driver calls the component to be tested. As shown in the diagram below 'component B' is called by the 'Driver'

C. Component testability

Component testability is an important quality meter that is very helpful during testing phase to support quality and reliability of component.

Characteristics that ensure good component testability [15]:

1) Component Traceability: It refers how to track the component behavior and attributes. It facilitates the monitoring of component behavior in software.
2) Component Observability: This shows how component facilitates the observation of its functions and operational behaviors.
3) Component Controllability: This shows component facilitates the control of its executions during validation.
4) Component Understandability: It refers to the information about component that represents how well component is presented to facilitate component understanding.
5) Component Test Support Capability: It is validated only during component test process and focuses during component validation.

In component based software engineering (CBSE) the primary goal is to develop reusable software components. Then third party engineers use these components according to the requirements given by their customers to build the new product. So at this level testing of all integrated components is necessary as for the final product. So the component testability supports enough for the component reliability.

D. Component Test Challenges

There are some difficulties and challenges in testing components and some of these facts may cause problem in the component based system testing [1].

1) Absence of information exchange between the consumer and the producers of the components in the case of third party component may cause difficulties to test the components used for new product.
2) The unavailability of the source code for the consumer of the components causes limitations for testing the components.
3) Although there are some tools that give some support to component testing but they are not integrated with development environment that cause limitations testing components with the help of testing tools.
4) There are some limitations to create test suites that contain generic test cases because component with the similar behavior may have different usage.
5) Users and developer are not provided with the component change information and unit testing suites.

GUIs are composed of graphical objects called widgets, such as buttons. Users interact with these widgets (eg. press a button) to produce an action that modifies the state of the system, the GUI or the data model. For instance, pressing the button delete of a drawing editor produces an action that deletes the selected shapes from the current drawing. Most of these standard widgets provide users with an interaction composed of a single input event (eg. pressing a button). Indeed, GUI model based testing approaches relying on standard widgets show their limits for testing such new kinds of advanced GUIs. If they demonstrated their efficiency to find crashes and regressions in standard GUIs, testing advanced GUIs requires the ability to test adhoc widgets and their complex interactions that existing approaches cannot test.

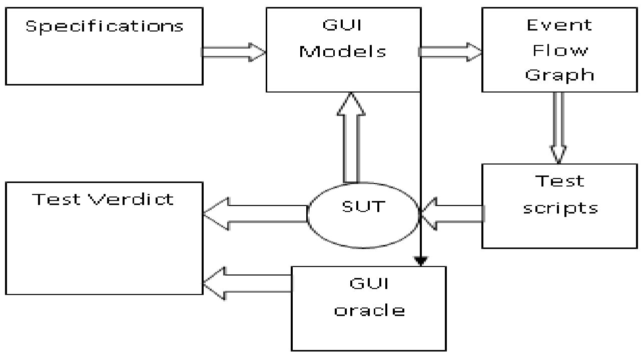

Figure.2: Standard GUI model based testing process

A primary goal of software testing is to find faults by running tests. Whether tests can find faults depends by running tests. Whether tests can find faults depends on two key factors: test inputs and test oracles. In our context, test inputs consist of method calls to a system under test (SUT) and necessary test values. A test oracle determines whether a test passes.In model based testing (MBT), a model partially specifies the behavior of a system. Abstract tests are generated to cover test requirements imposed by a coverage criterion.

III. CURRENT LIMITATIONS IN TESTING ADVANCED GUIS

Using Latex draw as an illustrative example, we explain in this section the limitations of the current approaches for testing advanced GUIs and thus for detecting errors. Studying errors found by current GUI testing frameworks GUITAR is one of the most widespread academic GUI testing frameworks that demonstrated its ability to find errors in standard GUIs . It follows the standard GUI testing process depicted by Figure 2 and can be thus studied to highlight and explain the limitations of the current GUI testing approaches for testing advanced GUIs. GUITAR developers provide information on 95 major errors detected by this tool during the last years in open-source interactive systems. An analysis of these errors shows that: 1) all of them has been provoked by the use of standard widgets with mono-event interactions, mainly buttons and text fields; 2) all of them are crashes. While several of the tested interactive systems provide advanced interactive features, all the reported errors are related to standard widgets. For instance, ArgoUML is a modeling tool having a drawing area for sketching diagrams similarly to Latexdraw. None of the errors found by GUITAR on ArgoUML has been detected by interacting with this drawing area.We applied GUITAR on Latexdraw to evaluate how it manages the mix of both standard widgets and ad hoc ones, i.e. the drawing area and its content. If standard widgets were successfully tested, no test script interacted with the drawing area. In the next section, we identify the reasons of this limit and explain what is mandatory for resolving it.

IV. LIMITS OF THE CURRENT GUI TESTING FRAMEWORKS

The main differences between standard widgets and ad hoc ones are: one can interact with standard widgets using a single mono-event interaction while ad hoc ones provide multiple and multi-event interactions; ad hoc widgets can contain other widgets and data representations (e.g. shapes or the handlers to scale shapes in the drawing area). Moreover, as previously explained, four elements are involved in the typical GUI testing process: the GUI oracle; the test model; the language used to build GUI models; the model creation process. In this section we explain how current GUI and test models hinder the ability to test advanced GUIs. Current GUI models are not expressive enough. Languages used to build GUI models, called UIDLs, are a corner-stone in the testing process. Their expressiveness has a direct impact on the concepts that can compose a GUI test model (e.g. an EFG). That has, therefore, an impact on the ability of generated GUI test cases to detect various GUI errors. For instance, GUITAR uses its own UIDL that captures GUI structures (the widgets that compose a GUI and their layout).However, in the current trend of developing highly interactive GUIs that use ad hoc widgets, current UIDLs used to test GUIs are no longer expressive enough. First, UIDLs currently used for GUIs testing describe the widgets but not how to interact with them.

The reason behind this choice is that current GUI testing frameworks test standard widgets, which behavior is the same in many GUI platforms. For instance, buttons work by pressing on it using a pointing device on all GUI platforms. This choice is no more adapted for advanced GUIs that rely more and more on ad hoc widgets and interactions. Indeed, the behavior of these tests has been developed specially for a GUI and is thus not standard. As depicted in Figure 2, GUITAR embeds the definition of how to interact with widgets directly in the Java code of the framework. Test scripts notify the framework of the widgets to use on the SUT. The framework uses its widgets definitions to interact with them.

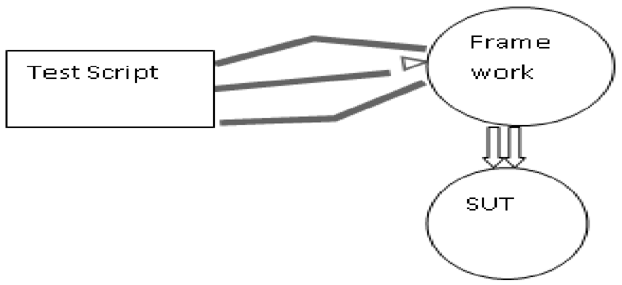

Figure 3: Representation of how interactions are currently managed and the current limit

So, supporting a new widget implies extending the framework. Even in this case, if users can interact with a widget using different interactions the framework randomly selects one of the possible interactions. Thus, the choice of the interaction to use must be clearly specified in GUI models. That will permit to generate a test model (e.g. an EFG) that can explore all the possible interactions instead of a single one. UIDLs must be expressive enough for expressing such interactions in GUI models.

V. MODEL BASED TESTING

A generic process of model-based testing then proceeds as follows (Fig. 4).
Step 1: A model of the SUT is built on the grounds of requirements or existing specification documents. This model encodes the intended behavior, and it can reside at various levels of abstraction. The most abstract variant maps each possible input to the output "no exception" or "no crash". It can also be abstract in that it neglects certain functionality, or disregards certain quality-of-service attributes such as timing or security.

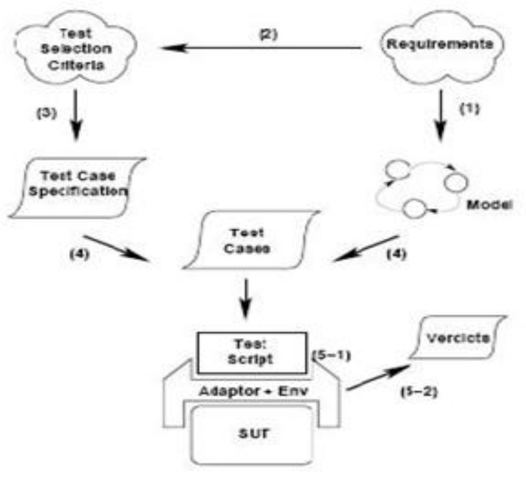

Figure 4: Model based testing

Step 2: Test selection criteria are defined. In general, it is difficult to define a "good test case" a-priori. Arguably, a good test case is one that is likely to detect severe and likely failures at an acceptable cost, and that is helpful the identifying the underlying fault. Unfortunately, this definition is not constructive. Test selection criteria try to approximate this notion by choosing a subset of behaviors of the model. A test selection criterion possibly informally describes a test suite. In general, test selection criteria can relate to a given functionality of the system (requirements based test selection criteria), to the structure of the model (state coverage, transition coverage, def-use coverage), to stochastic characterizations such as pure randomness or user profiles, and they can also relate to a well-defined set of faults.

Step 3: Test selection criteria are then transformed into test case specifications. Test case specifications formalize the notion of test selection criteria and render them operational: given a model and a test case specification, some automatic test case generator must be capable of deriving a test suite (see step 4). For instance, "state coverage" would translate into statements of the form "reach _" for all states _ of the (finite) state space, plus possibly further constraints on the length and number of the test cases. Each of these statements is one test case specification. The difference between a test case specification and a test suite is that the former is intensional ("fruit") while the latter is extensional ("apples, oranges,") all tests are explicitly enumerated.

Step 4: Once the model and the test case specification are defined, a test suite is generated. The set of test cases that satisfy a test case specification can be empty. Usually, however, there are many test cases that satisfy it. Test case generators then tend to pick some at random.

Step 5: Once the test suite has been generated, the test cases are run (sometimes, in particular inthe context of non-deterministic systems, generating and running tests are dove-tailed).

Running a test case includes two stages.
Step 5-1
Recall that model and SUT reside at different levels of abstraction, and that these different levels must be bridged [2]. Executing a test case then denotes the activity of applying the concretized input part of a test case to the SUT and recording the SUT's output. Concretization of the input part of a test case is performed by a component called the adaptor. The adaptor also takes care of abstracting the output (see Fig 3).

Step 5-2
A verdict is the result of the comparison of the output of the SUT with the expected output as provided by the test case. To this end, the output of the SUT must have been abstracted. Consider the example of testing a chip card that can compute digital signatures [7]. The verdict can take the outcomes pass, fail, and inconclusive. A test passes if expected and actual output conforms. It fails if they do not, and it is inconclusive when this decision cannot be made.

VI. IMPORTANCE OF MBT

The first obstacle to overcome in developing tests is to determine the test target. While this may sound trivial, it is often the first place things go wrong. A description of the product or application to be tested is essential. The form the description can come in may vary from a set of call flow graphs for a voice mail system, to the user guide for a billing system's GUI. A defined set of features and / or behaviors of a product is needed in order to define the scope of the work (both development and test). The traditional means of specifying the correct system behavior is with English prose in the form of a Requirement Specification or Functional Specification [1]. The specification, when in prose, is often incomplete - only the typical or ideal use of the feature(s) is defined, not all of the possible actions or use scenarios. This incomplete description forces the test engineer to wait until the system is delivered so that the entire context of the feature is known. When the complete context is understood, tests can be developed that will verify all of the possible remaining scenarios. Another problem with textual descriptions is that they are ambiguous, (for example "if an invalid digit is entered, it shall be handled appropriately.") The 'appropriate' action is never defined; rather, it is left to the reader's interpretation.

VII. MODELING BEHAVIOR TO GENERATE FUNCTIONAL TESTS AND OMPONENT TEST THROUGH THE USER INTERFACE

The Functional Requirements Specification and process flow documents define the behavior of the Application under Test (AUT). The AUT in this case is a forms-based system that builds a record and submits a transaction. The documents contain functional descriptions of the application as well as detailed data descriptions. The specifications are used at the highest level to determine valid use scenarios for processing the forms. At the lowest level they determine valid inputs and expected outputs for each field in a given transaction as well as error message descriptions. The purpose of testing each component proves the quality of the component and the overall performance of the system. If any component used in the software product fails to perform its tasks then it can lead to the overall software product failure [6].The model based approach captures this detail as well has the behavior represented by the specifications. The model is processed in order to produce the executable test files required by the Test Environment. The work order application is very straightforward to test. A simple scripting language is used to traverse from field to field and form to form. Elements of the scripting language are embedded on each transition in the model in such a way that when the model is processed, the model defines a test of the behavior of the application. For each valid flow (or path) derived from the model, a new test is created. These paths include the operational scenarios specified in the requirements specification as well as other valuable test scenarios.

VIII. VIII. INDUSTRY IMPORTANCE

Modeling is a very economical means of capturing knowledge about a system and then reusing this knowledge as the system grows. For a testing team, this information is gold; what percentage of a test engineer's task is spent trying to understand what the System Under Test (SUT) should be doing? (Not just is doing.) Once this information is understood, how is it preserved for the next engineer, the next release, or change order? If you are lucky it is in the test plan, but more typically buried in a test script or just lost, waiting to be rediscovered. By constructing a model of a system that defines the systems desired behavior for specified inputs to it, a team now has a mechanism for a structured analysis of the system. Scenarios are described as a sequence of actions to the system, with the correct responses of the system also being specified. Test coverage is understood and test plans are developed in the context of the SUT, the resources available and the coverage that can be delivered. The largest benefit is in reuse; all of this work is not lost. The next test cycle can start where this one left off. If the product has new features, they can be incrementally added to the model; if the quality must be improved, the model can be improved and the tests expanded; if there are new people on the team, they can quickly come up to speed by reviewing the model.

The increased complexity of systems as well as short product release schedules makes the task of testing challenging. One of the key problems is that testing typically comes late in the project release cycle, and traditional testing is performed manually. When bugs are detected, the cost of rework and additional regression testing is costly and further impacts the product release. The increased complexity of today’s software-intensive systems means that there are a potentially indefinite number of combinations of inputs and events that result in distinct system outputs and many of these combinations are often not covered by manual testing. We work with companies that have high process maturity levels, and excellent measurement data that shows that testing is more 50-75% of the total cost of a product release, yet these mature processes are not addressing this costly issue. Testtools may not replace human intelligence in testing, but without them testing complex systems at a reasonable cost will never be possible. There are commercial products to support automated testing, most based on capture/playback mechanisms, and organizations that have tried these tools quickly realize that these approaches are still manually intensive and difficult to maintain. Even small changes to the application functionality or GUI can render a captured test session useless. But more importantly, these tools don't help test organizations figure out what tests to write, nor do they give any information about test coverage of the functionality.

IX. CONCLUSION

The efforts spent on testing are enormous due to the continuing quest for better software quality, and the ever growing complexity of software systems. The situation is aggravated by the fact that the complexity of testing tends to grow faster than the complexity of the systems being tested, in the worst case even exponentially. Whereas development and construction methods for software allow the building of ever larger and more complex systems, there is a real danger that testing methods cannot keep pace with construction, hence these new systems cannot be sufficiently fast and thoroughly be tested. This may seriously hamper the development of future generations of software systems. One of the new technologies to meet the challenges imposed on software testing is model-based testing. Models can be utilized in many ways throughout the product life-cycle, including: improved quality of specifications, code generation, reliability analysis, and test generation. In the component based software development different already build components are used. To ensure the quality of such product component testing is very essential. But problems may arise for the tester in component testing phase due to the limited access of the component. In this paper component based testing & model based testing strategies are discussed and using a good strategy according to situation quality of the product can be improved. If best suitable practices are applied on new component based software it become more reliable and chances of failure become very less. Testing should be performed on every component and whole software product before delivering it. This paper will focus on the testing benefits from Model based testing methods and component based testing to share our experiences from a long term industrial evaluation on automatically extracting models of GUI applications and utilizing the extracted models to automate and support functional ,Component & GUI testing.

References

[1] IEEE standard for Requirements Specification (IEEE/ANSI Std. 830-1984) , IEEE Computer Society, (830-1993) IEEE Recommended Practice for Software Requirements Specifications (ANSI), IEEE Standard for Software Unit Testing (ANSI), IEEE Standard for Software Verification and Validation Plans(ANSI)found at: http://standards.ieee.org/catalog/it.html

[2] Proceedings of the Third IEEE International Symposium on Requirements Engineering IEEE Computer Society, 1997.

[3] Paulk, M., Curtis, B., Chrissis, M.B., and Weber, C., Capability Maturity Model, Version 1.1 The Software Engineering Institute, Carnegie MellonUniversity.Foundat: http://www.sei.cmu.edu/products/publications/96.reports/96.ar.cmm.v1.1.html

[4] T. Yue, S. Ali, and L. Briand. Automated transition from use cases to UML state machines to support state-based testing. In Proc. of ECMFA'11, pages 115–131. Springer-Verlag, 2011.

[5] X. Yuan, M. B. Cohen, and A. M. Memon. GUI interaction testing: Incorporating event context. IEEE Trans. Software Eng., 37(4):559–574, 2011.

[6] D.Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. Using GUI ripping for automated testing of android applications. In Proc. of ASE'12, pages 258–261, 2012.

[7] S.Arlt,P. Borromeo, M. Schäf, and A. Podelski.Parameterized GUI tests. In Proc of Testing Software and Systems, pages 247–262. 2012.

[8] A. Holmes and M. Kellogg, "Automating functional tests using selenium,"in Agile Conference, 2006, July 2006, pp. 6 pp.–275.

[9] B. Nguyen, B. Robbins, I. Banerjee, and A. Memon, "Guitar:an innovative tool for automated testing of gui-driven software," Automated Software Engineering,pp.1–41,2013.[Online].available: http://dx.doi.org/10.1007/s105 15-013- 0128-9

[10] F. Gross, G. Fraser, and A. Zeller, "Search-based System testing: High coverage, no false alarms," in Proceedings of the 2012 International Symposium on Software Testing and Analysis, ser. ISSTA '12. New York, NY, USA: ACM, 2012, pp. 67–77.

[11] G. Fraser and A. Arcuri, "Whole test suite generation," IEEE Transactions on Software Engineering, vol. 39, no. 2,
pp. 276–291, 2013.

[12] K. Sen, D. Marinov, and G. Agha, "Cute: a concolic unit testing engine for c," in Proceedings of the 10th European software engineering conference held jointly with 13th ACMIGSOFT international symposium on Foundations of software engineering, ser.SEC/FSE-13.New York, NY, USA: ACM, 2005,pp.263–272.[Online].

[13] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, ser. PLDI '05.New York, NY,USA:ACM,2005,pp.213–223. online].Available:http://doi.acm.org/1 45/1065010.1065036

[14] J. P. Galeotti, G. Fraser, and A. Arcuri, "Improving search- based test suite generation with dynamic symbolic execution," in 2013 IEEE 24th International symposium on Software Reliability Engineering (ISSRE),2013, pp. 360– 369.

[15] Weiqun Zheng,"Model-Based Software Component Testing", 2012.