Identifying Error-Prone Transactions in Enterprise Applications

Pavan Kumar Chittimalli, Sachin Patel, Vipul Shah
TCS Innovation Labs,
Tata Consultancy Services Limited,
Pune, India.
Email: {pavan.chittimalli, sachin.patel, v.shah}@tcs.com

***Abstract*—Independent testing teams use requirements as the basis to develop test cases and automated test scripts. The projects are executed under severe schedule constraints, due to which, the testers have to focus their testing efforts on error-prone and important features. Numerous source code based techniques for identifying error-prone features/components have been developed. However, they are based on source code analysis. Independent testing teams rarely have access to source code and they find it difficult to use code based techniques. In many cases, the domain experts use Business Process Model and Notation (BPMN) to represent the business requirements. In this paper, we propose an approach to identify error-prone transactions in enterprise applications using a BPMN. It helps in distinguishing between source code errors and test script errors. We have adapted this approach from an existing source code based technique. Our experiments with the approach show that it can identify the location of actual as well as seeded errors in both source code and test scripts.**

Keywords–Enterprise Application testing; BPMN; Stastical Bug Isolation; Bug Localization

I. INTRODUCTION

Business systems evolve due to various reasons such as correction of errors, adding new features, migrating to new environments, and improving performance. These changes may introduce infections [1], which propagate as the failure of the test-case. Testers face severe schedule constraints and they would like to spend their time on testing error-prone and important features. This necessitates the use of prioritization and fault localization techniques. There have been several fault localization techniques [1][2][3][4], proposed based on coverage information of the program entities and test executions logs. But teams which provide testing services do not have access to the source code. The testing team gets requirements in natural language or sometimes in formal notations like BPMN [5]. They use these as the basis to develop test cases and automated scripts [6]. In such scenarios, code based fault localization techniques cannot be used. The automated test scripts developed by testers are another source of error. It is difficult to differentiate between a test script failure and a source code failure. The manual trace analysis to identify test script errors, takes considerable amount of time and requires domain as well as technical expertise. This motivates us to develop techniques [7] for identifying error-prone features and test scripts of an application.

A. Motivating example

Listed below are two requirements of a billing application. The business transaction assumes to create an order and generate an invoice for it.

R1: If there *Exists Promotions* then apply the discount and generate invoice. If there *Exists No Promotion* then generate invoice without discount.

R2: In case of *Full Payment*, pay the generated invoice amount. In case of *Partial Payment*, pay amount less than generated invoice amount.

TABLE I. THE SAMPLE TEST-SCRIPTS FOR THE BILLING APPLICATION

#	Test Sequence
T_1	1) Login 2) Create an order 3) If (promotions == 1) { apply discount generate invoice } 4) If (Payment Option==1) pay the amount 5) Logout
T_2	1) Login 2) Create an order 3) If (promotions == 1) { apply discount generate invoice } 4) If (Payment Option==2) pay the partial amount 5) Logout
T_3	1) Login 2) Create an order 3) If (promotions == 3) { generate invoice } 4) If (Payment Option==1) pay the amount 5) Logout
T_4	1) Login 2) Create an order 3) If (promotions == 3) { generate invoice } 4) If (Payment Option==2) pay the partial amount 5) Logout

The tester has identified four test cases from these requirements. See test case T_1 in Table 1. The first step is a *Login* with customer details like name and password. In the second step, *Create an Order*, displays the list of items to choose. The user selects items from the specified list and creates an order. In the third step, he checks for the option of any existing promotions (i.e., *promotions* $==$ *1*). The corresponding discounts are applied to the items ordered. Payment is done in the fourth step. If the payment option is *full payment* (i.e., *payment option* $==$ *1*) then pay the full amount and generate the invoice accordingly. The last step is a *Logout* event, which terminates the user session. Similarly, the other test-cases T_2, T_3, and T_4 are written as shown in Table 1 and executed with corresponding test-data. The execution results in successful execution (*pass*) for test-cases T_1 and failed execution (*fail*) for test-cases T_2, T_3, T_4. The reason for the failure of test-cases T_3 and T_4 is a script error at step 3. The script was checking for *promotions* $==$ *3* instead of *promotions* $==$ *2*, which resulted in a failure of the test-case. The test-case T_2 is failed because of source-code error in processing partial payment task.

Analyzing such errors requires lot of effort and domain and technical expertise. In this paper, we describe our technique of adopting the existing source code based fault localization techniques to a model based representation of the system - BPMN. In Section 2, we describe the approach for fault localization, followed by a description of two experimental studies in Section 3. We conclude the paper with a discussion of future work in Section 4.

II. Our Approach

We choose BPMN as a representation for functional requirements of the application. In this section, we first describe the BPMN using our sample example and the later part of the section will give the details of our approach based on this representation. Our approach tries to address the following research questions: 1) Can we adopt source code based fault localization techniques to BPMN model entities? 2) Can we distinguish between a script error and source code error?

The following subsections gives details about our approach.

A. Business Process Model

In BPMN terminology, a business process P is defined as: $P =< PE, F, s, E >$. The process element (PE) in BPMN representation can be a *task*, *gateway*, or a *subprocess*. A *task* is used for defining a particular activity. The *gateway* is used for decision making where each flow edge out of *gateway* has a condition associated with it. There are *or*, *nor*, *and*, *xor* variants of *gateway* exists as a representation. A *subprocess* is a place-holder or callee point for another business process. A *flow element* (an item of F) is an edge between two process elements. s is the *start* element. An *end* can be normal end of the process in which the return edge to callee exists. But in the *terminate end* the called process never returns to callee and ends the flow at that point.

For example, Figure 1 is a BPMN representation of the test cases shown in Table 1.

B. Test case, Test script generation using BPMN

A scenario (s_i) in the process diagram is defined as a path pa_i from start node s to end node e where $e \in E$. A path is a sequence of process elements (pe_i) with flow elements f_i in between each of those process elements defines a scenario s_i. Kholkar et al. [8] have proposed automating functional testing using a BPMN representation of the business application. We augment the standard BPMN representation with *pre* and *post* test conditions to specify test conditions and assertions. This results in a set of valid scenarios C for a process representation. For each such valid scenario, a test-case T_i is generated along with the scenario. For example, consider BPMN shown in Figure 1 for the illustration. The model has four feasible scenarios $\{s_1, s_2, s_3, s_4\}$ resulting in four unique test-cases $\{T_1, T_2, T_3, T_4\}$. The generated scenarios are shown in the following Table 2.

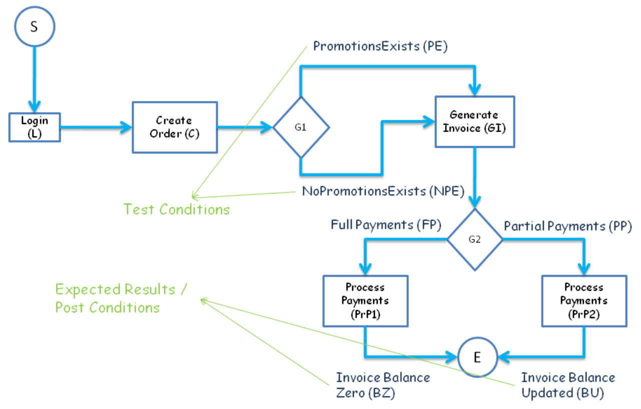

Figure 1. Annotate the billing application using functional requirements.

TABLE II. The scenarios for the example in Figure 1.

Id	Test case	Scenario
s_1	T_1	S → L → C → NPE → GI → FP → PrP1 → BZ → E
s_2	T_2	S → L → C → NPE → GI → PP → PrP2 → BU → E
s_3	T_3	S → L → C → PE → GI → FP → PrP1 → BZ → E
s_4	T_4	S → L → C → PE → GI → PP → PrP2 → BU → E

The test-case T_1 is generated for the scenario s_1 depicting a scenario - "*A registered user can create an order where there no promotions exists, and generate an invoice with full payment mode*". The same approach is described in our test automation tool [9]. This end-to-end script generation using BPMN representation of process diagrams are used in our approach for test-script generation.

C. Test execution and Traceability matrix building

The test automation tool in [9] is capable of capturing architectural, user interface, behavioral, and data models. This test automation tool records execution sequences at *entity* level for our process diagrams. The executions of these entities are then mapped to the corresponding test-cases using the execution traceability matrix. The empty cell in the traceability matrix indicates that the test-case does not execute the entity during the execution. The entry with a dark circle in the traceability matrix indicates that the entity has been executed during the test-case execution. The captured traceability matrix can be used in various regression testing and debugging activities [1][10][11]. The execution summary will result in either success or failure of the test-case. This execution status report (pass / fail information) and traceability matrix for the example in Figure 1 is shown in Table 3.

D. Identifying error-prone transactions

During the execution of test-cases on the system results in some failure and some successful executions. The cause of the failure can not be located by looking only into the failure test-cases [12]. In this subsection, we describe the adaptation of two source-code based fault localization techniques for use with BPMN. We first used Tarantula, a fault localization technique, invented by Jim Jones et al. [1][11][12]. Tarantula utilizes the pass / fail status of the test-case and the entities executed by each of the test-case. The other fault localization technique we used was Statistical Bug Isolation (SBI), a Liblit et al. [2]. In our approach, we adapted Tarantula, SBI and extended them to apply its metrics to entities in a BPMN model. Tarantula has two metrics *suspiciousness* and *confidence* to locate the error-prone entities in source code and *hue* and *brightness* to visually locate them. SBI has a metric called *Failure* to locate the error-prone entities in the source code. The *suspiciousness* of an entity *e* is defined as the level of being faulty that caused the failed test cases to fail. The value of suspiciousness metric ranges from '0' to '1', where '0', being least suspicious and '1' being high suspicious. Given a test-suite T, the suspiciousness metric for an entity *e* in Tarantula is defined as:

$$\begin{aligned} suspiciousness(e) &= \frac{\frac{failed(e)}{\#failed}}{\frac{passed(e)}{\#passed} + \frac{failed(e)}{\#failed}} \\ &= \frac{\%failed(e)}{\%passed(e) + \%failed(e)} \end{aligned} \quad (1)$$

In (1), $failed(e)$ represents the number of failed test-cases in T that have been executed by the entity e and $passed(e)$ represents the number of passed test-cases in T that have been executed by the entity e. $\#failed$ represents the total number failed test-cases and $\#passed$ represents the total number of passed test-cases in the test-suite T.The confidence metric is defined to state the confidence of the suspiciousness of the coverage *entity* that is being computed. The value of confidence ranges from '0' to '1' where '0' represents the least confidence and '1' represents the highest confidence, to the suspiciousness value. The *confidence* metric of entity *e* is defined as:

$$\begin{aligned} confidence(e) &= max\left(\frac{passed(e)}{\#passed}, \frac{failed(e)}{\#failed}\right) \\ &= max\left(\frac{\%passed(e)}{100}, \frac{\%failed(e)}{100}\right) \end{aligned} \quad (2)$$

In (2), the variables are same as in (1). The *max* takes the maximum value of fail/pass information available at that entity.

The $Failure$ of predicate *P* is defined as the probability of an atomic predicate (P) is true for failing runs and false for successful runs (i.e., $Pr(Crash|P\ observed\ to\ be\ true)$).

$$Failure(P) = \frac{F(P)}{S(P) + F(P)} \quad (3)$$

The $Failure(P)$ is expressed in the above equation where $S(P)$ denotes the number of successful runs in which *P* is observed true, and $F(P)$ denotes the failing runs in which P is observed to be true.

For example, consider in the process model defined in Figure 1. The test scenarios and test data are generated from annotated business process models [8]. The test script generation tool [13] is capable of capture and reply of the application. It records the coverage information of the entities in process model, which is used to build the traceability

TABLE III. TRACEABILITY MATRIX FOR THE TARANTULA TECHNIQUE

Entity Name	T_1	T_2	T_3	T_4	*suspiciousness*	*confidence*	*Failure*
$Start\ (S)$	●	●	●	●	.5	1	–
$Login\ (L)$	●	●	●	●	.5	1	–
$CreateOrder\ (C)$	●	●	●	●	.5	1	–
$gateway\ (G1)$	●	●	●	●	.5	1	–
$NoPromotionsExist(NPE)$			●	●	1	.6	.7
$PromotionsExist\ (PE)$	●	●			.25	1	.3
$GenerateInvoice\ (GI)$	●	●	●	●	.5	1	–
$gateway\ (G2)$	●	●	●	●	.5	1	–
$FullPayment\ (FP)$	●		●		.25	1	.3
$PartialPayment(PP)$		●		●	1	.6	.7
$ProcessPayment\ (PrP1)$	●		●		.25	1	–
$ProcessPayment\ (PrP2)$		●		●	1	.6	–
$BalanceZero\ (BZ)$	●		●		.25	1	–
$BalanceUpdate\ (BU)$		●		●	1	.6	–
$End\ (E)$	●	●	●	●	.5	1	–
Execution Status	**P** ✓	**F** ×	**F** ×	**F** ×			

matrix. The matrix is used to calculate the suspiciousness and confidence metrics. See the Table 3. In this case, test-case $\{T_1\}$ has passed whereas $\{T_2, T_3, T_4\}$ have failed. Using the coverage information and pass / fail information, the metrics *suspiciousness* and *confidence* have been computed. The entity *Start (S)* is executed by T_1 (pass) , T_2 (fail), T_3 (fail), T_4 (fail). Using the Tarantula approach, we calculated the corresponding *suspiciousness* for the *Start* entity as 0.5 and *confidence* as 1. Similarly, Using SBI approach, we computed the metrics for all other entities in the sample billing application. The rows 5, 10, 12, 14 in the Table 3 have the highest suspicious value '1'. For entity $NoPromotionsExist$ (NPE) (shown as suspicious in row 5) has been written wrongly in the test-cases T_3 and T_4 and categorized as test-script fault. The test-conditions $PartialPayments$ (in row 10) and $BalanceUpdate$ (in row 14) are with highest suspiciousness value (1) with a confidence value of 1. But the conditions do not have any faults so they are not classified as errors and $task\ processPayment$ $(PrP2)$ (in row 12) is categorized as source code error in implementation of processing the payments. Similarly the rows 5 and 10 show the highest $Failure$ (SBI metric) as '.7' in failing predicates. The first predicate is failed due to test-script error and second failed due to source-code error in processing payments.

While the metrics help in identifying the error-prone BPMN entities, a visual representation would make it much easier to locate [12]. To achieve this feature, we used the color computing metric used in Tarantula. This technique uses Hue, Saturation, and Brightness (HSB) from *red* to *green* color range. We use *hue* metric to compute the color range specified in equations shown below. The *colorrange* is defined as 0.33.

$$\begin{aligned} hue(e) &= 1 - suspiciousness(e) \\ &= \frac{\%passed(e)}{\%passed(e) + \%failed(e)} \end{aligned} \quad (4)$$

$$color(e) = color(red) + hue(e) * colorrange \quad (5)$$

For example, see the Figure 2. The entity colored in *red* (*NoPromotionsExist*) is a highest suspicious entity. Analysis of the BPMN and test scripts reveal that it is a test script error. Instead of writing a test condition as $promotions == 2$, the condition has been mentioned as $promotions == 3$. The test-script and source code faults are shown in clouded area in Figure 2.

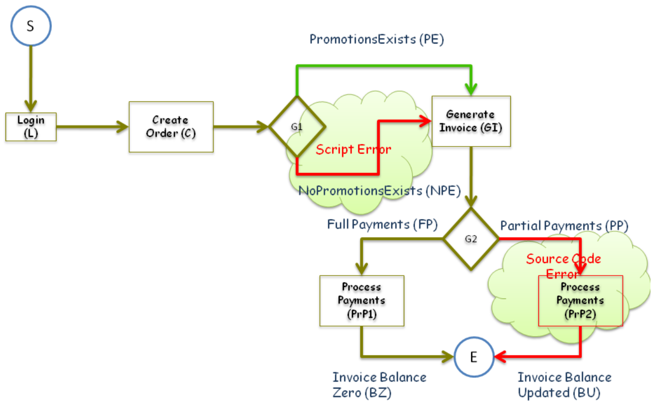

Figure 2. Using Tarantula visualization technique on billing application given in Figure 1.

The other suspicious entities shown in Figure 2 are *PartialPayments(PP)*, *InvoiceBalanceUpdated(IBU)*, *ProcessPayments(PrP2)*. On the careful observation of entities *PP* and *IBU*, we found that there is no script error exists. Hence the suspicious *task PrP2* is considered as source code error.

III. EXPERIMENTAL STUDIES

We evaluated the Tarantula [1] and Bin Liblit's approach [2] of SBI technique (Now, Co-operative Bug Isolation) along with our test generation toolset [8][13][9]. The systems under test were two open source enterprise applications, Jbilling and Mercury. Jbilling is an open-source billing system. We have configured it for a hardware construction material business. The process model for this application has 10 processes and 108 entities. 30 test-cases have been identified for Jbilling. The Mercury application is a online flight reservation system. The process model for this application has 3 processes and 29 entities. 13 test-cases have been identified for Mercury. The objective of our study is to 1) to locate parts of the business process that are error-prone 2) to distinguish between a source-code error and test-script error.

A. Study-1

We executed the identified test cases on the two applications. The results of the execution are shown in Table 4. In Table 4, the first column represents the subject. The second column shows the number of failures detected by Tarantula and SBI techniques. The third column shows the number of code failures detected by Tarantula and SBI. The fourth column shows the test-script failures detected by Tarantula and SBI respectively. The color coding provided by the tool helps locate the errors in source code, as well as test scripts. A red colored edge represents a test script error as this transition is caused by the test script and not the source code. If a fault is not test-script error then, we conclude it as a source-code error and point to the corresponding *task* in the process model. We located one such fault in the Jbilling application. Consider the first row in Table 4 for Jbilling subject. Tarantula has detected 6 faults, of which, 5 failures are source code failures and 1 failure is a test-script failure. Similarly, SBI has detected the same for Jbilling application.

TABLE IV. THE DETAILS OF THE STUDY-1 IN EXPERIMENTATION.

Subject	Total Failures		Code Failures		Test-script Failures	
	Tarantula	SBI	Tarantula	SBI	Tarantula	SBI
$Jbilling$	6	6	1	1	5	5
$Mercury$	3	3	0	0	3	3

B. Study-2

In this study, we used seeded faults by generating more test scenarios. These additional seeded faults are created by mutating the operators in the process flow conditions of the process diagrams. The edges (Flow Elements) in BPMN are associated with the flow conditions. We have selectively taken these conditions for seeding. We applied operator mutation on relational operators $(>, <, \geq, \leq)$, equality operators $(=, \neq)$. We modified the test scenario generation described in section 2-B to address this. The objective of this study is to see if the mutants are killed or caught by the test scripts. We observed that all mutants have been caught as shown in the Table 5.

IV. RELATED WORK

Most fault localization techniques in the literature have been based on code coverage. The common method has been to compare the coverage of failure runs and passing runs to determine the location of the faults. Jim Jones et al. [1][11][12] have done extensive research in the field of fault localization based on coverage of failure and passing runs. Their tool *Tarantula* uses the coverage information of entities at statement level to compute *suspiciousness*, *confidence*, *hue*, and *color* metrics. This tool is capable of showing the faults using visualization. But this tool was developed to locate source code faults. Liblit et al. [2] have proposed fault localization based on the coverage of predicates in failing and passing runs by sampling failure predicates. This is a lightweight technique as it uses very little program instrumentation compared to the *Tarantula* technique. Tarantula technique is more useful in in-house debugging whereas the SBI technique can be used in field debugging. Zeller et al. [4] have proposed a light weight instrumentation technique to capture the method call sequence coverage for locating the faults in java programs. Comparing the object-specific sequences predicts the defects better than just simply comparing the coverage. Naoya Maruyama and Satoshi Matsuoka [3] have proposed a fault localization technique in large computing systems using traces which capture function calls. They derive a model from the traces and compare them with failure traces to find the defect and computes suspect score to that failure.

In practice, the functional test teams carry out system and regression tests as independent test teams, treating the systems as a black box. Test teams prepare test plans and test scenarios from functional requirements that are available informally in natural language or sometimes semi-formally in notations like

TABLE V. THE DETAILS OF THE STUDY-2 IN EXPERIMENTATION.

Sub.	**No.of Test cases**	**Total Failures**		**Code Failures**		**Test-script Failures**		**Mutants Caught**	
		Tar	SBI	Tar	SBI	Tar	SBI	Tar	SBI
1	50	26	26	1	1	25	25	20	20
2	21	3	3	0	0	11	11	8	8

BPMN. Test scripts are manually or automatically generated from such models. The above fault localization techniques [1][11][12] that take into account coverage information of the program entities and test executions logs have been proposed. Independent test teams however do not have access to code nor the knowledge of the code to understand and interpret the results provided by current techniques. One of the additional challenges faced by the test teams, especially during the first test run in each release, is that the new test scripts may be faulty, or older test scripts may become out of sync with the requirements. A significant amount of time and effort is spent to determine if the faults are in the test scripts or source code. Further, with the advances happening in model-based testing, it is necessary to investigate if the code-based techniques developed so far have an utility in the model-based world. The work done in this paper is one such exploration.

V. CONCLUSIONS AND FUTURE WORK

The techniques applied in this paper have been extensively used with source code entities. We have applied them for a non-executable, model based representation. In this paper, we proposed BPMN as a system representation and extended the existing Tarantula, SBI techniques to identify error-prone transactions in an enterprise application. We also used Tarantula's visualization metric to locate faults in BPMN representation of the system.

Our preliminary experiments on in-house examples and openly available subjects showed encouraging results and caught all script and source code errors. These results have been manually verified. However, we have not applied this approach on a real-time project. A dependence fault, which appears only after fixing root faults, is not handled in current approach. Our assumption for locating source code fault has not verified in presence of dependence faults. We would like to apply this technique on bigger and more complex systems. Another possibility is to use test execution history to guide the test selection. Further studies will be required to understand the relationship between code-based and model-based metrics.

REFERENCES

[1] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, ser. ASE '05. New York, NY, USA: ACM, 2005, pp. 273–282.

[2] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 15–26.

[3] N. Maruyama and S. Matsuoka, "Model-based fault localization in large-scale computing systems," in Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on, April 2008, pp. 1–12.

[4] V. Dallmeier, C. Lindig, and A. Zeller, "Lightweight defect localization for java," in Proceedings of the 19th European Conference on Object-Oriented Programming, ser. ECOOP'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 528–550.

[5] "Business Process Model And Notation (BPMN)," http://www.omg.org/spec/BPMN/, [Online; accessed 04-July-2015].

[6] Q. Yuan, J. Wu, C. Liu, and L. Zhang, "A model driven approach toward business process test case generation," in Web Site Evolution, 2008. WSE 2008. 10th International Symposium on, Oct 2008, pp. 41–44.

[7] P. K. Chittimalli and V. Shah, "Fault localization during system testing," in Proceedings of International Conference on Program Comprehension (ICPC), May 2015.

[8] D. Kholkar, N. Goenka, and P. Gupta, "Automating functional testing using business process flows," in Proceedings of Workshop on Advances in Model-Based Software Engineering, ser. ISEC (2011), 2011, pp. 102–110.

[9] S. Patel, P. Gupta, and P. Surve, "Testdrive - A cost effective way to create and maintain test scripts for web applications," in Proceedings of the 22nd International Conference on Software Engineering & Knowledge Engineering (SEKE'2010), Redwood City, San Francisco Bay, CA, USA, July 1 - July 3, 2010, 2010, pp. 474–476.

[10] P. K. Chittimalli and M. J. Harrold, "Regression test selection on system requirements," in Proceedings of the 1st India Software Engineering Conference, ser. ISEC '08. New York, NY, USA: ACM, 2008, pp. 87–96.

[11] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in Proceedings of the 24th International Conference on Software Engineering, ser. ICSE '02. New York, NY, USA: ACM, 2002, pp. 467–477.

[12] J. A. Jones, "Semi-automatic fault localization," Ph.D. dissertation, Georgia Institute of Technology, Atlanta, Georgia, USA, April 2008.

[13] P. Gupta and P. Surve, "Model based approach to assist test case creation, execution, and maintenance for test automation," in Proceedings of the First International Workshop on End-to-End Test Script Engineering, ser. ETSE '11. New York, NY, USA: ACM, 2011, pp. 1–7.