The Proceedings of the Ninth International Research Workshop on Advances and Innovations in Software Testing

The University of Memphis
Systems Testing Excellence Program
FedEx Institute of Technology

The workshop is organized by the:

Systems Testing Excellence Program - STEP (http://www.memphis.edu/step/) at the University of Memphis. STEP is an interdisciplinary research program under the auspices of the FedEx Institute of Technology that brings together the departments of Business Information and Technology (formerly MIS), Computer Science, and Electrical & Computer Engineering.

http://www.memphis.edu/step/
October 19-20, 2015
Memphis, Tennessee
Abstract—Nowadays, software testing is becoming more critical to ensure that software will function properly in the production environment. Consequently, the effort, time, and fund invested on software testing activities increase significantly. As such, managers have to allocate the limited resources of an organization to those critical test cases rather than the exhausted set of test cases. This paper builds a value function for quantifying the unit value of a test case, which uses the value concept, presents as a quantified index, and can be used to single out the key test cases. We take three steps to establish the function. Firstly, we built an initial value function based on our understanding and knowledge as well as the relevant research. Next, we interviewed professionals and managerial staff who are working in the testing department or relevant positions to comment on the initial value function. Finally, based on the feedback of the interviews, we incorporated new factors emerged from the interviews and modified some other factors in the initial function which have different meanings according to the viewpoints of the interviewees. Through these steps, we established the final function, which includes tangible factors such as cost of a test case and amount of use, as well as intangible factors such as internal risk, external risk, and technical risk.

Keywords—unit value; functional test; business risk; technical risk; testing utility; testing frequency; unit cost

1. INTRODUCTION: THE VALUE OF TEST CASES

As a primary component of information technology, software development leads to organizational investments of billions of dollars on software development every year in United States, alone. To that end, technology development and support is the largest corporate expense outside employee costs [1, 2]. Surprisingly, as demonstrated in Table I, about 30% of the total investments in software development are consumed in the software testing phase [3, 4]. In view of the vagaries of requirements determination, the subsequent complexity of designing and writing code, and the translation issues involved in moving from one stage of software development to the next, testing software to make sure that it does what it was intended to do is a necessary and critical part of the development process [4, 5].

<table>
<thead>
<tr>
<th>Requirement Analysis</th>
<th>Preliminary Design</th>
<th>Detailed Design</th>
<th>Coding and Unit Testing</th>
<th>Integration and Test</th>
<th>System Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>1960s - 1970s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1980s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1990s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Source: M. Andersson, and J. Bergstrand, “Formalizing use cases with message sequence charts,” unpublished.

In practice, comprehensive and exhaustive testing regimes are difficult to achieve owing to limitations of time, funding and staff. At the same time, the Pareto Principle of 80% of value arising from 20% of software components [6] suggests that the determination of an optimal testing technique drawn from the extensive list of available testing approaches is not an easy task. In order to make the testing technique selection, managers take into account various factors in assembling an
optimal portfolio of test cases to arrive at maximal contribution. Testing as a support activity intertwined with other parts of the software life cycle cannot deliver significant contribution to the software development process unless the particular testing activities necessary for assessing high value software components are identified and implemented [7]. With that in mind, this paper employs the concept of “Unit Value” of testing in order to establish a mathematical function of software testing value useful for purposes of evaluating different tasting cases within an organization.

A. Contribution of the Study

The objective of this paper is to explore factors that play a significant role in impacting the value of functional test cases and how those factors can determine the relative value of a functional test case. Our contribution is threefold: first, the function we develop reveals the essence of value for a context-specific test case rather than a suite of test cases. In this way, individual assessment for optimization decision making is enhanced. Second, the test case value function fills a notable gap in the literature, as there currently exists no specific method to systematically determine and justify the value of test cases systematically in the context of functional test case analysis. Knowing this will likewise enhance the capabilities of software testing managers. Lastly, the specific exploration of the notion of test case establishes an important reference point for role of systems development in the critical corporate governance task of resource allocation.

B. Organization of the Article

This paper is organized into eight parts. Part 1 introduces software testing as currently practiced in industry. Part 2 describes relevant theories about value and risk, and the initial theorization on the factors influencing test case value. Part 3 delivers the initial unit value function, while part 4 documents an ongoing data collection process leading to finalized definitions of the value factors. Part 5 demonstrate the final form of the unit value function with its relevant operational mechanism and part 6 compares our emergent unit value function with another popular value determination rubric - return on investment. Part 7 provides guidance for the general application of the proffered unit value function, while part 8 addresses limitations of the function and offers future research directions for improving the model.

2. THEORETICAL BACKGROUND

A. The Nature of Value

Harvard’s Michael Porter [8] is well-known for his notions of value analysis. He mainly demonstrates the implication of value from the firm level, noting that it is the amount buyers are willing to pay for what a firm provides. Generic competitive strategies, then, revolve around creating value for buyers who are willing to pay more than the cost of providing that value. Although one of the basic competitive strategies is, indeed, cost-based, Porter is of the opinion that value rather that cost is the best factor to use in analyzing competitive position [8, 9]. Hass [7] gave another interpretation of the value in a software testing context. in testing level. He suggested that testing, itself, added value to the software development “product,” that organizations consumer, which is an intriguing concept. If testing increases the quality of the developed software, then it follows that value is positively impacted, so long as its costs are reasonable and do not exceed the perceived value construction calculus in the minds of the receiving constituent. This outcome obtains even if the product does not, in principle, change as a result of the testing process [7]. Even so, in the normal course of events there are generally discernable product improvement, decision improvement, and process improvement benefits that derive from the testing process, and this is well related in the expected calculus of the amount of an expected return or benefit beyond the committed cost.

B. Components of the Initial Unit Value Function

Based upon our development of the concept of value, above, we considered that the initial testing value function would contain two specific kinds of factors that influence the level of value in the process. One category is the expected return (or, benefit) which provides a positive influence on the function – it is able to increase value. In practice, there are countless factors that could add value, but as Hass [7] aptly notes, the best tests reduce the risk of defects remaining in the product when it is released to the customer. Hence, a very good way of evaluating positive factors in a proposed testing value function is to identify the capability such factors have for reducing the likelihood of defects escaping notice in testing. Fewer defects means higher value, essentially, and this is a risk-reduction calculation of value.

In terms of characterizing types of risk in software testing, there is currently no uniform classification scheme, although Hass’ [7] has offered a classification of four risk types: business risk, processes risk, project risk, and product risk. From the company-wide perspective, risks can be categorized as strategic, compliance-related, financial, operational and reputational [10]. Our view is that risks faced in software testing (when the risk event is the unintended release of a defect to the customer) are either indirect risks (comprised of business and operational issues no directly implicating the testing process) or testing-specific and direct, characterized as “technical risk.”

Business & operational risk relates to the importance of the software to the operation, integrity, or financial stability of the company. We think of these indirect risks as either Internal Risk (IR), which refers to risks arising from events taking place within the organization and External Risk (ER), which refers to the risk arising from the events taking place outside of the organization. To aid the reader in conceptualizing the risk factors in a value-laden framework, Table II provides a visual characterization of the risk factors against specific operational instances that may manifest in company operation. For internal risk, we consider that executive pressure within the company and either resource deficiencies or poor organization of resources are two primary items impacting the goodness of testing results. Externally, crucial impacts are failure in production, relevant regulations, and fierceness of competition.

| TABLE II. | KEY FACTORS OF UNIT VALUE IN INITIAL FUNCTION |
Technical Risk (TR) begins with the complexity of the software. A more complex piece of software is inherently riskier in production and testing than a less complex piece of software. Furthermore, a larger program or portion thereof is riskier than a smaller program simply because the larger program presents more opportunities for error. Technical risk also involves factors related to programmers, testers, and their tools. More experienced programmers present less risk to the finished product than less experienced programmers do, for example. New testing technologies in use are riskier than established technologies. To that end, we have derived primary components of technical risk in Table II: complexity, technology issues, requirements, personnel, dependency issues, issues with previous testing, and issues with the test environment.

The other major factor is related to costs that negatively impact value. Simply put, value decreases in the face of increasing costs, and, all things being equal, decision makers prefer to choose test cases that carry fewer costs in order to preserve value in the process. Therefore, in the initial unit value function, Unit Cost of a test case (UC) becomes a key factor. When estimating UC, one must take into account the costs of test case creation, the costs of running a case, the costs of determining success or failure for the case, and, possibly, related costs of using it to fix a subsequently uncovered defect in the code. A detailed explication of the cost structure for test cases is found in Gillenson et al. [11]. Our value function, derived from these costs, is characterized by the factors represented in Table III.

And, our proposed function for the determination of Unit Value of Test Cases is as follows:

\[
\text{Unit Cost} = \text{Prep Costs} + \text{CIV} + \text{CEO} + \sum^{n}_{1}(\text{CRT} + \text{CRR} + \text{CER} + \text{CME}) + \sum^{a}_{1}(\text{CMD} + \text{CFC} + \text{CRF})
\]

where \( n \) is the number of times the test case is run and, where \( a \) is the number of times a test case fails.

Besides the two main factors discussed in regard to our proposed value function, another special factor related to external risk is the Amount of Use (AU) of the Software Under Test (SUT). A frequently used piece of software, whether it is a full application or a feature of an application, is inherently riskier than an infrequently used piece of software. This may seem counter intuitive, until one considers that external risk increase for widely used applications that experience production failures, as the impact of the failure is more widely distributed and more costly to correct. Therefore, we combine external risk and amount of use of the SUT together in the initial function to demonstrate this potential negative impact on value.

In sum, we consider the business and operational risks, technical risks, unit costs, as well as amount of use as the primary factors combined to evaluate the value of a test case.

<table>
<thead>
<tr>
<th>No.</th>
<th>Category</th>
<th>Code</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Business &amp; Operational Risk</td>
<td>Internal Risk (IR)</td>
<td>Executive pressure within the company</td>
</tr>
<tr>
<td>2</td>
<td>Business &amp; Operational Risk</td>
<td>Internal Risk (IR)</td>
<td>Deficiency or poor organization of resources</td>
</tr>
<tr>
<td>3</td>
<td>External Risk (ER)</td>
<td>Crucial impacts of failure in production</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>External Risk (ER)</td>
<td>Relevant regulations</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Executive pressure within the company</td>
<td>Fierce competition</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Amount of Use (AU)</td>
<td>The relative amount of use in production</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Technical Risk (TR)</td>
<td>Complexity: complex code or function</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Technical Risk (TR)</td>
<td>Technology issues</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Technical Risk (TR)</td>
<td>Requirements issues</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Technical Risk (TR)</td>
<td>Personnel issues</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Technical Risk (TR)</td>
<td>Dependency issues: failure may trigger other failures</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Technical Risk (TR)</td>
<td>Previous testing issues: little unit testing was done</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>Technical Risk (TR)</td>
<td>Test environment issues: does not match production environment well</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>Unit Cost of a Test Case (UC)</td>
<td>Preparation Costs</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Unit Cost of a Test Case (UC)</td>
<td>Creation Costs</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>Unit Cost of a Test Case (UC)</td>
<td>Run Costs</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>Unit Cost of a Test Case (UC)</td>
<td>Failure Costs</td>
<td></td>
</tr>
</tbody>
</table>
3. INITIAL UNIT VALUE FUNCTION

In the previous section, we introduced IR, ER and TR as the three key elements which can (when properly managed) increase the unit value of a test case. Each of those risks, when judiciously resolved in the software testing process, act to decrease the probability of software failure and hence increases the value of the test case. On the other hand, any test case will consume resources such as funds and personnel, and this is also represented in the unit cost calculation. It is important to recognize that different test cases present different IR, ER, AU and TR elements, so in order to adjust the diverse influence of the four factors in the function, weighting is applied for each factor in the calculation. Therefore, in consideration of these notions, above, the initial function of unit value of a test case is represented as follows:

\[
\text{Unit Value} = \text{IR} \times w_1 + (\text{ER} \times w_2)(\text{AU} \times w_3) + \text{TR} \times w_4 - \text{UC} \times w_5
\]

where \( w_i \) for \( i = 1, \ldots, n \) is the weight of each factor.

In this function, there are several points should be noted. First, value is the relative worth of expending additional resources to add the test case under consideration to the testing effort. Also, SUT can be a feature of an application component, an entire application, or even a collection of integrated applications. IR, ER, AU, and TR are all based on the SUT while UC is based on the test case under consideration. Mathematically, all five of the factors are scaled from 1-n in whole numbers; factor weights can range from 0, upwards, and can include fractional components. The point of multiplying ER and AU arises from the consideration that a major failure in a heavily used application should likely have synergistically deleterious effects; further refinement can be made with the weighting coefficients \( w_2 \) and \( w_3 \). Lastly, for UC the entire unit cost function (not just the up-front preparation cost and creation cost factors) should be used, because an estimate of the eventual run costs and failure costs based on history is an important component in the calculation.

A. DATA COLLECTION AND ANALYSIS

For purposes of examining the initial unit value function, we plan to arrange 25 qualitative depth interviews with software testing professionals from different companies. At current stage, we have conducted 12 interviews from FedEx (a worldwide logistics firm) and its American branches. In next stage, we will conduct interview from other companies.

In order to promote objective responses, interview questions were general and not related to specific test cases or circumstances. The interview process was limited to one hour, and was guided by a documented and uniform procedure (see Appendix): introducing the initial function, explaining key function concepts, presenting the prepared questions and providing time for ad-hoc questions that might arise in the process of the interview interaction.

Table IV lists the tabular results of interview process. We found in the process of analyzing interview results that several concepts which had been excluded in the initial derivation of the value function were actually critical in practical contexts and were mentioned frequently or discussed extensively by respondents. These concepts include priority, revenue generation and code coverage. Several points that were included in the initial function were found to have different meanings or interpretations in view of respondent feedback. These included risk as a general concept, amount of use and unit value. Respondents were sometimes equivocal on the interpretations of these various elements, of course, and no universal criterion exists to identify potential consensus modifications to the function.
### TABLE IV. DISTRIBUTION OF KEY CONCEPTS FROM FEEDBACK KEY

<table>
<thead>
<tr>
<th>No.</th>
<th>Discussed Concept</th>
<th>Frequency</th>
<th>Frequency Percentage</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Priority</td>
<td>19</td>
<td>22.35%</td>
<td>New Concept</td>
</tr>
<tr>
<td>2</td>
<td>Revenue Generation</td>
<td>5</td>
<td>5.88%</td>
<td>New Concept</td>
</tr>
<tr>
<td>3</td>
<td>Cash Flow</td>
<td>1</td>
<td>1.18%</td>
<td>New Concept</td>
</tr>
<tr>
<td>4</td>
<td>Code Coverage</td>
<td>7</td>
<td>8.24%</td>
<td>New Concept</td>
</tr>
<tr>
<td>5</td>
<td>Testing Utility</td>
<td>2</td>
<td>2.35%</td>
<td>New Concept</td>
</tr>
<tr>
<td>6</td>
<td>Testing Frequency</td>
<td>1</td>
<td>1.18%</td>
<td>New Concept</td>
</tr>
<tr>
<td>7</td>
<td>Unit Value</td>
<td>13</td>
<td>15.29%</td>
<td>Existed Concept</td>
</tr>
<tr>
<td>8</td>
<td>Internal Risk</td>
<td>6</td>
<td>7.06%</td>
<td>Existed Concept</td>
</tr>
<tr>
<td>9</td>
<td>External Risk</td>
<td>5</td>
<td>5.88%</td>
<td>Existed Concept</td>
</tr>
<tr>
<td>10</td>
<td>Amount of Use</td>
<td>14</td>
<td>16.47%</td>
<td>Existed Concept</td>
</tr>
<tr>
<td>11</td>
<td>Technical Risk</td>
<td>9</td>
<td>10.59%</td>
<td>Existed Concept</td>
</tr>
<tr>
<td>12</td>
<td>Unit Cost</td>
<td>1</td>
<td>1.18%</td>
<td>Existed Concept</td>
</tr>
<tr>
<td>13</td>
<td>Weight</td>
<td>2</td>
<td>2.35%</td>
<td>Existed Concept</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>85</td>
<td>100%</td>
<td></td>
</tr>
</tbody>
</table>

4. DATA ANALYSIS

#### A. New Concepts

In consideration of the data from our interviews, priority is the word most frequently mentioned. The notion of priority is best expressed in the operational wisdom that test cases with high priority, other conditions being equal, always have a higher value than the test cases with low priority. There was a sharp dichotomy between respondents as to what priority implied in practical terms, however, and one view was that the priority given a particular case should be a separate factor in the function. This arises from the idea that priority is closely related to not only the value of a specific test case but also to the value of an entire application in which the case resides.

Continuing with the notion that test cases with high priority possess higher value than the test cases with low priority, suppose that there are two test cases developed for assessing the same financial calculation application -- one case testing an ordering function and the other testing a character-display function. It seems clear that the ordering function would have a higher priority than the character-display function because if the former malfunctioned, critical financial calculations might be flawed. Compared with the test case for character-display functions, in which accurate calculations might simply be displayed poorly, the test case for the ordering function would appear to have the higher value between the two.

A contrasting view is that priority is already embedded in internal and external risk factors, and would be hard to differentiate as a separate factor on its own. For example, the releasing deadline for an application pertains to internal risk, applications designed for handling customer complaints imply external risk, and dealing with cutting edge technologies being used by competitors might suggest both internal and external risk factors.

The point of contention among respondents was that priority could be taken into account at much more than its normal impact if it was listed in function as a separate factor, considering that it already plays roles in IR, ER and TR. To that end, we opted not to institute a new factor in the value function for the priority issue.

Revenue Generation (RG) is another hot concept discussed by nearly all interview respondents. The concern for revenue in the value function spanned both business management roles and technologist development roles, alike. Generally speaking, with regard to for-profit organizations, the most important goal of operation is revenue generation. In such contexts, where revenue generation underlying the profit motive is a key concern, testing as a necessary part of application development would also have to create some direct or indirect revenue stream to justify its own value as an organizational function. Some might suggest that revenue generation implies a degree of internal risk, given the high visibility of the revenue production process for executive oversight and competitive capabilities. In order to clearly demonstrate the core factors of unit value for a test case, then, we consider that revenue generation has the potential to significantly influence unit value, and should represent a new factor in the function.

Cash flow is a central concept of revenue, and thus might also be expected to play a significant role in value creation. In view of the critical nature of liquid funding in corporate operation, perhaps the value contribution role of cash flow is even greater than revenue generation. However, the distinction between revenue generation and cash flow is beyond the scope of this paper, and we will not examine it further. Hence, to facilitate a clear explication of the test case value function, we use revenue generation rather than cash flow as a contributory factor in the function, while suggesting that contexts in which cash flow bears a relatively greater import, that the substitute...
of cash flow for revenue generation be made in the function for those specific purposes.

**Code Coverage** is a special testing concept for testing which was not included in the initial function based on our interview results. Code coverage reveals how many application features are covered in a given testing case, with higher code coverage testing always covering relatively more application features. The interview participants tend to believe that test case value do not depends on code coverage or application features which are tested, but rather depends on whether the requirement of testing are satisfied. In view of this, it was considered inappropriate to include code coverage as a factor in the value function.

**Testing Utility (TU)** indicate the degree of satisfaction with the requirement of the testing process. In practice, considering that high code coverage testing can result in neither locating defects accurately nor providing clues to guide testers, justifying the value of a test case from the amount of code is not appropriate. Considering testing utility as a substitute for code coverage, an obvious shortcoming of testing utility is that utility scoring is far more subjective than the measurement of code coverage. Even so, testing utility still makes much more sense as a reliable indicator of value than does code coverage. Therefore, we set testing utility as a new factor in the function.

**Testing Frequency (TF)** is another critical standard which can be utilized to evaluate value of a test case. High usage frequency of a test case always presents greater value and priority compared to test cases with low usage frequency, and from the responses in the interview process, we see that testers typically prefer to use a test case in a regression test suite rather than writing a new test case due to effort and time savings. Moreover, as the amount of testing frequency grows, the value of the frequently used test case undoubtedly would be greater, so we multiple testing utility by testing frequency in the function in recognition of the close relationship between the two factors.

**B. Existing Concepts**

In the introduction of the initial value function, we introduced the concepts of unit value, internal risk, external risk, amount of use, technical risk, and unit cost as the key components of the initial function. As we gained additional perspectives from the outcomes of our interviews, meanings and feature of the concepts to be included in the testing context are gradually uncovered. In the following, we include and define specific features derived from the interview results.

**Unit Value (UV)** is a broad concept, fundamental to the entire notion of test case valuation. In order to explicitly demonstrate the unit value of test cases, it was necessary to clarify certain assumptions of the study for our respondents during the interview process. First, in our study we were only focusing on unit value of functional test cases. In other words, other types of test cases such as performance test cases or system test cases were specifically not under consideration. Secondly, the unit value of a test case is held to be a predictive value to be estimated before application execution rather than afterwards. Third, in a ceteris paribus consideration of capabilities, all the test cases under consideration could be completed if there were the necessary resources such as time, budget, personnel and so forth. In other words, the consideration of failed test cases or incomplete test cases in the function is beyond the scope of the study.

Given these assumptions, all the interviewees addressed their insight based on the initial function. Subsequent data suggests the unit value of a test case is a comprehensive concept that presents the predicted return projected by the tested application, potential direct or indirect loss decreased by the tested application, the effectiveness and efficiency of the test case, and the unit cost of the test case. Each factor in the function except for unit costs has positive impact on the value consideration.

**Internal Risk (IR)** is the risk arising from the events which take place within the organization but not related to technical operations, and which could result in potential losses that could be eliminated or eased by the testing. There are three critical points of emphasis requisite to justify internal risk: the risk derives from internal operations of the organization, the risk may directly or indirectly lead to near-term loss, and the risk could be resolved by testing. Based on common themes in the interview data, two pervasive administrative issues -- executive pressure within the company and deficient or poor organization of resources -- are considered as the main indicators of internal risk.

**External Risk (ER)**, risk factors exogenous to the organization, arises from events which are outside the firm but not related to technical operations, and losses arising from such risks could be eliminated or eased by the testing. As to the indicators of this risk factor, they are the same as those driving internal risk (executive pressure and deficient or poor organization of resources), except the risk source derives from outside the firm. From the view of the participants, “Impact of failure in production” and “Relevant regulations from law and convention,” serve to indicate external risk as a factor.

As some interviewees mentioned, ER and IR can be dynamically interactive with each other. For example, as marketplace competition subsides, internal executive pressure which is part of IR might be eased accordingly. Nevertheless, the value function offered here is only for static calculations, to evaluate unit value at a certain time point where user is estimating the test case, thus the case of dynamic interaction is not considered in this study.

**Amount of Use (AU)** is the relative index used to indicate the degree of use seen by an application. Since AU is relative index and will vary with different industries, this is a factor that will need to be independently estimated by context. For example, AU should vary widely between the context of a globally utilized social networking application and a local weather report application. AU is also dependent upon the software customer, which is a significant factor directly influencing ER rather than IR, and for this reason ER is multiplied by AU to present this relationship in the function.

**Technical Risk (TR)** derives from technical issues which could not be included in the business and operational risk factors. In this case, IR, ER, and the potential loss TR brings could be eliminated or eased by testing. Just as was the case
with TR in the initial function, all seven items -- **Complexity Issues** (CI), **Technology Issues** (TI), **Requirements Issues** (RI), **Personnel Issues** (PI), **Dependency Issues** (DI), **Previous Testing Issues** (PTI) and **Test Environment Issues** (TEI) -- are retained in the revised function based upon comment received from interview participants. Moreover, **Vendor Issue** (VI) was added to the function as a new item which represents issues caused by software testing vendors in regard to the test case, (items added to the function are include in Table IV).

The eight items of TR with separate weight are listed in the following revision of the function:

\[
TR = CI \cdot w_i + TI \cdot w_j + RI \cdot w_3 + PI \cdot w_4 + DI \cdot w_5 + PTI \cdot w_6 + VI \cdot w_7 + TEI \cdot w_8
\]

where \( w_i \) for \( i = 1, \ldots, n \) is the weight of each factor.

**Unit Costs** (UC) is a complicated concept in testing, with specific contextual variations. However, we adopt the calculus for unit cost determination directly from our prior study of costs of test cases [11], for specific use as a value-impacting factor, here.

5. **REVISED UNIT VALUE FUNCTION**

A. **Unit Value Function**

Based on previous discussion of new concepts revealed in field interviews and existing concepts from literature and prior studies, the final function of unit value is derived and listed below, with the finalized key factors of the function listed in Table V. Specifically, the unit value of a test case is a relative value that subtract the score of the only negative factor in the function, or say, UC, which lead UV decrease as its score goes up, from the score summation of all the positive factors, which can stimulate the UV increase as the factor score increases.

\[
\text{Unit Value} = RG \cdot w_1 + IR \cdot w_2 + (ER \cdot w_3)(AU \cdot w_4) + TR \cdot w_5 + (TU \cdot w_6)(TF \cdot w_7) - UC \cdot w_8
\]

where \( w_i \) for \( i = 1, \ldots, n \) is the weight of each factor.

### TABLE V. KEY FACTORS OF UNIT VALUE IN REVISED FUNCTION

<table>
<thead>
<tr>
<th>No.</th>
<th>Category</th>
<th>Primary Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Revenue Generation (RG)</td>
<td>The revenue generated by the application where the test case applied successfully</td>
</tr>
<tr>
<td>2</td>
<td>Business &amp; Operational Risk</td>
<td>Executive pressure within the company</td>
</tr>
<tr>
<td>3</td>
<td>Internal Risk (IR)</td>
<td>Deficiency or poor organization of resources</td>
</tr>
<tr>
<td>4</td>
<td>External Risk (ER)</td>
<td>Impacts of failure in production*</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>Relevant regulations from law and convention*</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>Fierce competition</td>
</tr>
<tr>
<td>7</td>
<td>Amount of Use (AU)</td>
<td>The relative amount of use in production</td>
</tr>
<tr>
<td>8</td>
<td>Technical Risk (TR)</td>
<td>Complexity Issues (CI): complex code or function</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>Technology Issues (TI)</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>Requirements Issues (RI)</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>Personnel Issues (PI)</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>Dependency Issues (DI): failure may trigger other failures</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>Previous Testing Issues (PTI): little unit testing was done</td>
</tr>
<tr>
<td>14</td>
<td>Vendor Issues (VI)</td>
<td>Test Environment Issues (TEI): does not match production environment well</td>
</tr>
<tr>
<td>15</td>
<td>Testing Utility (TU)</td>
<td>The effect of the test case</td>
</tr>
<tr>
<td>16</td>
<td>Testing Frequency (TF)</td>
<td>The general use frequency of the test case</td>
</tr>
<tr>
<td>17</td>
<td>Unit Cost of a Test Case (UC)</td>
<td>Preparation Costs</td>
</tr>
<tr>
<td>18</td>
<td></td>
<td>Creation Costs</td>
</tr>
<tr>
<td>19</td>
<td></td>
<td>Run Costs</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>Failure Costs</td>
</tr>
</tbody>
</table>

Note: † indicates new concept which do not exist in initial function; * indicates revised concept which exists in initial function.

### TABLE VI. THE SCALE OF SCORE AND WEIGHT OF FUNCTION FACTORS

<table>
<thead>
<tr>
<th>No.</th>
<th>Category</th>
<th>Score Range</th>
<th>Weight Range (%)</th>
</tr>
</thead>
</table>

B. **Scoring and Weighting**

Given the unit value function, standardizing each factor on a unified norm could differentially impact the final result significantly. The range of score and weight for each factor in our function have been listed in the Table VI and Table VII, and in the following paragraphs we introduce the score and weight features in the function.
TABLE VII. THE SCALE OF SCORE AND WEIGHT OF TECHNICAL RISK ITEMS

<table>
<thead>
<tr>
<th>No.</th>
<th>Category</th>
<th>Score Range</th>
<th>Weight Range (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Complexity Issues (CI)</td>
<td>0 - 15</td>
<td>0 - 100</td>
</tr>
<tr>
<td>2</td>
<td>Technology Issues (TI)</td>
<td>0 - 15</td>
<td>0 - 100</td>
</tr>
<tr>
<td>3</td>
<td>Requirements Issues (RI)</td>
<td>0 - 15</td>
<td>0 - 100</td>
</tr>
<tr>
<td>4</td>
<td>Personnel Issues (PI)</td>
<td>0 - 15</td>
<td>0 - 100</td>
</tr>
<tr>
<td>5</td>
<td>Dependency Issues (DI)</td>
<td>0 - 15</td>
<td>0 - 100</td>
</tr>
<tr>
<td>6</td>
<td>Previous Testing Issues (PTI)</td>
<td>0 - 15</td>
<td>0 - 100</td>
</tr>
<tr>
<td>7</td>
<td>Vendor Issues (VI)</td>
<td>0 - 15</td>
<td>0 - 100</td>
</tr>
<tr>
<td>8</td>
<td>Test Environment Issues (TEI)</td>
<td>0 - 15</td>
<td>0 - 100</td>
</tr>
</tbody>
</table>

Score for each factor except UC is in the form of a relative index with magnitude ranging from 0 to 120 (see Table VI). Zero indicates the lowest possible value and 120, the highest. UC uses real scores calculated by the UC function [11] to indicate the magnitude of the factor. A high score means high costs for the unit case and a low score means low costs. The score of each TR item ranges from 0 to 15 which differs from the factors of the function, and the summation score of each TR items is the TR score (see Table VII).

Weight is a percentage coefficient that indicates the specific contribution and important of each factor in the larger function. The score of the weight percentage ranges from 0 to 100. The impact of a factor in the function gradually increases as the score of the weight goes up from 0, which indicates the factor has no impact in the function of the test case, to 100, which means the factor has full impact in the function of the test case. This rule is also applied to the weight of TR items.

There are two general criterions requiring attention when scoring or weighting. The first is Integrity. When scoring or weighting, users should prudently consider the impact difference between the scoring/weighting factor and the other non-scoring/weighting factors in the function. The difference should be accurately presented by the score or weight percentage assigned. A second criterion is Consistency. The score or weight of the same test case may vary between different users in a company due to the diversity of subjective judgment. In order to ease the potential impact of such variation, we suggest users set their own specific scoring and weighting standards.

C. Calculational Process

Given the unit value function as well as scoring and weighting standards, the calculation process requires a degree of orientation for proper use of the function, and for providing users a map for applying the function. There are three steps in applying the function in calculational terms, as demonstrated in Fig. 1.

The first step is the application level procedure which processes the factors affected by the application under test. Specifically, RG, IR, ER, AU, as well as their according weights from $w_1$ to $w_4$ are considered. One point requires emphasis: the score and weight of the factors are based on the impact of the application under test, and this differs from the second step.
The second step is the unit level procedure that processes the factors affected by the unit test. Specifically, TR, TU, TF, UC, as well as their according weights from w₁ to w₈ are considered. In contrast to the first step, this step focuses specifically on the test case itself, rather than the tested application. Moreover, it is recommended that UC be based on real cost information derived from financial data and accounting statements from company operations, which will result in a UC score that is more reliable and consistent [12].

In the last step, the final unit value of the test case can be achieved by subtracting the result of the second step from the result of the first step. Considering the wide range of test cases and testing cost and value factors across different industries, the final unit value may result in several potential outcomes: positive value, negative value, or zero value and negative value. For the latter two forms, it is unrealistic to claim that a test case either has no value or results in a loss of value. That said, it is important to remember that regardless of the valance of the outcome, the unit value of the test case is a relative calculation. When comparing the result of one test case with that of others does the calculated unit value has meaning; it has little meaning in isolation from other company information. Specifically, unit value calculations inform users as to which test cases are better than others, based on relative value considerations.

D. COMPARISON WITH ROI

Before research investigated the nature of the unit value of test cases, there were few guides for assessing the relative worth of functional test cases to guide decision making. Return on Investment (ROI) is one performance measure widely used for evaluating business projects which shares some features with and bears some similarity to the UV function. In some cases, ROI could even be a candidate tool for estimating the value of functional test cases. For that reason, we specifically contrast the UV Functional approach with the ROI approach to evaluating test cases to give users a clear picture for practical reference.

In Phillips’ [13] study, ROI (as listed below in functional form) was defined as a percentage figure, arising from net program benefits divided by program costs, with net program benefits represented by total program benefits minus costs. As noted by Cresswell [14], ROI is prevalent in various industries, projects and programs and despite minor variations in calculational formulae, the basic principle of ROI remains the same:

\[
\text{ROI} = \frac{\text{Net Program Benefits}}{\text{Program Costs}} \times 100
\]

From the functional calculations underlying both UV and ROI, we find several distinct differences. The UV function is constituted not only by revenue/benefit factors but also considers the impact of risks, utility and use frequency factors. In contrast, ROI does not include these factors, and, accordingly, is not able to wholly present the value of test cases. To that end, UV is much complicated (one could say, comprehensive) than ROI, with UV constituted by 8 factors, and ROI by only two. Despite the more complex functional form, the calculation of unit values is easily programmed for processing by computer. Interestingly, UV is not constrained by factor metrics. UV organizes various factors with different characteristics at unitary level. In other words, all the factors are represented by their relative score and weight. In contrast, only the factors measured by monetary units are included in ROI. Lastly, UV is able to present the scale impact on value, whereas ROI is a basic ratio that can compare any project or program, regardless of size, potentially leading to misdirection when the easily foreseeable outcome of a very small test case with extremely high ROI results in preferential choice, in contrast to a very large test case with lower ROI. Since UV is not a ratio, the impact of test case scale is considered in the UV function.

6. GUIDE FOR APPLYING FUNCTION

Now that the unit value function has been developed and demonstrated, we offer helpful tips for prospective users industry working environments who may wish to apply its principles. In this, we demonstrate the goal of the function, identify relevant users, and detail procedural attention matters for using the function.

A. Who Can Make Valuable use of the Function?

The purpose of unit value function is to aid decisions on choosing appropriate test cases by providing a quantitative method to discriminate between the difference of relative value rather than absolute value between a given set of potential test cases. To this end, primary users might be managerial staff from testing groups, ranging to middle-to-high level positions such as director or project manager, and whom bear responsibility for scheduling test cases. Other users could be departmental accountants responsible for project budgets or marketing executives responsible for the cost-effective launch of software brands.

B. How to Collect Function Data?

Although the function is reasonably easy to understand, finding the data with which to fit it is more challenging. Such data might derive from different departments and workgroups, and in some cases it might not even be recorded by the firm. In order to reveal the value of a test case, objectively and efficiently, we suggest application-level data should be collected from marketing or business operational departments and that unit level data could be collected directly from testing or development groups.
For existing data, the form it might take could include budgets, accounting reports and operational statistics. For nonexistent data, estimates could be made based on corporate planning and strategic documents and from industry news reports. For example, the RG data of one application may be acquired easily from and accounting department, but AU data of an application might not be recorded, requiring an estimate derived from a related statistic index in the company or even from news report or other data created by a third-party.

C. How to Estimate Scores and Weights for Factors?

As we have indicated, nonexistent data may need to be estimated or interpolated, and this method also applies to developing the weights for factors when objective information is not available. Naturally, when data underlying key factors is estimated, this has the potential to skew unit valuations, so it is important to do so as objectively as possible. To that end, we strongly recommend that users who are developing unit value functions in the same company should establish, a priori, specific standards and procedures for estimating scores and weights.

D. How to Interpret Results?

Each application of the unit value function calculates a result for one specific test case, and the value is the relevant scale that is used to compare the assessed case with others to determine their relative worth in use. Therefore, when two or more test case values have been acquired from the function, the accurate interpretation of these results lies in comparisons of magnitudes of relative value between cases. To that end, we recommend strongly against interpreting the calculated unit value as the real and objective value of a given test case, and the unit value functions has little meaning the consideration of only one test case value function. It is meant to be used across cases as a decision tool for optimizing the selection process. There is little meaning in the process without the relative comparison process it implies.

7. LIMITATIONS

Although we think the unit value function we have developed and offered here is a comprehensive method for estimating the value of a test case, some limitations to the process accrue. Our approach is specifically limited to the consideration of functional test cases, only. Further, in terms of assumptions underlying the function, we definitely assume that all test cases are finished on time. The direct and indirect impact of testing delays is not considered in the function and its development. Subjective estimation techniques were advocated for situations in which there was no data or in which weighting information was difficult to obtain. Clearly, the dangers of subjectivity pertain, and results can only be as good as the dedication to objectivity in estimation that is employed when real data is not easily available.

Though it is an appealing notion, the comparison of test case calculation across different industries is not supported. It will be difficult enough to maintain objectivity and consistency within industries, notwithstanding the notion of comparisons to other industries.

8. FUTURE RESEARCH

This paper explored a new method to estimate the unit value of a functional test case, but as described in the limitations section the function still has some aspects that could be improved or modified for more extensive or more accurate use.

A large amount of testing data spanning a number of companies and industries could be collected to test the reliability and validity of the function in differing contexts. This is a critical process for detecting function defects which may be hidden by situational context in which the function was developed. Establishing a standard for estimating factor scores and weights another future advancement the evolving unit value function. Uniform standards have critical influence on the results of the unit value calculation. Lastly, though we caution against it in the case of the function offered here, future directions should consider how to compare the unit value of test cases across different companies and industries.

REFERENCES

APPENDIX

Interview Questions

This interview is related to evaluating unit value of a test case. Each participant is assumed that they have a bunch of unit tests can be applied, but to implement all the tests is not practical. In this situation, participant should make some choices from those tests based on their own justification. Please answer the following questions and give your reason.

1. What factors (Unit Value, Unit Cost, or other factors) you mainly depends on to make choice? Why?
2. Do you agree or disagree with the initial function of unit value of a test case if you consider Unit Value is the key factor to impact your choice? Why?

3. If you agree with the initial function of unit value of a test case, should Business & Operational Risk, Technical Risk and Unit Cost be expanded in the function, as opposed to having a separate table for its components? Why?
4. Do you agree or disagree with the components of Business & Operational Risk and Technical Risk? Give your opinion?
5. Do you think Weight, standard scale setting for each factors and Amount of Use can establish a relatively clear and practical approach to assess the unit value of a test case? What’s your opinion?
6. Do you have any other comments on this function?
At Asynchrony we’ve had to learn to crawl, walk, and run over and over as technologies and our customers have changed throughout the course of the development of the Mobile Field Kit (MFK), one of our government projects. Our project is complex with many moving parts; it’s got multiple technical stacks, changing customers, lots of unusual equipment, lots of developers, and, with this being a software development company, we have had a good number of developers who have cycled in and out of the project team. We feel we have proven ourselves at being really good at changing and adapting. The success of this project has been due in large part to its ability to remain flexible, continuing to meet our customer needs as they evolve, and the creation and achievement of high quality processes. With all these moving pieces, how do we assure we are creating a high quality product and advocating for quality throughout our processes? We are able to do so because of our ability to combine the best traditional testing practices with unconventional quality processes to help our users succeed in their missions.

1. BACKGROUND
The Mobile Field Kit (MFK) puts state-of-the-art tools and technology directly in the hands of field team members, including first responders, civil support teams, and other physical security teams. The MFK allows them to acquire, store, assess, and share information, both within the team and across organizational boundaries. The system was initially created by the U.S. Navy for the Joint Explosive Ordnance Disposal community. Through Department of Defense-sponsored activities, the Army’s 20th Support Command and National Guard Civil Support Teams currently utilize the MFK as a tactical tool for Weapons of Mass Destruction (WMD) defeat missions specifically focused on radiological, nuclear, and chemical threats. The software has been used for everything from search and entry such as determining if a suspicious smell coming from someone’s home is potentially threatening, to reconnaissance, with missions ranging from detecting radiation/chemicals in the public water system after a chemical dumping, to providing WMD defeat support for such events as the presidential inauguration, the Super Bowl or the Boston Marathon.

The system is made up of multiple technologies. The application runs on a Windows machine and on Android. It has interoperability with multiple sensors, radios, and other Off-the-shelf applications. The product utilizes a mixture of communication technologies ranging from mobile networks to closed secure portable tactical radios. The MFK software is typically configured in a package that includes the software, hardened tablets and/or laptop computers, Android phones, secure wireless mesh-networking and hardware facilitating WAN connectivity via cellular, satellite or other networks. The MFK configuration can be housed in a portable, hardened case that includes everything needed for deployment across the city or halfway around the globe. It’s built to easily integrate a wide array of communication and sensor suites within a field-tested, standards-based platform.

2. MY STORY
2.1 History
Below is a timeline that shows how the MFK quality process as evolved over the years. As you can see, we have definitely needed the agile principles in place to handle the transformation of users and the application.

At the same time there have been many changes that have impacted testing. We will discuss some of these in detail below.
2.2 Automation

In 2007 we started using some of the principles found in Extreme Programming. However, without someone there to assist them as we started this journey, the team was lackluster and would only occasionally pair and only sometimes write unit tests. A Quality Assurance (QA) resource was added to the team to develop a suite of manual user acceptance tests (UATs) for regression testing and to write tests as stories were completed. This introduced a new problem when we found it difficult to find sufficient time to run all those manual tests. We decided that rather than running the tests frequently, we would wait until there was a potential release candidate and would then set aside time to run all the tests. This would cause some bugs to be found late, which in turn could cause a delay in the release. We then ended up with large pieces of code that had not been tested, creating fragile code that would break as future changes to the code were made.

In the beginning of 2008, a new technical lead was added to the team that insisted on full unit testing. We began creating automated acceptance tests for stories as they went through the board. The team initially did an investigation into testing tools and selected a tool called FIT. In FIT test cases are displayed in HTML tables, which can enable customers to assist with writing the tests. The team went with FIT and ran with it for 1-2 years, but it was painful. In order to test with FIT we had to put in our production code the ability for FIT to inspect the objects so we could see the state of the object. We actually created a design pattern called the “violator pattern” to allow this. In retrospect, we wish we would have done an experiment where we used FIT for a month and WindowTester for a month to see which worked better.

In September of 2009 after our evaluation we purchased the required licenses to use a tool called WindowTester. Once the developers started using WindowTester we found it easier and wished we had used it from the beginning. The team continued automating as many of the manual tests as we could. Now we don’t have to spend weeks and weeks doing manual regression testing and we have confidence that new code is not breaking any existing functionality. We monitor the pass/failing of builds on a big monitor located in within view of the entire team. When a test fails, the team sees it immediately so that it becomes a high priority and we can address those fixes during our daily Kanban flow. As our customers have continued to request new functionality, the code base has continued to grow, and the number of tests has continued to increase.

After some time, we noticed the automated UATs started taking over an hour to execute, so the team decided to implement a matrix build process that would allow them to spread out the load. When running the tests linearly,
they could take up to more than two hours to run. Currently, we have 18 build boxes and we are averaging 20
minutes for a build. From a testing pyramid perspective, we currently have approximately 12,000 unit tests that
take approximately 6 ½ minutes to run. We have approximately 1000 automated user acceptance tests that take
about 20 minutes to run, and 200 manual acceptance tests that take 2 people about 3 days to execute.

An additional concern about our automated testing until recently has been our inability to test our integration with
sensors in an automated fashion. One of the most noteworthy benefits of the MFK is its ability to quickly integrate
with a sensor. Sensor integration along with the MFK communication strategy provides the most value to our
users of the MFK. The ability to automate tests using a real sensor is difficult in itself. In many situations we
found that we may not even have access to an expensive radiological sensor. Initially, the team would try as hard
as they could to make sure we could get the sensor that we were going to integrate into the office, but as time went
on, we found times when we could only connect to a sensor over the Internet, or that we could only get it in the
office for a couple days. Unfortunately, this didn’t enable us to get all of the data we would like, and we were only
able to test a limited amount of the sensor’s capabilities. Our sensor integration is fortunately dependent on an IP
based system where most of the data is similar. Team members seeing all the pain it took to get these sensors in
house and how we were unable to test all the necessary functionality decided to figure out a way to ‘simulate’ a
sensor. We received agreement from the team to allocate time and a pair to develop an XML simulator. We
basically took a simplistic approach to parsing of the data. The simulator generically reads data from a file and
then sends the data back over the network. The big win is how generic it is and with most of our data being pretty
similar, we can use this simulator to test about 6 of the 10 sensors we are currently integrated with.

An additional challenge we have been able to overcome using WindowTester is the ability to start and test multiple
instances of the MFK that can communicate and coordinate with each other. Having this allows the team to test
their ability to send data from a sensor to the MFK and its nodes to verify the data is transmitted and displayed
correctly. With the integration of the Android platform, we can also verify that the data coming in from the
Android platform is parsed correctly and displayed correctly within the MFK.

2.3  Field Testing

We try to do functional testing that will give us the most bang for our buck. One way we ensure quality and
usability of our application is by going out with the equipment and testing using scenarios that our users might
actually participate in. The team originally started going out to Forest Park in St. Louis a couple of weeks before a
release to test a build candidate in a more realistic environment with a mesh network. Testing in the park allows
us to get away from tall buildings which can cause lots of interference and test the application on a more reliable
wireless network where there is little interference. We also know that in lab testing vs. real world testing we
always see differences, things will work fine in the office but not so well when we go outside. Because our system
is dependent on the network, when you get on the edge of the network, things behave differently, and odd cases can
happen. Another advantage is seeing the application in the sun; since, our users are outside using the software.
We need to test the user interface with a glare on it. This has led us to implementing higher contrasting colors or
making changes in buttons/windows to be more visible. These types of tests have always fielded something, such
as issues with synchronization when things are going in and out of network to data out of order.

QA saw the benefits of testing in Forest Park and wanted to do this type of testing on a more regular basis to
enable us to find some of our bugs earlier. We agreed to try and start doing field tests around our office building,
but we ended up often canceling because every time we went to do a field test, someone would say let us finish this
one story and then we’ll do the field test. This would go on for six months or more where we were always putting
off field testing until tomorrow for whatever reason. Finally, we got agreement from the team to perform field tests
every Tuesday morning no matter what. We might modify the scenarios based upon what’s in the build but we
always test something. The tests are meant to verify functionality that was completed in the past week is fit for
customer use. We also see benefit because it gets the developers away from the code and actually looking at the
application and thinking like their users.

In the beginning, the tests were mostly focused on equipment, most importantly the radios. We struggled many
times to not make the test about the equipment but rather about the application because frequently it was difficult
to get all of the equipment to work. Something that has helped us is using cellular for our network for most of our
tests. This enabled us to concentrate more on the software and less on the application. On a negative side though
we again fell into this trap of not seeing the edge cases which would possibly lead to a support call regarding
someone having an issue with the MFK on a radio network. We recognized we need to test sometimes with cell,
sometimes with the Wi-Fi radios, and other times with both. Crawl, walk, run, when we started we didn’t even
know how to crawl, we didn’t know how to perform basic tasks, and we kept pushing it off, and pushing it off. Then we said, “You just got to do it,” and we have practiced it, and then as we’ve learned, we’ve optimized it.

2.4 Getting To Know Our users

After the initial success of the project and as our user base was changing, we ran into challenges with obtaining access to our users. Civil Support Teams are busy and their schedules are more packed than ever before. Our users are very educated and specialized men and women who are needed on the ground protecting the country. Unfortunately, this has made it hard to get them to have time to assist with the writing of stories or test cases and spending time at our Downtown St. Louis office. Ultimately, our success comes down to our ability to deliver quality to our customers and part of that assurance process is actually sitting down with our users. In the past we would do this with occasional trips and trainings where we could get feedback and that in turn would come back and get filtered down to the team.

For example, we were testing our application based upon a standard pattern of users going down range a short distance with a wireless mesh network that, due to the terrain, had little interference. As it turned out, the Civil Support Team users in the meantime were struggling to use the system in a more urban area, where tall buildings and concrete infrastructure caused issues with the mesh network. They were also beginning to use the system across long distances. We didn’t realize this until the program manager, a developer and I had gone to a CST site to provide training on our latest release. Our program manager decided that not only would we do training but this would be a good time to have the CSTs show us how they were actually using the application. The CSTs immediately began to show us a disturbing issue where they would have communication problems when a team goes out on a mission. They were feeling frustration because they couldn’t communicate with the person next to them since they had lost connectivity with the hub of our hub-and-spoke network architecture. The users creatively came up with a new-work around for how our network is setup. Their idea is to have multiple hubs going down range, so that all of the people in that group near the hub would always be connected. So instead of the usual one hub/command node topology, they were having multiple hubs, one with each team so they could stay in contact with each other. This solidified to us the importance of tight feedback cycles with end users/customers. We knew we needed a mechanism to obtain an understanding of our users to assure the Development and QA teams were creating and testing the ‘right’ things. This visit also made us realize the importance of getting QA and developers in contact with our users and to provide support not only with our application, but with the entire mission. We started becoming an integral part of the CSTs workflow by providing capabilities and the hardware to run cell networks that allowed them to cover their growing need to support new types of missions.

In 2014 we had a great idea to bring all of our customers together for a “community of interest” meeting, which would also have a secondary benefit of helping us in assuring quality. Appreciating what we had done our users agreed to us hosting the meeting. All pilot CSTs came to the first User Group meeting in March of 2014. This meeting was the first time to get users together and to have good interactions between them and our development team. We all learned from each other. There were lots of hands-on activities with the MFK using different aspects of the system. We had them walk us through their pain points. We took these and identified solutions to improve their missions, and QA became smarter in their testing. This year’s meeting was a little different because the goal was to get feedback from all the CSTs on how they use the system. The teams again came together and met on May 2015, for a week these teams dedicated themselves to sharing how they had planned on using the application in upcoming events. The QA team could then take what was learned and prepare realistic testing scenarios and have an appreciation of the conditions in which the users actually used the application. Working with users remotely who cannot be embedded with the team or physically present on a regular basis, highlights the critical importance of our user group meetings.

2.5 Automated Endurance Test

We’re only ever doing a field test for couple hours, but we know some of users have to use the application for multiple days. We also have a long standing issue where if you have multiple tacticals and multiple sensors setup and running for a long time the system eventually just bogs down and becomes unusable. Trying to capture the problem in a way that is repeatable has been challenging. We needed to develop a load testing mechanism that is as automated as possible so that it can run regularly and efficiently. This is exactly what we’re working on now. We had developed a set of manual stress tests that involved setting up a large number of computers connected to various sensors, and this alone could take as much as a half a day to assemble. We didn’t want something that would take hours to setup, and we didn’t want to bring back a process that only gets tested right before a release, potentially causing us to catch bugs late in the game. We needed to be able to run this test as a part of our continuous integration so that we test it early and frequently. I have two tests in particular that I would like to share.
The first basically took the test mentioned above and put it together using WindowTester running multiple nodes and sensors. We have set this test up so that it runs continuously until the application breaks, and then error logs are created to capture any issues. These logs are then verified manually by the development team. When we initially started running this test, it would only make it a couple days before the system would fail. After we were able to pin-point what some of the issues were and fixed them, we were able to run this test for over a week. This is a very unusual test because it runs for a very long time and we intentionally let it run until our application fails. Some questions that we were trying to answer are for example; At what point does the application break down with x many sensors? How long can we run the application before we start to see the application slowing down?

The second test involves verifying that we do not see an increase in the amount of time it takes for the data in our application to sync. For this test, we again put together two MFK nodes and have them lose connectivity for x number of hours. We then monitor the time it takes for them to get their data back in sync. We plot the duration it takes to sync over time to determine if we see an increase in the amount of time it takes for the data to sync. This test result also gets displayed on our monitor in the MFK area to ensure when we see an increase in sync time that we can turn around and fix it quickly.

3. WHAT WE LEARNED

We started with development approaches that have led to high quality production such as test driven development, pair programming, continuous integration and continuous improvement. Our quality assurance processes are designed to prevent bugs, not just fix them. Before we write a single line of code, we create tests to validate whether each unit of code will work as intended and we continuously integrate and test to ensure that the system works exactly as it’s designed to work after each and every change. By putting together a combination of traditional testing practices, automated testing practices, as well as continuous learning and improvement. We have become the application our users depend on when doing security for such events like the Boston marathon. We feel we have had great success on our project, and the future looks great for our project.

4. ACKNOWLEDGEMENTS

Special thanks to the following who allowed me to ask them a ton of questions and gave me the right answers to assist with the writing of this paper: Mark Dinman, Kevin Pfarr, Jake McCormick Dan King, and Mark Balbes.

I want to thank my fellow team members Kyle Ladd who helped me get this paper started, Danny Elfanbaum who managed to help me with my grammar and putting the story together, and finally to Tom Hermann who was also able to help me fill in some of the blanks.

Special thanks to Jason Tice who is my advocate and pushed me to do a presentation as a ‘growing’ experience.

Thanks to our Government sponsor, Luke Erikson.

Thanks to my shepherd, Robert Nord. Your advice was sound and your patience was masterful! You even came looking for me when I was missing ;)
Software Testing and Cyber Security

Robin Poston, University of Memphis
Nicole Samson, University of Memphis

Abstract: In December 2013, Target Stores’ systems were breached with over 40 million customer credit cards and other personal information reaching the hands of cyber-criminals. If Target had performed more exhaustive testing of their enterprise systems, it is possible they could have caught and fixed some or all of the vulnerabilities that led to the intrusion. What we learn from Target’s loss is that all companies should be testing for security vulnerabilities. As security breaches become more frequent, it becomes increasingly important to protect data systems and ensure they stay continuously protected. Yet, how do we know when our systems are protected? More importantly, how do we effectively test system security? By thoroughly testing the security of the system, we can potentially prevent security breaches from infiltrating our systems. Initially, we conducted a review of the scholarly and best practices literature to address these questions. Through the review, we created an initial draft of a comprehensive matrix, which we call the Security Testing Coverage Matrix, illustrating which test procedures may help with early identification of security vulnerabilities. The goal is to identify potential security weaknesses that may be fixed prior to being breached. The Matrix will then be presented to a panel of experts who will offer feedback for further refinement. The Matrix is useful to security and testing experts in the field who may utilize the Matrix in making testing coverage decisions about new and existing system tests to prevent data breaches. This Matrix is offered to help organizations formulate a comprehensive cyber security test plan.

Keywords: software testing; cyber security

1. Introduction

Cyber security has become a hot topic in the past several years, and it will become even more important in the future. Data breaches are becoming more commonplace, and hackers are becoming more brazen in their attempts to gain entry to protected systems and steal any data they can find. The Target breach (Radichel, 2014) shows that it is absolutely essential to fully protect our systems and that achieving minimum industry standards is not enough.

Target Stores’ systems were breached in December 2013 which resulted in the theft of over 40 million customer credit cards and other personal information. The hackers were able to gain entry through a vendor portal and ultimately installed spyware on the POS systems used by Target. This spyware stored information from the POS system’s memory into a DLL file that was then moved to its final destination: an FTP site where the hackers could download the data.

The Target breach took place over the course of about two weeks. During that time, an alert was triggered for security staff in India. The staff in India then forwarded the information to Target employees, but those employees failed to act on the alert.

Had Target sufficiently tested their systems, it is possible this breach could have been prevented. This paper seeks to help testers determine what to test and, more importantly, how to test systems from a software testing standpoint. A literature review was conducted to gather a list of security areas and software testing types and techniques. A Security Testing Coverage Matrix was then developed to pinpoint which tests are useful for testing each security area.

2. Literature Review

A literature review was conducted answer one basic question: How do we test for cyber security? The answer was surprising. We should be testing our systems for security, but very little is written about exactly how to...
test systems. This potentially leaves teams testing for only the most basic security requirements as outlined in company or industry security standards documentation. In some cases, such as the Target breach, this may not be enough.

Industry and security standards generally offer a baseline set of security requirements that a company should implement. It is up to each company to either stop there or develop a more stringent set of security requirements and then test to those requirements. At this point, there is now an additional question: What types of cyber security threats exist?

Further review of the existing literature helps to answer this second question. *Roadmap to Information Security* (Mattord & Whitman, 2011) details common cyber security threats and presents the NIST security controls (NIST SP 800-53 Rev 1). These lists were then pared down with a software testing focus (see Figure 1).

Research was conducted to also find a list of all major types of software testing (Pradhan, 2012), from the everyday to the more rigorous. The original list contained over 100 different types of software testing. This list was pared down to remove duplicates. The list was further massaged to categorize related testing types into more broad categories or classifications of testing types.

3. **Security Testing Coverage Matrix**

Based on the findings in the literature review, a matrix has been developed that examines several major testing types and security areas. These testing types are examined for each security area to determine which software testing techniques can help teams test the security areas. This matrix has been named the Security Testing Coverage Matrix (see Figure 2) and will be useful not just for security experts but for testing and software development experts as well.

The top of the Matrix contains the major cyber security areas that should be implemented and tested, sectioned off by category or control area. The left side of the Matrix contains the major software testing types used specifically for testing software, sectioned off by category.

The cross-section of the testing types and security areas indicates which tests will be useful when testing each cyber security area. These tests can then be incorporated into a comprehensive testing plan used by any testing team, whether or not that team is a security team.

4. **Conclusion and Next Steps**

The Security Testing Coverage Matrix will be helpful for testing teams, security experts, and development teams who need to ensure that systems are secure. Development teams can develop software according to the security areas, testing teams can develop and execute test cases based on which testing types help with each security area, and security experts can ensure that systems are secure by also referring to the Matrix.

As the next step, the Security Testing Coverage Matrix will be presented to several experts in the fields of testing and security who will offer feedback for refinement of the Matrix. This may include determining whether or not a test will help with a security area or even moving a software testing type from one category to another. The goal is to have a comprehensive matrix that will help experts secure systems and avoid future breaches where possible.
Figure 1 Realm of Potential Security Concerns

Figure 1 Realm of Potential Security Concerns
Appendix

Cyber Testing Matrix Definitions

Testing Types

Black Box Testing

Ad-hoc:
Definition from Wikipedia: “performed without planning and documentation”

Definition from the web: “informal type of software testing that is performed by software testers, business analyst, developers or any stake holder without referring to test cases or documentation.”

“It is performed by improvisation: the tester seeks to find bugs by any means that seem appropriate.”

Alpha:
Definition from Wikipedia: “developers generally test the software using white-box techniques. Additional validation is then performed using black-box or gray-box techniques, by another testing team.”

Definition from the web: “formal type of testing that is performed by end customers at development site.”

Beta:
Definition from Wikipedia: “The focus of beta testing is reducing impacts to users, often incorporating usability testing.”
Definition from the web: “formal type of software testing that is carried out by end customers before releasing or handing over software to end users.”

**Boundary Value:**
Definition from Wikipedia: “tests are designed to include representatives of boundary values in a range.”
Definition from the web: “testing is done extensively to check for defects at boundary conditions.”
AKA: Boundary-value Analysis

**Comparison:**
Definition from the web: “involves making a comparison of software weaknesses and strengths with competing software(s)/products.”
Definition from the web: “performed to compare features, pros and cons of the product with competitor products or different versions of the product.”

**Conformance:**
Definition from Wikipedia: “testing to determine whether a product or system or just a medium complies with the requirements of a specification, contract or regulation.”
“verifies that a product performs according to its specified standards.”
Definition from the web: “done to endorse a product that it meets the requirements it was supposed to fulfill.”
AKA: type testing

**Domain:**
Definition from Wikipedia: “It is a method of selecting a small number of test cases from a nearly infinite group of candidate test cases.”

**Equivalence Class:**
Definition from Wikipedia: “divides the input data of a software unit into partitions of equivalent data from which test cases can be derived.”
AKA: Equivalence Partitioning, Equivalence Class Partitioning

**Exploratory:**
Definition from Wikipedia: “seeks to find out how the software actually works, and to ask questions about how it will handle difficult and easy cases.”

**Happy Path:**
Definition from Wikipedia: “well-defined test case using known input, which executes without exception and produces an expected output.”
AKA: Happy day scenario, golden path

**Interface:**
Definition from the web: “performed to evaluate whether systems or components pass data and control correctly to one another. It is to verify if all the interactions between these modules are working properly and errors are handled properly.”

**Keyword-Driven**
Definition from Wikipedia: “This methodology uses keywords (or action words) to symbolize a functionality to be tested”
“A Keyword or Action Word is a defined combination of actions on a test object which describes how test lines must be executed. An action word contains arguments and is defined by a test analyst.”
Orthogonal Array:
Definition from Wikipedia: “used when the number of inputs to the system is relatively small, but too large to allow for exhaustive testing of every possible input to the systems.”

“instead of testing the system for each combination of parameters, we can use an orthogonal array to select only a subset of these combinations.”

Pairwise:
Definition from Wikipedia: “for each pair of input parameters to a system (typically, a software algorithm), tests all possible discrete combinations of those parameters.”

Definition from the web: “testing method where in for each input is tested in pairs of inputs, which helps to test software works as expected with all possible input combinations.”

AKA: All-pairs testing

Prescriptive:
Definition from the web: “A method by which test conditions are developed directly from the requirements, often before any actual code is written.”

Random
Definition from Wikipedia: “programs are tested by generating random, independent inputs. Results of the output are compared against software specifications to verify that the test output is pass or fail.”

System:
Definition from Wikipedia: “conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements.”

Compatibility Testing

Backward Compatibility:
Definition from the web: “performed to check newer version of the software can work successfully installed over previous version of the software and newer version of the software works as fine with table structure, data structures, files that were created by previous version of the software.”

Binary Portability:
Definition from the web: “tests compatibility of binary or executable files from one hardware and/or operating system to other.”

Browser Compatibility:
Definition from the web: “performed for web applications with combination of different browsers and operating systems.”

Certification:
Definition from the web: “product will be certified as fit to use.”

Conversion:
Definition from the web: “check programs or software used for converting data from existing systems to new or replacement systems.”

Forward Compatibility:
Definition from the web: “type of software testing that is done to check if the software works as expected when software is downgraded from higher version to lower version.”

AKA: Upward Compatibility Testing
Parallel:  
Definition from the web: “test two or more versions of the software “the current version” and “previous version or versions” of the software together to see the differences of existing functionality.”

Coverage Testing

Branch Coverage:  
Definition from the web: “method for designing test cases to test code for every branching condition.”

Condition Coverage:  
Definition from the web: “developer tests for all the condition statements like if, if else, case etc., in the code being unit tested.”

Data Flow Coverage:  
Definition from the web: “the control flowgraph is annotated with information about how the program variables are defined and used. Different criteria exercise with varying degrees of precision how a value assigned to a variable is used along different control flow paths.”

Decision Coverage:  
Definition from the web: “expertise and validate each and every decisions made in the code.”

End-to-End:  
Definition from the web: “involves testing information flow across applications.”

Fault Injection:  
Definition from the web: “Faults or errors are induced in the application and existing tests are executed, in case the introduced errors are caught, that means existing tests are good and robust else additional tests have to be included.”

Path Coverage:  
Definition from the web (basis path testing): “method for designing test cases to test code for every execution path of code at least once.”

AKA: Basis path testing

Operational Readiness Testing

Acceptance:  
Definition from the web: “formal type of software testing that is performed by end customer to check if the software confirms to their business needs and to the requirements provided earlier.”

Accessibility:  
Definition from the web: “formal type of software testing that helps to determine whether the software can be used by people with disability.”

Installation/Un-Installation:  
Definition from the web: “type of testing performed by testing team to find defects related to installation and un-installation of software.”

Model-Based:  
Definition from Wikipedia: “application of model-based design for designing and optionally also executing artifacts to perform software testing or system testing. Models can be used to represent the desired behavior of a System Under Test (SUT), or to represent testing strategies and a test environment.”
**Operational Readiness:**
Definition from the web: “intends to validate the production environment after new version of the software is deployed in production environment.”

AKA: ORT, pre go-live testing

**Specification-Based:**
Definition from the web: “testing against any published claim about how the program is supposed to work.”

**Upgrade:**
Definition from the web: “more often carried out after an hardware upgrade or operating system upgrade or any other supporting software.”

**Functional Testing**

**Failover:**
Definition from the web: “focuses on testing functionality or features that enable software to recover from some of the anticipated failures.”

**Gorilla:**
Definition from the web: “exercise one or few functionality thoroughly or exhaustively by having multiple people test the same functionality.”

**GUI:**
Definition from the web: “aimed at testing the software GUI (Graphical User Interface) of the software meets the requirements as mentioned in the GUI mockups and Detailed designed documents.”

**Regression:**
Definition from the web: “find defects that got introduced to defect fix(es) or introduction of new feature(s).”

**Sanity:**
Definition from the web: “quick evaluation of the software, environment, network, external systems are up & running, software environment as a whole is stable enough to proceed with extensive testing.”

**Smoke:**
Definition from the web: “intended to find “show stopper” defects that can prevent testers from testing the application in detail.”

AKA: build verification test

**Usability:**
Definition from the web: “performed to understand how user friendly the software is.”

**Integration Testing**

**Big Bang Integration:**
Definition from the web: “all or all most all of the modules are developed and then coupled together.”

**Bottom up Integration:**
Definition from the web: “testing starts with smaller pieces or sub systems of the software till all the way up covering entire software system.”

**Hybrid Integration:**
Definition from the web: “combination of both bottom up and top down integration testing techniques. By combining both of these once can find more defects at the same time provide better integration test coverage.”

AKA: sandwich testing
Note: Covered under bottom-up and top-down. Can it be removed from the list?

**System Integration:**
Definition from the web: “test for errors related to integration among different applications, services, third party vendor applications etc.”

**Top Down Integration:**
Definition from the web: “incremental testing approach for integration testing where in testing of top level modules are done first before moving on to testing of branch modules.”

**Nonfunctional Testing**

**Compliance:**
Definition from the web: “test software meets the required standards, government laws, company policies etc.”

**Database:**
Definition from Wikipedia: “many kinds of implementation and integration errors may occur in large database systems, which negatively affect the system's performance, reliability, consistency and security. Thus, it is important to test in order to obtain a database system which satisfies the ACID properties (Atomicity, Consistency, Isolation, and Durability) of a database management system.”

**Documentation:**
Definition from the web: “project documents are checked for thoroughness and completeness.”

**Globalization:**
Definition from the web: “detects problems with in application design related to usage of the software with different languages, different character sets.”

AKA: Internationalization Testing, I18n Testing, Localization Testing

**Recovery:**
Definition from the web: “aims at checking how soon and how efficiently software can recover from software crashes, Operating system crashes, and hardware failures.”

**Resilience:**
Definition from the web: “assess how stable the software is when it is subject to incorrect data, large workloads and more volume of data to be processes.”

**Static:**
Definition from the web: “software code is not executed instead it is reviewed for syntax, commenting, naming convention, size of the functions and methods etc.”

**Volume:**
Definition from the web: “carried out to find the response of the software with different sizes of the data being received or to be processed by the software.”

**Performance Testing**

**Aging:**
Definition from the web: “type of performance testing that is carried out by running software for longer duration like weeks or months and check performance of software to see if the software performance degrades or shows any signs of degradation after running for a longer period of time.”

AKA: soak testing, longevity testing, endurance testing
**Benchmark:**
Definition from the web: “measure performance of different versions of the software with predefined configuration.”

**Concurrency:**
Definition from the web: “check if application performs as expected when multiple users use the software, logins, features and database etc.”

**Configuration:**
Definition from the web: “finding optimal configuration settings that can make software perform at its best for hardware or for a given operating system.”

**Load:**
Definition from the web: “done to check the behavior of the software under normal and over peak load conditions.”

**Ramp:**
Definition from the web: “conducted to check the response of the software with constant increase in workload on the software.”

**Scalability:**
Definition from the web: “intended to test one of the software quality attributes i.e. “Scalability”. Scalability test is not focused on just one or few functionality of the software instead performance of software as a whole.”

**Stability:**
Definition from the web: “focuses on testing how stable software is when it is subject to loads at acceptable levels, peak loads, loads generated in spikes, with more volumes of data to be processed.”

**Stress:**
Definition from the web: “software is subjected to peak loads and even to a break point to observe how the software would behave at breakpoint. Stress testing also tests the behavior of the software with insufficient resources like CPU, Memory, Network bandwidth, Disk space etc.”

**Spike:**
Definition from the web: “check how software responds to workloads that are sent in very short span of time and which are not constant over period of time.”

**Security Testing**

**Penetration:**
Definition from the web: “tests how secure software and its environments (Hardware, Operating system and network) are when subject to attack by an external or internal intruder.”

AKA: pentest

**Risk Based:**
Definition from the web: “requirements and functionality of software to be tested are prioritized as Critical, High, Medium and low. In this approach, all critical and High priority tests are tested and them followed by Medium. Low priority or low risk functionality are tested at the end or may not based on the time available for testing.”

**Vulnerability:**
Definition from the web: “involves identifying, exposing the software, hardware or network Vulnerabilities that can be exploited by hackers and other malicious programs likes viruses or worms.”
Unit Testing

API:
Definition from the web: “type of testing that is similar to unit testing. Each of the Software APIs are tested as per API specification.”

Error Handling:
Definition from the web: “focuses on error handling capabilities of software.”

Monkey:
Definition from the web: “Objective of monkey testing is to use the software without any specific tests in mind. Monkey test tries to break the software by entering incorrect dates like 31-Feb-2012 or long strings of text or numbers or special characters etc.”

Negative:
Definition from the web: “functional and non-functional tests that are intended to break the software by entering incorrect data like incorrect date, time or string or upload binary file when text files supposed to be upload or enter huge text string for input fields etc.”

AKA: Fuzz Testing

Note: While this is similar to monkey testing, I think it’s important enough that it should be split out by itself.

White Box Testing

Code-Driven:
Definition from the web: “related to types of testing that makes uses of frameworks.”

Component:
Definition from the web: “involves testing a group of units as code together as a whole rather than testing individual functions, methods.”

Destructive:
Definition from the web: “intended to find failure points of a software, be it by inputting incorrect data, inputting corrupt data, incorrect format, by volume or by number requests etc.”

Security Areas

Miscellaneous

Phishing/Pharming:
Definition from textbook (phishing): “hybrids that combine both social engineering and malicious software.”

Definition from textbook (pharming): “attack for implementing an MITM attack where the host’s file on the client system is modified to resolve the target address to the address of the hostile intermediary.”

Bot/Botnet:
Definition from textbook (bot): “automated software program that executes certain commands when it receives a specific input.”

Access Controls

Account Management:
Definition from textbook: “The organization manages information system accounts; their creation, activation, modification, review, disabling, and deletion. The accounts are reviewed periodically and at least annually.”
Password Cracking:
Definition from textbook: “attempt to reverse-calculate a password.”

Brute Force Attack:
Definition from textbook: “attempt to try every possible combination of options for a password.”

Dictionary Attack:
Definition from textbook: “variation of the brute force attack that narrows the field selecting specific target accounts and using a list of commonly used passwords (the dictionary) instead of random combinations.”

Account Enforcement:
Definition from textbook: “The information system enforces authorizations in controlling access to the system in accordance with policy.”

Information Flow Enforcement:
Definition from textbook: “The system enforces authorizations in controlling information flows within the system and between interconnected systems in accordance with policy.”

Least Privilege:
Definition from textbook: “The system enforces the minimal set of privileges required to perform a specific task.”

Unsuccessful Logon Attempts:
Definition from textbook: “The system enforces a specified limit on the number of invalid access attempts by a user over a specified period of time. When the limit is exceeded, the system will either lock the affected account or impose a delay before another logon can be attempted.”

System Use Notification:
Definition from textbook: “The system displays an approved system use notification message before granting system access. This notification may take the form of a login banner that informs the user that this is a private system, access is by permission only, activities may be monitored and recorded, etc.”

Previous Logon Notification:
Definition from textbook: “The system notifies the user upon successful logon of the date and time of the last successful logon and the number of unsuccessful logon attempts since that successful logon.”

Concurrent Session Limit:
Definition from textbook: “The system limits the number of concurrent sessions to a specified limit.”

Session Lock:
Definition from textbook: “After a specified period of inactivity, the system prevents further access by initiating a session lock that remains in effect until the user reestablishes access using appropriate identification and authorization procedures.”

Session Termination:
Definition from textbook: “After a specified period of inactivity, the system automatically terminates a remote session.”

Permitted Actions without Identification or Authentication:
Definition from textbook: “The organization identifies and documents specific user actions that can be performed without identification or authentication.”

Automated Marking:
Definition from textbook: “The system marks output using standard naming conventions to identify any special dissemination, handling, or distribution restrictions.”
Automated Labeling:
Definition from textbook: “The system appropriately labels information in storage (at rest), in process (in use), and in transmission (in flight).”

Remote Access:
Definition from textbook: “The organization authorizes, monitors, and controls all methods of remote access to the information system.”

Wireless Access Restrictions:
Definition from textbook: “The organization both establishes usage restrictions and implementation guidance for wireless technologies and authorizes, monitors, and controls wireless access.”

Access Controls for Portable and mobile Devices:
Definition from textbook: “The organization: (i) establishes usage restrictions and implementation guidance for wireless technologies; and (ii) authorizes, monitors, and controls wireless access to the information system.”

Use of External Information Systems:
Definition from textbook: “The organization: (i) establishes usage restrictions and implementation guidance for organization-controlled portable and mobile devices; and (ii) authorizes, monitors, and controls device access to organizational information systems.

Audit and Accountability Controls

Auditable Events:
Definition from textbook: “The information system generates audit records for the specified events. Moderate impact systems must additionally periodically review and update the list of auditable events.”

Content of Audit Records:
Definition from textbook: “The information system produces audit records that contain sufficient information to establish what events occurred, the sources of the events, and the outcomes of the events.”

Audit Storage Capacity:
Definition from textbook: “The organization allocates sufficient audit record storage capacity and configures auditing to reduce the likelihood of such capacity being exceeded.”

Response to Audit Processing Failures:
Definition from textbook: “The information system alerts appropriate organizational officials in the event of an audit processing failure and takes specified actions (e.g., shut down the system, overwrite oldest log records, etc.).”

Audit Monitoring, Analysis, and Reporting:
Definition from textbook: “The organization regularly reviews/analyzes information system audit records for indications of inappropriate or unusual activity, investigates suspicious activity or suspected violations, reports findings to appropriate officials, and takes necessary actions.”

Audit Reduction and Report Generation:
Definition from textbook: “The information system provides an audit reduction and report generation capability.”

Time Stamps:
Definition from textbook: “The information system provides time stamps for use in audit record generation.”

Protection of Audit Information:
Definition from textbook: “The information system protects audit information and audit tools from unauthorized access, modification, and deletion.”
Nonrepudiation:
Definition from textbook: “The information system provides the capability to determine whether a given individual took a particular action. This is a core support for providing accountability.”

Audit Record Retention:
Definition from textbook: “The organization retains audit records for an appropriate, specified period of time to provide support for after-the-fact investigations of security incidents and to meet regulatory and organizational information retention requirements.”

Identification and Authentication Controls

User Identification and Authentication:
Definition from textbook: “The information system uniquely identifies and authenticates users (or processes acting on behalf of users).”

Device Identification and Authentication:
Definition from textbook: “The information system identifies and authenticates specific devices before establishing a connection.”

Identifier Management:
Definition from textbook: “The organization manages user identifiers by: (i) uniquely identifying each user; (ii) verifying the identity of each user; (iii) receiving authorization to issue a user identifier from an appropriate organization official; (iv) issuing the user identifier to the intended party; (v) disabling the user identifier after a specified period of inactivity; and (vi) archiving user identifiers.”

Authenticator Management:
Definition from textbook: “The organization manages information system authenticators by: (i) defining initial authenticator content; (ii) establishing administrative procedures for initial authenticator distribution, for lost/compromised or damaged authenticators, and for revoking authenticators; (iii) changing default authenticators upon information system installation; and (iv) changing/refreshing authenticators periodically.”

Authenticator Feedback:
Definition from textbook: “The information system obscures feedback of authentication information during the authentication process to protect the information from possible exploitation/use by unauthorized individuals.”

Cryptographic Module Authentication:
Definition from textbook: “The information system employs authentication methods that meet the requirements of applicable laws, executive orders, directives, policies, regulations, standards, and guidance for authentication to a cryptographic module.”

Media Protection Controls

Media Access:
Definition from textbook: “The organization restricts access to information system media to authorized individuals.”

Media Labeling:
Definition from textbook: “The organization: (i) affixes external labels to removable information system media and information system output indicating the distribution limitations, handling caveats and applicable security markings (if any) of the information; and (ii) exempts a specified list of media from labeling as long as they remain within a specified protected environment.”

Media Storage:
Definition from textbook: “The organization physically controls and securely stores information system media within controlled areas.”
Media Transport:
Definition from textbook: “The organization protects and controls information system media during transport outside of controlled areas and restricts the activities associated with transport of such media to authorized personnel.”

Media Sanitization and Disposal:
Definition from textbook: “The organization sanitizes information system media, both digital and nondigital, prior to disposal or release for reuse.”

System and Communication Protection Controls

Application Partitioning:
Definition from textbook: “The information system separates user functionality (including user interface services) from information system management functionality.”

Security Function Isolation:
Definition from textbook: “The information system isolates security functions from nonsecurity functions.”

Information Remanence:
Definition from textbook: “The information system prevents unauthorized and unintended information transfer via shared system resources.”

Denial-of-Service Protection:
Definition from textbook: “The information system protects against or limits the effects of relevant or common denial-of-service attacks.”

Definition from textbook (denial of service attack): “occurs when the attacker sends a large number of connection or information requests to a target that will either crash or overload the system and cannot respond to legitimate requests for service.”

Resource Priority:
Definition from textbook: “The information system limits the use of resources by priority.”

Boundary Protection:
Definition from textbook: “The information system monitors and controls communications at the external boundary of the information system and at key internal boundaries within the system.”

Transmission Integrity:
Definition from textbook: “The information system protects the integrity of transmitted information.”

Transmission Confidentiality:
Definition from textbook: “The information system protects the confidentiality of transmitted information.”

Network Disconnect:
Definition from textbook: “The information system terminates a network connection at the end of a session or after a specified period of inactivity.”

Trusted Path:
Definition from textbook: “The information system establishes a trusted communications path between the user and significant security functions.”

Cryptography:
Definition from textbook: “When cryptography is required and employed within the information system, the organization establishes and manages cryptographic keys using automated mechanisms with supporting procedures or manual procedures.”
Use of Cryptography:
Definition from textbook: “For information requiring cryptographic protection, the information system implements cryptographic mechanisms that comply with applicable laws, executive orders, directives, policies, regulations, standards, and guidance.”

Public Access Protections:
Definition from textbook: “The information system protects the integrity and availability of publicly available information and applications.”

Collaborative Computing:
Definition from textbook: “The information system prohibits remote activation of collaborative computing mechanisms and provides an explicit indication of use to the local users.”

Transmission of Security Parameters:
Definition from textbook: “The information system reliably associates security parameters with information exchanged between information systems.”

Public Key Infrastructure Certificates:
Definition from textbook: “The organization issues public key certificates under an appropriate certificate policy or obtains public key certificates under an appropriate certificate policy from an approved service provider.”

Mobile Code:
Definition from textbook: “The organization: (i) establishes usage restrictions and implementation guidance for mobile code technologies based on the potential to cause damage to the information system if used maliciously; and (ii) authorizes, monitors, and controls the use of mobile code within the information system.”

Voice Over Internet Protocol:
Definition from textbook: “The organization: (i) establishes usage restrictions and implementation guidance for Voice over Internet Protocol (VoIP) technologies based on the potential to cause damage to the information system if used maliciously; and (ii) authorizes, monitors, and controls the use of VoIP within the information system.”

Secure Name/Address Resolution Service:
Definition from textbook: “The information system that provides name/address resolution service provides additional data origin and integrity artifacts along with the authoritative data it returns in response to resolution queries.”

Architecture and Provisioning for Name/Address Resolution Service:
Definition from textbook: “Service—The information systems that collectively provide name/address resolution service for an organization are fault tolerant and implement role separation.”

Session Authenticity:
Definition from textbook: “The information system provides mechanisms to protect the authenticity of communications sessions.”

System and Information Integrity Controls

Flaw Remediation:
Definition from textbook: “The organization identifies, reports, and corrects information system flaws.”

Malicious Code Protection:
Definition from textbook: “The information system implements malicious code protection.”
Information System Monitoring Tools and Techniques:
Definition from textbook: “The organization employs tools and techniques to monitor events on the information system, detect attacks, and provide identification of unauthorized use of the system.”

Security Alerts and Advisories:
Definition from textbook: “The organization receives information system security alerts/advisories on a regular basis, issues alerts/advisories to appropriate personnel, and takes appropriate actions in response.”

Security Functionality Verification:
Definition from textbook: “The information system verifies the correct operation of security functions periodically as needed and takes appropriate action when anomalies are discovered. An example of an anomaly would be a cryptographic appliance that suddenly began transmitting clear text instead of cipher text.”

Software and Information Integrity:
Definition from textbook: “The information system detects and protects against unauthorized changes to software and information.”

Spam Protection:
Definition from textbook: “The information system implements spam protection.”

Information Input Restrictions:
Definition from textbook: “The organization restricts the capability to input information to the information system to authorized personnel.”

Information Accuracy, Completeness, Validity, and Authenticity:
Definition from textbook: “The information system checks information for accuracy, completeness, validity, and authenticity.”

Error Handling:
Definition from textbook: “The information system identifies and handles error conditions in an expeditious manner without providing information that could be exploited by adversaries.”

Information Output Handling and Retention:
Definition from textbook: “The organization handles and retains output from the information system in accordance with applicable laws, executive orders, directives, policies, regulations, standards, and operational requirements. The new Federal Rules of Civil Procedure (FRCP) explicitly recognize the importance of electronically stored information (ESI) and this creates a responsibility on a party to a lawsuit to preserve all relevant ESI. Thus, the organization must have policy and procedures in place to modify retention requirements for ESI relevant to pending litigation.”
References


Formal Methods and Optimization of Software Test Cases

Rodrigo Silva-Lugo
Computer Science Department
University of Memphis
Memphis, TN 38152
URL: http://umpeople.memphis.edu/rsilva

Abstract—A comprehensive approach to software test case optimization is proposed, including a review of optimization techniques, and their application to three problems: generation of test cases, selection of the test cases, and prioritization of the execution of test cases.

I. INTRODUCTION

Software test cases are a key element in the development of modern software systems. They provide a way to manage and execute changes in large software projects in a way that the changed code will still comply with the initial specifications.

This paper is a research-in-progress report that proposes a comprehensive approach to software test case optimization, including a literature review of test case optimization techniques, and how they can be applied to problems such as: 1) selection of the test cases, 2) prioritization of the execution of test cases, and 3) generation of test cases.

This paper is organized as follows: Section II describes the research question and summarizes the current state and application of test case optimization. The bulk of this work is section III, which presents a review of the literature in the area of test case optimization. Finally, Section V describes potential directions for further research.

II. RESEARCH QUESTION

The purpose of this research is to integrate the concepts of formal methods used in the context of test case generation, with the optimization of test case selection and prioritization. Examples of test case generation can be found in [1], [2], and [3]; there have also been studies about how to efficiently use these concepts [4], and their potential application to real world problems [5], and [3].

The illustration of the proposed approach is illustrated below. Figure 1 shows a conceptual diagram of the problem, possible solutions, and how these are related.

Figure 2 shows a more specific approach to the problem. The four proposed phases are:

1) Generation of test cases from specification.
2) Selection of test cases.
3) Prioritization of test cases.
4) Integration of all the problems in a comprehensive framework, and optimization.

In summary, the research question is: How can we integrate these four fields in a coherent and efficient way, so formal methods and optimization can be applied to the software testing of large-scale, real-world implementations?

III. LITERATURE REVIEW

For the purpose of this proposal, the optimization problems have been divided into two categories: selection/prioritization of test cases, and generation of test cases. In test case selection and prioritization, the goal is to select and/or order a group of test cases, from a larger set, such that the execution of the chosen test cases is optimized given the constrained resources available (e.g. time, computational power). On the other hand, the generation problem consists in creating the test cases such that their execution is optimal, and they satisfy the software specification.

The optimization techniques used for these two problems include Genetic Algorithms (GAs) [6], [7], Particle Swarm Optimization (PSO) [8], [9], Ant Colony Optimization (ACO) [10], Integer Programming (IP) [11], [12] among others.

Table I shows a list of the reviewed papers, and what topics and techniques each one of them covers.

Figure 1 shows the conceptual representation of the areas covered in the rest of this paper, and the relationships between them. The initial idea is to integrate formal methods and software test case optimization techniques in a comprehensive and efficient framework, that could be used in real-world development scenarios.

The rest of this section analyzes the different approaches that were found to solve the two optimization problems, i.e. selection / prioritization, and generation of test cases.

A. Selection & Prioritization of Test Cases

In [13] the authors present a technique for the optimization of test case selection and prioritization based on IP. In this case the number of cases to be selected is fixed, and is part of the IP problem constraints. The IP optimization uses relaxation and two different coverage criteria. The results obtained were mixed, since the proposed technique only outperformed the others in 40% of the cases, and was outperformed in 20% of the cases. There seemed to be a quite small number of tests (only 10) and the number of programs was even smaller (5 Java programs). There are a few gaps in the paper, such as the lack of understanding of the variation in performance, the absence
of test programs with real faults, and the lack of consideration for test cases running times, which we assume should be part of the cost considered when performing both regression test selection and prioritization.

A different approach was used by [14], [15], [16], in this case the optimization technique used was ACO applied to selection and prioritization. The authors found the technique to give optimal fault coverage, which would result in reduced costs and time associated with testing. However their optimization technique was only applied to a very small scenario, i.e. 8 test cases with at most 10 faults, and moreover, the faults were known in advance. This technique may yield close to optimal results in selection and prioritization, however it is not clear if its performance will scale when applied to real world problems, i.e. larger number of test cases with potentially unknown faults.

In [17] the authors used multi-objective optimization for the selection of test cases. The technique was based in PSO, and the purpose was to maximize the coverage of the test cases, and minimizing the effort (time) of the cases execution. Two variants of PSO were tested, and compared against a random search algorithm, The Crowd Distance Roulette variant of PSO outperformed both regular PSO and the random search algorithm in all but one of the three evaluated metrics. This paper used a much larger set of test cases (2 suites, 80 cases each) and the results presented were based on 60 runs of the search algorithms for each suite. The paper presents a nice formal definition of the algorithms and the metrics used to measure the quality of the solution, and may be a good start for the application of formal methods in the test case optimization domain. This can be done in conjunction with the proposed further work of using the technique in other domains besides mobile telephony test cases, and using other metrics for measuring optimality, as well as comparison with other search techniques.

PSO was also used by [18] to optimize the prioritization of test cases, in this case, in the domain of embedded real time software systems. In this case, the authors used PSO to sort the execution order of 20 test cases taken from the JUnit Test Suite. The results indicated that PSO had a better coverage (64%) at once 50% of the test cases had been run, when compared with a greedy algorithm alternative (48%). The proposed PSO technique also proved to be more efficient in terms of time complexity. However, the test cases used for the experiment seemed to be too basic, and too general. It would be interesting to see the performance of this technique when applied to more realistic test cases, such as the ones presented in [17].

Finally, for the problem of selection, [19] present a hybrid optimization method based on GA and local search. They apply this technique to a set of test cases designed to test several pieces of software, both academic and industrial, and compare their proposed Hybrid Genetic Algorithm (HGA) with existing GA and Bacteriological Algorithm (BA) approaches. Their results suggest that the GA-only approach tends to get stuck in local optima, and that HGA is better suited when the goal is to reach global optima. However, the HGA solution seemed to converge too rapidly sometimes (around 10 generations, out of a maximum of 200) and this may call for a closer look at the topology of the search space and the algorithms parameters. Also, it would be worth investigating if a further reduction of the test case set, which they measured at around 80%, could be achieved by adjusting the algorithms parameters.

The techniques presented in this section, for the optimization of prioritization and selection of software test cases, seem to favor the use of evolutionary, multi-objective approaches. In most of the cases they outperformed their classical counterparts, both in terms of correctness and time complexity. However, it is also true that most of these works were tested only with too small, or too basic test cases or target software, and in some of this cases, the proposed solution would converge too quickly, which raises the question of why using such a complex technique in the first place.

B. Test Case Generation

The generation of test cases is presented in [20], here the authors use CO to minimize the number of test generated cases, while maximizing their coverage of faults. The procedure
consists in translating the text case generation problem into a CO one, then solving it, and the result is then translated into a test case. They showed results of the technique applied to several benchmark problems, and provided a detailed, somewhat formal description of the process. However, the solver implemented seemed to be a bit slow when its performance was measured in the simulations, and since the results shown are for benchmark problems, it would be interesting to see how this approach performs when applied to real-world software testing problems. Also the detailed formal definitions of the problem and the technique would be helpful in case that a formal method approach were to be applied.

The use of evolutionary optimization techniques for test case generation is extensive. In [21] the authors use a Memetic Algorithm (MA) in conjunction with Cuckoo Search (CK) in order to optimize the number of test cases generated for the given Software Under Test (SUT). To do this, they first create a set of mutants based on the original SUT, then apply randomly generated test cases to these, and use CK to find the optimal cases. This is an iterative process, so the optimal cases are used as the next generation and the process is repeated. The results shown seemed very promising, although little detail was given about how the performance figures were obtained. Finally, the software used to obtain the results, was taken from a synthetic set, so it would be interesting to see if this technique claimed performance is still good when applied to real-world software. Other ideas related to the CK technique might be of use for the problem of test case prioritization.
GAs are used in [22] to generate optimized test cases for critical systems. The authors proposed a framework was based in a mutant testing paradigm, and used two measures of coverage to calculate the fitness of the generated cases. The framework was tested with a benchmark problem in avionics. The performance of the GA approach was then compared with Taguchis method for experiment design, and showed no improvement. Furthermore, there were no performance benefits discussed, so there might be plenty of room for further exploration, i.e. use of other measures for fitness besides coverage, application of the framework to real-world problems, or in other critical domains (e.g. nuclear power).

The use of GAs for the generation of test cases is also presented in [23]. The authors used a dynamic fitness function that can focus on three elements of the test case population: novelty of the test case, proximity to other test cases, and severity of the failure of a particular test. They also introduced an interesting concept of fossil record, where past populations of the GA are stored, and aided by visualization techniques, can be used for the analysis of the fitness landscape. The authors also presented a brief classification of previous research in software testing by the fitness function used, this might prove useful for our own problem. Other aspect that may be explored, is analyzing the efficiency of the fossil record implementation, since due to its size, it may become unmanageable for large software testing problems. Finally it would be interesting to explore these ideas applied to real-world test case optimization, since the benchmark problem used in this paper (triangle classification) is more academic than practical.

In [24] GAs are also used for the optimization of test case generation. In this case the authors added tabu search to the mutation operation to avoid local optima stagnation. A general description of the GA was given, as well as the description of the population used in the simulation. However, it was not clear how the initial population of test cases was generated, and since this population was used to generate the rest of test cases, it is a critical gap in the papers argument. It may be worth exploring the role of tabu search, and other techniques to help GAs avoid local optima stagnation, especially when applying this technique to real-world large test case sets.

Finally, [25] proposes to use an evolutionary algorithm to optimize generated test cases. It is not clear however if the technique used is truly an evolutionary one, or a greedy algorithm, since the latter is mentioned swiftly as the optimization process. The results shown are confusing and it was not clear what the purpose of the optimization was. It may be worthwhile to explore the tool mentioned by the authors (TCGOJ), although no reference was given for it in the paper. Finally, the manual addition of assertions during the optimization process raises questions about the possibility of automation or escalation to larger suites/problems, although the authors mention this is one place where further work can be done.

A slightly different approach was used by [26], [27], and [28]. In this case, the authors examined both a GA and BA approach, and tested them on a small implementation of a parser in the C# programming language. The BA algorithm differs from a regular GA in that no crossover operators are used, and also it implements a “best individual” memory, which seemed very similar to the concept of elitism in GAs. The results obtained were better when compared with a GA implementation of the same problem: BA detected ten times more failures; BA converged faster (30 generations vs. 200 for GA); and BA reached a higher fitness value (95% vs 90%). Despite the good results shown in the paper, it seemed the technique used to generate mutants from the original code, only introduces syntactic failures, and higher level logical failures would never be detected by this technique. Also the lack of crossover operations raises the question of proper exploitation of good solutions, although this may be alleviated by the use of memorization of the best individuals. Finally, it would be worth exploring how the proposed model performs in other domains different from the parsing example presented, and how it could be extended to more abstract and high level operators for the generation of mutants from the original code.

IV. FORMAL METHODS

In [1], the authors propose an approach for the automatic generation of test cases from formal specifications. They describe what the test strategy is, what the required criteria are, and give a set of algorithms to generate the tests in different scenarios. All these procedures assume that there is a formal specification available for the software to be tested. The proposed approach was implemented and compared with a random test approach. The performance of the proposed technique was better than the random one, which is to be expected (72% coverage vs. 26% coverage), however, it was not clear if the difference was due to the superiority of the technique, or the flaws of the random test specification. In addition, the proposed technique can work well only if some constraints are satisfied, e.g. non-linearity of arithmetic expressions involving input variables would be a problem, potentially impossible to test with the approach proposed by the authors. It would also be of interest to test this approach with other kinds of problems, closer to real-world ones, explore other measures of effectiveness for the generated test cases besides coverage, and check the performance against other test methods besides the random one used in the paper.

A different approach is presented in [2], where the authors describe a theoretical technique for the generation of test cases, from an existing formal specification. For this purpose, they use the Counterexample Guided Abstraction Refinement (CEGAR) in combination with partition refinement. An ab-struct model is obtained from the formal specification (in this case assumed to be in the B language), and then used to generate a state transition graph, for which paths represent test cases for the original software. Although this is a purely theoretical paper, a couple of small examples are shown for problems using flat B specification, if the the proposed approach could be extended to more complex problems and other specification languages easily, it could be extended to larger implementations, and therefore possibly applied to real-world problems.

For application of formal methods in real-world systems, [5] present an approach to automate test case generation using formal methods and commercially available tools. This approach is used in the domain of avionics, and two implementations were done in flight control, and flight warning systems. The technique obtained the test cases from functional
test objectives, which in turn were generated from the system to be tested. The coverage of such test cases was then analyzed and found to be satisfactory for the systems requirements. The claim of the authors is that formal proof and automated test case generation are feasible in practice, however no comparison with manually generated test cases is presented. The authors also suggest that the coverage analysis at specification level might be necessary, instead of at code level, and this makes perfect sense since the code should ideally be obtained by implementation from the formal specification. Finally, the proposed techniques in this paper might be of great interest for other critical systems, such as distribution and logistics, and if the necessary tools and formal specifications are available, it would prove very useful for the purposes of the research presented in this proposal.

A list of techniques in the reviewed literature is shown in Table I.

V. CONCLUSION

This paper proposed a comprehensive approach to software test case optimization, and a literature review of test case optimization techniques, and how they can be applied to specific software test case optimization problems such as selection, prioritization, and generation.

There is plenty of room for improvement in this area, especially when it comes to the use of real-world test cases and software.

Also, the use of formal methods in this case may be tested for the measure of test case coverage, when compared against the target software formal specification, and also as a mean to generate the test cases based on the software requirements and specification.

ACKNOWLEDGMENTS

This work would not have been possible without the support and collaboration of: Dr. Sajjan Shiva (Computer Science), Dr. Robin Poston (Business Information Technology), and Dr. Jasbir Dhaliwal (FedEx Institute).

ACRONYMS

ACO Ant Colony Optimization.
BA Bacteriological Algorithm.
CEGAR Counterexample Guided Abstraction Refinement.
CK Cuckoo Search.
GA Genetic Algorithm.
HGA Hybrid Genetic Algorithm.
IP Integer Programming.
MA Memetic Algorithm.
PSO Particle Swarm Optimization.
SUT Software Under Test.

REFERENCES


<table>
<thead>
<tr>
<th>What/How</th>
<th>Sel</th>
<th>Pri</th>
<th>Gen</th>
<th>FM</th>
<th>GA</th>
<th>GEO</th>
<th>BA</th>
<th>MA</th>
<th>CK</th>
<th>IP</th>
<th>LS</th>
<th>ACO</th>
<th>ABC</th>
<th>PSO</th>
<th>CO</th>
</tr>
</thead>
<tbody>
<tr>
<td>zhang14</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>jeyamala14</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ahmed14</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>samatha12</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mirarab12</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lam12</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>yano11</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sur11a</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sur11b</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>de souza11</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>singh10</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>jeyamala10</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lin10</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mohapatra09</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>hla08</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>windisch07</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>devaranana09</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>saipathy09</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>baudry05</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bernard04</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>legeard04</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>durrieu04</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bernard03</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>baudry02a</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>baudry02b</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>