Designing Influence Metric at the Architectural Level for Improving the Reliability of a System

Mitrabinda Ray
CS & E Dept.,
National Institute of Technology, Rourkela.

Durga Prasad Mohapatra
CS & E Dept.,
National Institute of Technology, Rourkela.

ABSTRACT
Though some components play a major role for enhancing the quality of a system, but exactly identifying those components at the early stage is a big challenge. Metrics that are designed at the early stage guide both the test manager and the system analyst in decision making. In this paper, we propose an *Influence Metric* at the architectural level to get the influence of a component towards the system failures. First, we generate an intermediate graph called Sequence Diagram Graph (SDG) for a sequence diagram and compute the occurrence probability of each event within the sequence diagram based on *operational profile* of the system. Then, we propose an algorithm called Influence Computation Algorithm (ICA) to compute the influence of a component within a use case and within the whole system. The influence of a component *c* is decided by checking how many components are calling directly or indirectly the component *c* and the probabilities of their call to *c*. A component with high influence value is more sensitive towards system failures. The influence metric is applied on two well known case studies and the sensitivity analysis is conducted through a set of experiments to validate our approach.

Keywords
Operational Profile, Sequence Dependence Graph, Influence Metric;

1. INTRODUCTION
The main aim of testing is to achieve high quality system within the available test budget. For this, the test activities should be planned well in advance. Bugs in some components cause more frequent and more severe failures compared to those in others. The user's view on the reliability of a system is improved, when bugs which occur in the most frequently used parts of the source code is almost removed [1-4]. However, the length of time a part of the source code is executed does not wholly determine the importance of the part in the perceived reliability of the system. It is possible that the result produced by a component which is executed only for a small duration is saved and used by many components with high execution probability. Hence, a component would have a very high impact on the reliability of the system even though it is itself getting executed only for a small duration. This concept motivates us to consider the interaction among components through coupling and compute the *influence metric* at various stages of software development life cycle. In our previous work [5, 6], we have proposed algorithms to compute the static and dynamic influence of a component at the implementation level.

Metrics designed at the early stage guides both the test manager and the system analyst in decision making. At the early stage, it is required to make a decision about what to test more or test less within the available test budget. Sensitivity analysis at the architectural level saves both the development and testing cost [7]. We study the sensitivity of the application reliability towards the reliabilities of its elements within the system and take a valuable decision on allocating test resources to various components. Conducting sensitivity analysis at the early stage and allocating test resources accordingly helps to get a more reliable system within the available test budget and also improves the customer's satisfaction on the reliability of the system.

We are motivated to identify the critical components[1] in advance based on the data collected up to the architectural level and plan for testing accordingly. Our aim is to develop a novel approach that will enable the software analysts and the test manager to:

- Compute the influence of a component within a use case.
- Compute the influence of a component within a system.
- Generate a list of components ranked by their relative influence values.

We propose an algorithm to compute the influence of a component towards the system failure at the architectural level. The influence value of a component is decided through the analysis of behavioral dependencies among components within various use cases and Operational Profile designed for the system. First, all scenarios of a use case are derived from the sequence diagram that is designed for the use case and then, the scenarios are integrated into a graph called Sequence Diagram Graph (SDG) [8]. SDG helps to find the impact of an event within a use case. Then, the events are prioritized within the SDG based on their probability of occurrence within the use case. Based on the priority values of events, influence value of a component within a use case is computed by applying forward *slicing technique* on SDG. A forward slice provides the answer to the question which statements of a program will be affected by the slicing criterion?" [9]. The influence value of a component for the overall system is computed by considering all use cases within the system. The components within a system are prioritized for testing based on their influence values within the system. For achieving a higher reliable system, test effort

[1] A part is critical if a bug in the part is responsible for frequent failures.

should be assigned to various components within a system according to their test priority. This work is the extension of our previous work [10]. In the previous work, we have prioritized the components based on their influence values and allocated test efforts to various components according to their priority to obtain a high reliable system. For this, we have proposed a test case selection schema based on *Genetic Algorithm* that selects a given number of test cases from a pool of test cases to obtain high reliability of a system. In this paper, we have conducted a sensitivity analysis through a set of experiments and rank the components according to their influence values within a use case and within the whole system. Our ranking technique is used as a guideline both for developer and tester as, it is obtained at the analysis stage, before the coding starts.

The rest of the paper is organized as follows. In Section 2, we propose our approach to compute the influence of a component within a system at the architectural level and prioritize the components accordingly. In Section 3, we apply our proposed approach on two case studies to identify the critical components within a system and validate our approach through sensitivity analysis. We present the review of the related work in Section 4 and conclude the paper in Section 5.

2. OUR APPROACH TO PRIORITIZE COMPONENTS

We first discuss the concept of *Operational Profile* in Section 2.1 and then, discuss SDG in Section 2.2. Once SDG is constructed, we compute the occurrence probability of an event within a scenario in Section 2.3. We propose an algorithm to compute the influence value of a component in Section 2.4 and the working of the algorithm is discussed in Section 2.5.

2.1. Operational Profile

An Operational Profile assigns probability values to various use cases based on probability of use of the high level functions (use cases) by different user types [2]. Suppose we have drawn a use case diagram consisting of *m* types of users and *n* number of use cases for a practical application system. Each user type has assigned a probability of using the application system. Let u_i is the probability assigned to i^{th} user type of accessing the system such that $\sum_{i=1}^{m} u_i = 1.$

We identify the use cases with the high-level functional requirements of the system. Let q_{ij} is the probability of requesting the functionality of j^{th} use case (j=1...n) by i^{th} type user (i=1...m) such that $\sum_{j=1}^{n} q_{ij} = 1.$

Then, the probability value of a use case *x*, p(x), denotes the likelihood of the use case being executed by an average user is as follow.

$P(x) = \sum_{j=1}^{m} u_j * q_{jx} \ldots$ **(1)**

We consider that the functionality of any system can be modeled through a set of scenarios derived from use cases. From each use case, a number of scenarios are identified by drawing the sequence diagram for each possible use of use cases. Now, we will assign non uniform probability distribution to each scenario based on its frequency of execution. As per the domain knowledge, each scenario of a use case is assigned some frequency value based on the number of executions in a particular environment for a particular time period. Let $f_i(j)$ is the frequency of j^{th} scenario of i^{th} use case such that $\sum_{j=1}^{nos_i} f_i(j) = 1$ where, nos_i is the total number of scenarios of i^{th} use case. Then, the probability of execution of k^{th} scenario of i^{th} use case is:

$P(k_i)=p(i)* f_i(k)$…(2)

2.2 Sequence Diagram Graph (SDG)

We construct an intermediate graph of a sequence diagram called sequence diagram graph (SDG). SDG is a combination of all events of operation scenarios and all transitions among the events [8]. Mathematically, SDG=(S, E, s_0, f_0) where, S is the set of nodes representing various states of operation scenario; each node basically represents an event. E is the set of edges representing transitions from one state to another. s_0 is the initial node representing a state from which an operation begins. f_0 is the set of final nodes representing states where an operation terminates. An operation scenario is a quadruple, aOpnScn: (ScnId; StartState; MessageSet; NextState). A unique number called *ScnID* identifies each operation scenario. Here, StartState is a starting point of the ScnId, that is, where a scenario starts. MessageSet denotes the set of all events that occur in an operation scenario. NextState is the state that a system enters after the completion of a scenario. This is the end state of a use case. A SDG has a single start state and one or more end states depending on different operation scenarios. An event in a MessageSet is denoted by a tuple, aEvent: (messageName; fromObject; toObject [/guard]) where, messageName is the name of the message with its signature, fromObject is the sender of the message and toObject is the receiver of the message and the optional part /guard is the guard condition subject to which the aEvent will take place. An aEvent with * is an indication of an iterative event. A message consists of interaction between two objects with or without a guard condition.

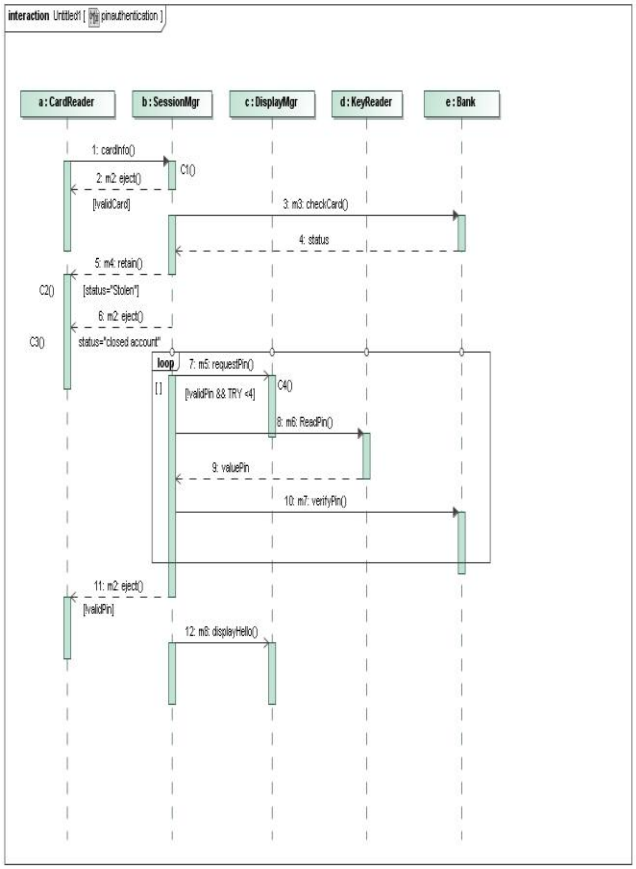
The sequence diagram associated with the use case *PIN Authentication* in a usual ATM system along with its five scenarios and the SDG of the use case is shown in Figure 1. We have extended the SDG proposed in [8], by adding the data dependency edges. We have considered the edge set E= (E1, E2) where, E1 represents control dependency and E2 represents data dependency among events within a use case. If a method is returning a wrong value, a number of events are affected by this due to using it in guard condition or in method call. If the activation of an event *A* depends on another event *B* then, *A* is control dependent on *B*. If an event *B* is defining a data and event *A* is using the data then, *A* is data dependent on *B*. In Figure 1.c, event S2 and S3 are control dependent on S1 and event S8, consisting of message verifyPin(), is data dependent on event S7, consisting of message readPin(). If method

readPin() of class KeyReader will return wrong value then, method verifyPin() of class Bank will be affected as, it is using that output at the time of its implementation. Control dependency is showing that the chance of activation of an event is depending on the output of other event whereas, data dependency is showing the chance of producing correct output by an event is depending on other events. Each event of a program belongs to one or more scenarios. That is, each scenario is typically implemented by several events and an event may participate in several scenarios. A scenario is a path in a SDG from start node to any end node through the control edges. $F_i(j)$ is a set of events within SDG_i which are activated when, scenario $SC_i(j)$ (j^{th} scenario of i^{th} use case) gets executed by the user. Once, the SDG of a use case is constructed, we calculate the probabilities of various nodes (events) of SDG based on probabilities of various scenarios of the use case within a system.

2.3 Probability Assignment for Events

Each event used within a use case is assigned probability value based on its frequency of execution within the use case. $pv(S_i(j))$, the probability of j^{th} event within SDG_i is as calculated as follow.

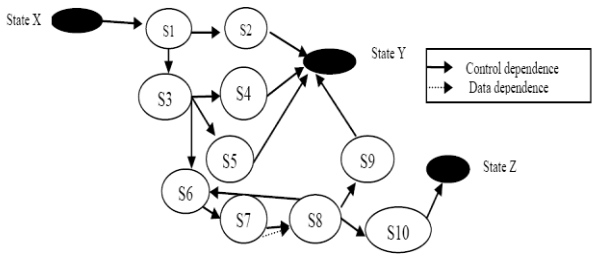
$$pv(S_i(j)) = \sum_{j=1}^{nos_i} q(j_i) \quad \ldots \quad \textbf{(3)}$$



(a) **Sequence diagram for *Pin Authentication* use case.**

<scn1	<scn2	<scn3	<scn4	<scn5
State X	State X	State X	State X	State X
s1: (m_1, a, b)	s1: (m_1, a, b)	s1: (m_1, a, b)	s1: (m_1, a, b)	s1: (m_1, a, b)
s2: (m_1, b, a)| c1	s3: (m_3, b, e)	s3: (m_3, b, e)	s3: (m_3, b, e)	s3: (m_3, b, e)
State Y>	s4: (m_4, b, a)| c2	s5: (m_2, b, a)| c3	s6: (m_5, b, c)| c4*	s6: (m_5, b, c)| c4*
	State Y>	State Y>	s7: (m_6, b, d)| c4*	s7: (m_6, b, d)| c4*
			s8: (m_7, b, e)| c4*	s8: (m_7, b, e)| c4*
			s9: (m_2, b, a)| c5	s10: (m_8, b, c)
			State Y>	State Z>

(b) **Various Scenarios of the use case**



(c) **SDG of the use case**

Figure 1: ***Pin Authentication* use case with its scenarios and SDG.**

In the above equation, $q(j_i) = p(j_i)$, if $S_i(j) \in F_i(j)$ else $q(j_i) = 0$. $F_i(j)$ is the set of events corresponding to scenario $SC_i(j)$ (j^{th} scenario of i^{th} use case) and nos_i is the number of scenarios within i^{th} use case. As discussed earlier, $p(j_i)$ represents the probability of j^{th} scenario of i^{th} use case within the system. Equation 3 expresses the fact that the probability value assigned to an event (node of SDG) within a use case is the summation of the probability values of all those scenarios of the use case to which the event belongs. The probability value of an event in a use case intuitively indicates the probability of it being executed during an actual operation of a use case. $pv(S_i(j)) = 0.8$ represents that the j^{th} event, $S_i(j)$, would get executed 80% of the time when, a user executes the i^{th} use case.

We present an algorithm called *Event Probability Computation Algorithm (EPCA)* in Figure 2 to compute the probabilities of various events within a use case. The variables used in our algorithm are as follows.

nos and *noe* represents the number of scenarios and number of events within the use case respectively.
$pv(S(k))$ is the probability value of the k^{th} event within a use case.
$F(j)$ contains the set of events that are activated at the execution of j^{th} scenario of the use case.

Algorithm 1 Algorithm: EPCA

Require: SDG of a use case and probabilities of various scenarios of the use case within the system.
Return Probabilities of various events within the use case.
1: **for** k ← 1 to noe **do**
2: pv(S(k)) ← 0.
3: **end for**
4: **for** j ← 1 to nos **do**
5: **for** each event S(k) ∈ {F(j)} **do**
6: pv(S(k)) ← pv(S(k)) + p(j).
7: **end for**
8: **end for**
9: **return** probability values computed for various events (nodes) of the use case (SDG).

Figure 2: **Event probability computation Algorithm.**

2.4 Computing the Influence of a component

If a method of a class returns a wrong value then, several events might be affected. It is because the wrongly computed returned value may be used in guard conditions of various events. Our aim is to get the failure-proneness of a component within a system at the architectural level and allocate test efforts accordingly. For this, we compute the influence value of a component through the data collected at the analysis and design stage. The influence value of a component *c* within a scenario is decided by checking how many components is using directly or indirectly the results computed by *c* within the scenario. For this, we maintain a set of events affected by a component within a scenario. We consider various use cases within a system and compute the influence value of a component for the overall system. For this, we propose an algorithm named *Influence Computation Algorithm* **(ICA),** shown in Figure 3.
The variables used in our algorithm, ICA, are as follows.
Inf_val(C_k): The influence value of k^{th} component, C_k, within a system.
Inf_set(C_k): The influence set of the component C_k. This contains the set of nodes, which are directly or indirectly dependent on the component within the use case.
n and *noc* represent the number of use cases and components within a system respectively.
noe_i and $Slice_i(j)$ represent the number of nodes (events) within SDG_i and slice of SDG_i with respect to j^{th} node respectively.
$pv(S_i(k))$ is the probability value of the k^{th} event (node) within i^{th} use case (SDG_i).

Algorithm 2 Influence Computation Algorithm (ICA)

Require: Set of SDG within a system and the probability value of each event within a SDG. {One use case has one SDG}
Return Influence values of various components interacting within the system.
1: **for** i ← 1 to noc **do**
2: inf_set(C_i) ← φ.
3: inf_val(C_i) ← 0.
4: **end for**
5: **for** i ← 1 to n **do**
6: **for** j ← 1 to noe_i **do**
7: $Slice_i(j)$ ← Slice($S_i(j)$, SDG_i).
8: Inf_set(toObject) ← Inf_set(toObject) ∪ $Slice_i(j)$
{The receiver component of the event $S_i(j)$ is named as *toObject*}
9: **end for**
{Compute the influence value of a component from the events in its influence set}
10: **for** k ← 1 to noc **do**
11: **if** Inf_set(C_k)≠ φ **then**
12: $$inf \leftarrow \sum_{S_i(k)\in T} pv(S_i(k)) \Big/ \sum_{S_i(k)\in SDG_i} pv(S_i(k))$$
Where, T = Inf_set(C_j).
13: inf_val(C_j)+ = inf
14: inf_ set(C_j) ← φ.
15: **end if**
16: **end for**
17: **end for**
18: **for** i ← 1 to noc **do**
19: **return** Inf_set(C_i)
20: **endfor**

Figure 3: **Influence computation Algorithm.**

2.5 Working of ICA

Consider the use case, *Pin Authentication*, within the ATM. The execution probabilities assigned to five scenarios (see Figure 1) based on Operational Profile are 0.05, 0.1, 0.1, 0.3, and 0.45. Based on our proposed algorithm 1 (see Figure 2), we assign probability to each node (event) in the SDG of the use case (see Figure 1.c). The probability value of event S1 is 1.0 as it is used in all the scenarios of the use case *Pin Authentication*. The probability value of S2...S10 is 0.05, 0.95, 0.1, 0.1, 0.75, 0.75, 0.75, 0.3 and 0.45. Then, we apply our proposed algorithm, ICA, (See Figure 3) to get the influence value of each component within the system.
Step-1: After forward slicing of node S1, inf_set(SessionMgr) = *{*S1, S2, S3, S4, S5, S6, S7, S8, S9, S10*}*. It means these events are dependent (control or data) on the return value of methods of class SessionMgr.
Step-2: After forward slicing of node S2, Inf_set(CardReader) = *{*S2*}*.
Step-3: After forward slicing of node S3, Inf_set(Bank) = *{*S3, S4, S5, S6, S7, S8, S9, S10*}*.
Step-4: After forward slicing of node S4, Inf_set(CardReader) = inf-set(CardReader) and S4} = {S2, S4}.
Step-5: After forward slicing of node S5, Inf_set(CardReader) = *{*S2, S4, S5*}*.
Step-6: After forward slicing of node S6, inf_set(DisplayMgr) = *{*S6, S7, S8, S9, S10*}*.
Step-7: After forward slicing of node S7, Inf_set(KeyReader) = *{*S6, S7, S8, S9, S10*}*.
Step-8: After forward slicing of node S8, Inf_set(Bank) = *{*S3, S4, S5, S6, S7, S8, S9, S10*}*.

Step-9: After forward slicing of node S9, Inf_set(CardReader) = *{*S2, S4, S5, S9*}*.
Step-10: After forward slicing of node S10, Inf_set(DisplayMgr) = *{*S6,S7,S8,S9,S10*}*.

Statement number 13 and 14 of Algorithm 2 (see Figure 3) computes the influence value of a component from the set of events in its influence set. The influence value computed for component *Bank* = (pv(S3) + .. + pv(S10)) / (pv(S1) + .. + pv(S10)) *100 = 80%. Similarly the influence values of SessionMgr, CardReader, KeyReader and DisplayMgr are 100%, 12%, 56% and 56% respectively. From our proposed method, it is found that the descending order of the components based on their influence value are SessionMgr, Bank, DisplayMgr, KeyReader and CardReader within the use case *PIN Authentication* in ATM system.

3. SENSITIVITY ANALYSIS

Using our estimated *influence value* for various components at the architectural level, we investigate the criticality of a component within a use case through the following procedure.
Procedure:-
Step 1. We select a component within a use case.
Step 2. We decrease the reliability[2] of the component from 1 to 0.5 in a step-wise manner, while fixing the reliabilities of other components to 1.0, for the sake of comparison.
Step 3. We check the variation in the reliability of a use case within a system by decreasing the reliabilities of its components (one at a time).
For sensitivity analysis, we have considered two case studies- Library Management System (LMS) and Automatic Teller Machine (ATM). LMS is already explained in our previous work [5]. The reader can refer [11] for a detailed view of ATM. The sensitivity analysis for various use cases of ATM case study is shown in Figure 4 and the sensitivity analysis of LMS case study is shown in Figure 5.

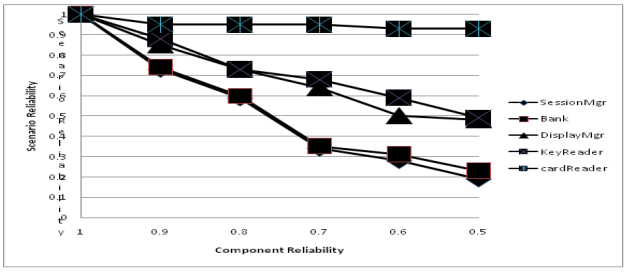
Figure 4.a shows that the variation in reliability of the use case *Pin Authentication* of ATM as a function of variation of its components reliabilities. From the figure, it is observed that the reliability of the use case is drastically decreased, when the reliability of components *SessionMgr* or component *Bank* is decreased whereas, this is not true for other components of the scenario. When the reliability of SessionMgr (Bank) is decreased by half, the reliability of the overall scenario is decreased to 0.19 (0.23). From the figure, we observe that when the reliability decreases for a component with high influence value, the reliability of the use case decreases in a high rate. As the influence values of components- *SessionMgr* and *Bank*- are high, the decrease in their reliability (one at a time) affects the use case more, whereas this is not true for the component *CardReader*, as it has very less influence value. Similarly, we conduct a sensitivity analysis and generate a list of components that are ranked according to their influence value within Withdraw and Deposit scenario of ATM case study. Table 1 and Table 2 show the influence values of various components within *Deposit* and *Withdraw* use cases respectively. Figure 4.b and Figure 4.c show the sensitivity analysis of the use cases *Deposit* and *Withdraw*. Similarly, we conduct the sensitivity analysis of *Issue-book* and *Delete-item* use case of LMS in Figure 5.a and 5.b. In LMS, the influence values of various components within Issue-book and Delete-item use cases are shown in Table 3 and Table 4. From the figures of both the case studies (see Figure 4 & 5), it is observed that components with high influence values within a use case have high impact on the reliability of the use case.

Table 1: ***Influence values*** **of various components within *Deposit* use case.**

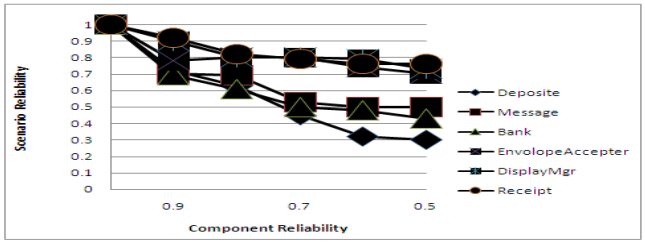
SessionMgr	Deposit	Message	Bank	EnvelopAccepter	DisplayMgr	Receipt
100%	92%	67%	92%	56%	49%	38%

Table 2: ***Influence values*** **of various components within *Withdraw* use case.**

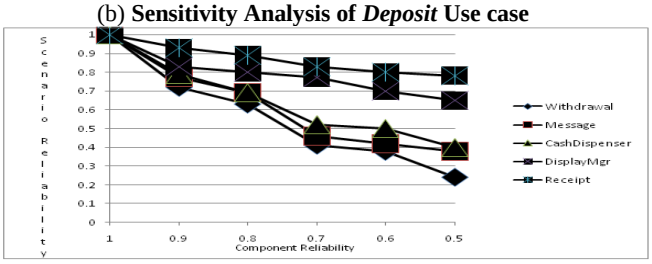
SessionMgr	Withdraw	Message	CashDispenser	DisplayMgr	Receipt
100%	92%	67%	86%	37%	28%



(a) **Sensitivity Analysis of *Pin Authentication* Use case**



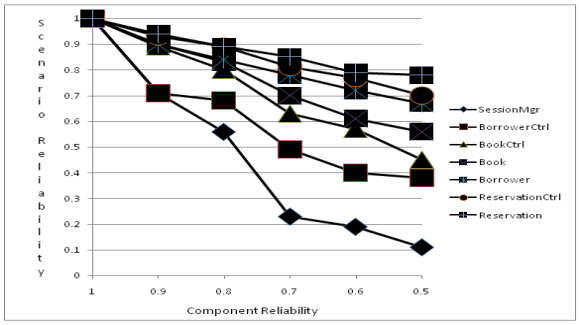
(b) **Sensitivity Analysis of *Deposit* Use case**



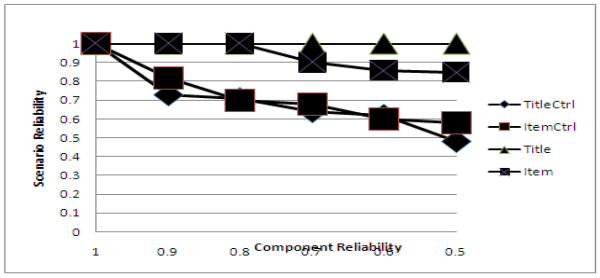
(c) **Sensitivity Analysis of *Withdraw* Use case**

Figure 4**: Use case reliability as a function of components reliabilities (one at a time) within various scenarios of ATM.**

[2] Techniques for class reliability estimation are a step wise procedure that includes fault injection, testing and retrospective analysis. We are assuming an estimate is available; this is used as a parameter for observing the failure rate to analyze the sensitivity of the application.



(a) **Sensitivity Analysis of *Issue-book* Use case**



(b) **Sensitivity Analysis of *Delete-item* Use case**

Figure 5: **Use case reliability as a function of components reliabilities (one at a time) within various scenarios of LMS.**

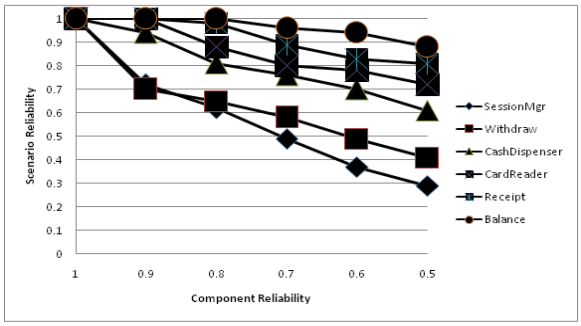
Table 3: ***Influence values* of various components within *Issue-book* use case.**

SessionMgr	BorrowerCtrl	BookCtrl	Book	Borrower	ReservationCtrl	Reservation
100%	76%	71%	58%	52%	39%	28%

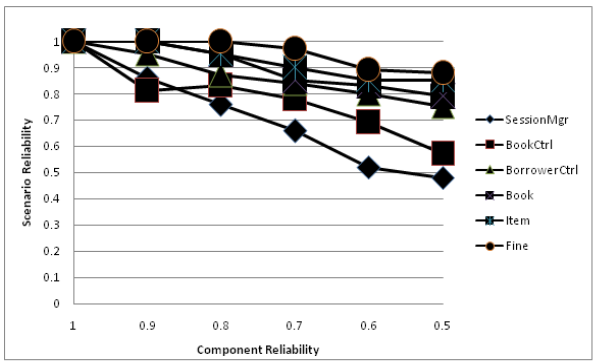
Table 4: ***Influence values* of components within *Delete-item* use case.**

TitleCtrl	ItemCtrl	Title	Item
100%	68%	37%	48%

Finally, we investigate the failure rate of the whole system based on the failure of its individual components. We have done it in three phases. In the first phase, we selected the component having the highest influence value from a case study and decreased its reliability, while fixing the reliabilities of others to 1.0, for the sake of comparison. To observe the failure rate of the application, we selected randomly *100* numbers of test cases (randomly selected scenarios) based on Operational Profile. A test case is responsible for the execution of one scenario at the system level. We continued our process by slowly decreasing the reliability of the selected component in a step wise manner and observed the failure rate of the system under test at each reliability point of that component for the same set of test cases. Same process and same test cases were also applied to a component having medium influence value and also to the component having the lowest influence value within the system.
As failures are observed by executing the same set of test cases at each reliability point of selected components (one at a time), it helps us to analyze the sensitivity of a component towards the system failures. The graphs in Figure 6 show the failure rates of the whole system (LMS and TAS case studies) by decreasing the reliability of the highest priority component, some medium priority components and the lowest priority component (one at a time) within a system. We have considered six components of each case study including the components having the highest and lowest influence values. In ATM and LMS, the components having the lowest influence values are *Balance* and *Fine* respectively and their influence towards failure is very much low compared to other components. In ATM, when the reliability of the component *Balance* is decreased to 0.5, the system reliability is 0.87 whereas, the system reliability is 0.28, when the reliability of the component *Withdraw* is decreased to 0.5. From Figure 6, it is observed that the system reliability is slowly affected by the decrease in the reliability of components with less influence value.



(a) **Sensitivity Analysis of *ATM***



(b) **Sensitivity Analysis of LMS**

Figure 6: **System reliability as a function of components reliabilities (one at a time)**

4. LITERATURE STUDY

A lot of research is going on to provide high reliable products to the customer within the available test resources. A number of testing approaches are proposed in the area of reliability analysis. These approaches are broadly classified into two types. The aim of the first one is to minimize the residual bugs in the software. For this, various methods are proposed to identify the faulty components within a system. It is found that many times a system having a number of faulty components can continue to provide reliable service as long as the service requested is not influenced by the adverse effects of the defective parts. Since the sequence of code executed in a particular run is dependent on the input data, an error in the non-executed statements or

branches does not have any effect on the output of the program. Hence, the system reliability depends on the probability that a bug is activated in the run. Based on this idea, the aim of second type of technique for reliability improvement is to identify the components which have a high impact on the reliability of the system. For this, a number of researchers have estimated reliability at the early stage and have conducted sensitivity analysis. In Section 4.1, we present a brief summary of work done relating to the identification of fault-prone components both at implementation level and architectural level. In Section 4.2, we discuss the work related to reliability estimation and sensitivity analysis at the architectural level for guiding the tester and the developer.

4.1. Identifying Fault-prone Components

Our work is related to the existing work on computation of fault-proneness of a component. The existing work in this area identifies the faulty components in a system and test effort prioritization is done accordingly. It estimates the probability of the presence of faults in a component, which helps to take valuable decisions on testing. A lot of research work has been done to identify the faulty components in a system [12-15], which are very relevant to our work. Different authors have focused on different characteristics associated with a component for counting faults. Eaddy et. al. [12] have experimentally proved that concern oriented metrics[3] are more appropriate predictors of software quality than structural complexity measures and there is a strong relationship between scattering and defects. Ostrand et.al. [13] have proposed a novel approach to identify the faulty files in the next release. For prioritizing test efforts, their approach considers the factors that are obtained from the modification requests and the version control system. These factors are (i) the file size (ii) whether the file was new to the system (iii) fault status in previous release (whether the file contained faults in earlier releases, and if so, how many) (iv) number of changes made (v) programming language used for implementation. For some initial releases, the models were customized based on the above observed factors. Based on the experimental results, the authors conclude that their methodology can be implemented in the real world without extensive statistical expertise or modeling effort. Ostrand et. al [15] have proposed a negative binomial regression model, in which the binomial model is used to predict the expected number of faults in each file of the next release of a system. The predictions are based on the code of the file in the current release, and fault and modification history of the file from previous releases. Emam et. al. [14] found that a class having high export coupling value is more fault-prone. A complex program might contain more faults compared to a simple program [16]. As the factor complexity is the most important defect generator, researchers [17, 18] have used the *complexity metric* as a parameter for testing

Some researchers have proposed prediction of faulty components from design metric at the architectural level. Researchers [19, 20] relate the structural complexity metric (CK metric suite [21]) to fault-proneness. From the discussed research work, it is observed that the estimated defect density (fault- proneness) that is computed through static analysis and the pre-release defect density that is computed through testing are strongly correlated. Unlike these papers, our aim is not to investigate the characteristics of a component to check which components have high fault densities. Our aim is to make the testing process more effective by finding defects from critical components within the system, so that the reliability of a system can be improved without increasing the testing budget.

[3] A concern is anything a stake holder may want to consider as a conceptual unit, including features, nonfunctional requirements and design idioms.

4.2. Early Reliability Estimation and Sensitivity Analysis at the architectural level

At the early stage of development life cycle, a lot of alternative reusable assets are available. So, to get a more reliable system within the testing budget, it is required to study the sensitivity of the application reliability to reliabilities of its elements within the system. It helps the system architect to select elements with suitable reliability characteristics. Cortellessa et.al. [22] have proposed an early estimation of time distribution for components from UML model. Their approach on system reliability prediction is based on component and connector failure rates. Three different types of UML diagrams: Use Case, Sequence and Deployment diagrams are used for reliability analysis. For estimating the time spent in each component, they have counted the number of times a class is busy in a scenario. Both component failure and connector failure probabilities are considered. According to their approach, a component which is busy more times is more failure-prone. Similar to this, Yacoub et. al. [7] also proposed a path-based approach to get the early reliability of a system at the analysis phase. They proposed an algorithm named Scenario-Based Reliability Analysis (SBRA). SBRA is used to identify critical components and critical component interfaces, and to investigate the sensitivity of the application reliability to changes in the reliabilities of components and their interfaces. The technique is suitable for systems whose analysis is based on valid scenarios with timed sequence diagrams.

Garousi et al. [23] have proposed a Behavioral Dependency Analysis (BDA) technique for UML model to measure the dependency between two entities in a system. They have calculated dependency index between two entities based on criticality of a message, amount of data carried by a message, frequency of use of return value and operational profile. Though BDA technique is not estimating any reliability of a system, but like our approach, it helps the analysts to devise appropriate provisions for the most crucial entity of a system and forecast test effort for each entity before implementation to improve the reliability of a system.

5. CONCLUSION AND FUTURE WORK

We have proposed a technique to prioritize components at the architectural level in accordance to their impact on the reliability of the system. The priority of a class within a system is decided based on its influence towards system failures. The inputs considered for influence computation are sequence diagram and the Operational Profile designed for the system. First, we generate an intermediate graph of a sequence diagram and apply the forward slicing method on the graph to compute the influence of a component within a use case. Then, we use it to

get the influence of the component within the whole system. Using our priority estimation, we investigate the failure rate of a use case and also the failure rate of the whole application based on the failure of its individual components. We conducted a set of experiments and concluded that the system reliability is decreased at a higher rate when, the reliability of a high priority class is decreased, but this is not true for a low priority class. Hence, our proposed *Influence Metric* helps to improve the reliability of a system by exposing critical components within the system at the architectural level. The *Influence metric* is used as a guideline both for the tester and the developer throughout the development cycle.

We have only prioritized a component based on its influence towards system failures. We have not considered the impact of a failure within a system. We want to minimize the failures which have a negative impact on the user's perception on the system reliability or create a financial loss to the organization, when the system is executed for some duration in the operational environment. For this, in our future work, we are planning to consider two external factors: (i) the impact of a failure (the severity of a failure) within a use case and (ii) the business value (value that comes from customer and market) associated with a use case, for prioritization.

6. REFERENCES

[1] Sommerville, I.: Software Engineering. 5th Edn. Pearson (1995).

[2] Musa, J.D.: Operational profiles in Software-Reliability Engineering. IEEE Softw. 10(2) (1993), pp. 14-32.

[3] Cobb, R.H., Mills, H.D.: Engineering Software under Statistical Quality Control. IEEE Softw. 7(6) (1990), pp. 44-54.

[4] Musa, J.D.: Software Reliability Engineering: More Reliable Software Faster and Cheaper. AuthorHouse (2004).

[5] Ray, M., Kumawat, K., L., and Mohapatra, D., P.: Source code prioritization using forward slicing for exposing critical elements in a program. Journal of Computer Science and Technology 26(2) (2011), pp. 314-327.

[6] Ray, M., Mohapatra, D.P.: Reliability improvement based on prioritization of source code. In Janowski, T., Mohanty, H., eds.: ICDCIT. Volume 5966 of Lecture Notes in Computer Science. Springer (2010), pp. 212-223.

[7] Yacoub, S., M., Cukic, B., and Ammar, H., H.. Scenario-based reliability analysis of component-based software. *IEEE Transactions on Reliability*, 53(04) (2004), pp.465–480.

[8] Sarma, M. and Mall, R.. Automatic test case generation from UML models. In *10th International Conference on Information Technology (*2007), pp. 197–201.

[9] Srikant, Y.N., Shankar, P., eds.: The Compiler Design Handbook: Optimizations and Machine Code Generation. CRC Press (2002).

[10] Ray, M. and Mohapatra, D., P.: A scheme to prioritize classes at the early stage for improving observable reliability. In Proceedings of the 3rd India software engineering conference, ACM, New York, NY, USA, ISEC (2010), pp. 69-72.

[11] ATM case study available: http://www.mathcs.gordon.edu/courses/cs211/ATMExample/.

[12] Eaddy, M., Zimmermann, T., Sherwood, K.D., Garg, V., Murphy, G.C., Nagappan, N., Aho, A.V.: Do crosscutting concerns cause defects? IEEE Transactions on Software Engineering **34** (2008), pp. 497–515.

[13] Ostrand, T.J., Weyuker, E.J., Bell, R.M.: Automating algorithms for the identification of fault-prone files. In: proceedings of the 2007 international symposium on Software testing and analysis. (2007), pp-219–227.

[14] Emam, K., Melo, W., Machado, C., J.: The prediction of faulty classes using object-oriented design metrics. Journal of Systems and Software **56**(1) (2001), pp. 63–75.

[15] Ostrand, T.J., Weyuker, E.J., Bell, R.M.: Predicting the location and number of faults in large software systems. IEEE Transactions on Software Engineering **31** (2005), pp. 340–355.

[16] Munson, J.C., Khoshgoftaar, T.M.: The detection of fault-prone programs. IEEE Trans. Softw. Eng. **18**(5) (1992), pp. 423–433.

[17] Subramanyam, R., Krishnan, M.S.: Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. IEEE Trans. Softw. Eng. **29**(4) (2003), pp-297–310.

[18] Lyu, M.R.: Software reliability engineering: A roadmap. In: FOSE '07: 2007 Future of Software Engineering (2007), pp.153–170.

[19] Briand, L.C., W¨ust, J., Daly, J.W., Porter, D.V.: Exploring the relationship between design measures and software quality in object-oriented systems. J. Syst. Softw. **51**(3) (2000), pp. 245–273.

[20] Subramanyam, R., Krishnan, M.: Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. IEEE Transactions on Software Engineering **29** (2003), pp. 297–310.

[21] Chidamber, S.R., Kemerer, C.F.: A metrics suite for object-oriented design. IEEE Trans. Softw. Eng. **20**(6) (1994), pp. 476–493.

[22] Cortellessa, V., Singh, H., and Cukic, B.. Early reliability assessment of UML based software models. In *WOSP*'02: Proceedings of the 3rd international workshop on Software and performance (2002), pp. 302–309.

[23] Garousi, V., Briand, L., C., and Labiche, Y.. Analysis and visualization of behavioral dependencies among distributed objects based on UML models. Technical Report TR SCE-06-03, Carleton University, Ottawa, Canada, (2006).