Effective Regression Testing Using Requirements and Risks

Charitha Hettiarachchi
North Dakota State University
charitha.hettiarachc@ndsu.edu

Hyunsook Do
North Dakota State University
Ewha Womans University
hyunsook.do@ndsu.edu

Byoungju Choi
Ewha Womans University
bjchoi@ewha.ac.kr

***Abstract*—The use of system requirements and their risks enables software testers to identify more important test cases that can reveal faults associated with risky components. Having identified those test cases, software testers can manage the testing schedule more effectively by running such test cases earlier so that they can fix faults sooner. Some work on this area has been done, but the previous approaches and studies have some limitations, such as an improper use of requirements risks in prioritization and an inadequate evaluation method. To address the limitations, we implemented a new requirements risk-based prioritization technique and evaluated it considering whether the proposed approach can detect faults earlier overall. It can also detect faults associated with risky components earlier. Our results indicate that the proposed approach is effective for detecting faults early and even better for finding faults associated with risky components of the system earlier than the existing techniques.**

***Keywords*-Regression testing, requirements risks-based testing, test case prioritization, empirical study**

I. Introduction

Regression testing and maintenance are important activities to ensure high quality for modified software systems. Typically, these activities require a great deal of time and effort, so it can be a big burden for software companies that often have a time pressure with product release. One way to help this situation is to apply test case prioritization that identifies more important test cases (e.g., test cases that detect more faults) and runs them early within the limited time block. With this approach, companies can increase the chances to detect and fix faults early.

Due to their appealing benefits in practice, various test case prioritization techniques have been proposed and studied by researchers and practitioners [1], [2], [3], and many empirical studies have shown the effectiveness of test case prioritization [4], [5]. While the majority of test case prioritization approaches utilize source code information, some researchers have investigated using other software artifacts produced during early development phases, such as system requirements and design documents [6], [7], [8]. For example, Srikanth et al. [8] present an approach that prioritizes test cases using system requirements as well as their importance and fault proneness, and Arafeen and Do [6] cluster test cases using requirements similarities and prioritize them by incorporating code information. Krishnamoorthi and Mary [7] present a technique that prioritizes test cases using requirements and several factors related the requirements, such as implementation complexity and customer priority.

This trend is encouraging because such artifacts could provide a better understanding about the source of errors. Using system requirements and their risk information, software testers can manage the testing schedule better by identifying more important test cases that are likely to detect defects associated with the risks faced by the system (e.g., safety or security risks). To gain such benefits, a few researchers have started investigating the use of risks with the requirements, and these studies found that using risks along with requirements could improve the effectiveness of test case prioritization [9], [10], [11].

These approaches, however, consider one limited risk type (e.g., fault information collected from the previous version) [9], [10] or do not consider a direct relationship between requirements risks and test cases when they prioritize test cases [11]. Typically, software systems contain various types of risk, thus considering only one limited risk type fails to properly expose important potential risks in the software system. Also, without utilizing a direct relationship between requirements risks and test cases, prioritization techniques could fail to produce an effect order of test cases that can early expose faults in risk areas. Further, like other test case prioritization research, these studies have evaluated the proposed approaches by measuring how fast the reordered test cases detect faults. However, the approaches utilize risk information to prioritize test cases, so they should be evaluated by measuring whether the detected faults are, indeed, from the locations where risks reside in the product.

To address these limitations, we propose a new test case prioritization technique that uses risk levels of potential defect types to identify risky requirements and that prioritizes test cases based on the relationship between test cases and these requirements. To evaluate our approach, we define a new evaluation method that shows how fast the reordered test cases can detect faults in the risky areas. To investigate the effectiveness of our approach, we have designed and performed an empirical study using an open source program written in Java with multiple versions and requirements documents. Our results shows that the technique is effective in finding faults early and even better in finding faults in the risky components earlier than the existing techniques.

Section II describes our new prioritization technique in detail. Sections III and IV present our experiment, including design, results, and analysis. Section V discusses our results and their implications. Section VI describes the related work relevant to test case prioritization techniques. Finally, Section VII presents conclusions and discusses possible future work.

II. Prioritization Approach

In this section, we describe the prioritization approach that uses system requirements risks, which has five main steps:

1) Evaluate risks by correlating with the requirements
2) Calculate the risk weights for requirements
3) Calculate the risk exposure values
4) Evaluate other factors to prioritize test cases
5) Prioritize requirements and test cases

Figure 1 gives an overview of the proposed technique. The main steps of the approach are shown in light blue boxes while the inputs and outputs for each step are shown in the ovals. The first four steps are used for calculating requirements priorities, and the last step is used for prioritizing test cases based on the results produced by the first four steps. Note that "Req-Test Case Mapping" activity in the figure is not listed as a part of steps, but this activity is required for the last step ("Prioritize requirements and test case") to provide the relationship between requirements and test cases. The following subsections describe the rest of the steps in detail.

A. Evaluate Risks by Correlating with Requirements

Risks related to each requirement of the system were determined by a graduate student who has more than four years of software industry experience. The graduate student surveyed the risks related to software product requirements found in the literature [12], [13]. Several risk factors were considered in determining requirement risks, such as requirement complexity, potential security threats residing in the requirements (i.e., confidentiality, integrity, availability, and privacy related threats), risks in terms of system quality, and inconsistencies. Requirement risks can vary from system to system, including system's operational environments. For instance, for our experimental application (*iTrust*), some examples of the risks are as follows: coding difficulties caused by the interdependencies among software components (some requirements have high complexity,) a security risk due to multiple transactions required to satisfy a particular requirement, and mishandling user requests due to many user classes with different privilege levels.

As we can see from the risks listed above, not all software risks have the same impact on the software product. Some risks can be acceptable, and some can be catastrophic. Further, their probabilities of occurrence can vary. Thus, for each risk and requirement, we assign the probability of a risk's occurrence associated with a requirement (risk likelihood) and the degree of the possible damage (risk impact) caused by a risk for a requirement. (The example is given in the next section.) These values are used in the next step to calculate the risk weights for the requirements to quantify the risk level of each requirement.

B. Calculate the Risk Weights for Requirements

Using two risk weight factors that we obtained from the previous step, risk likelihood (RL) and risk impact (RI), we calculate risk weight (RW) by following Amland's risk model [14].

The RW values are obtained using the following equation:

$$RW_{(Req_j)} = \sum_{i=1}^{N} (W_i * R_{ji}) \quad (1)$$

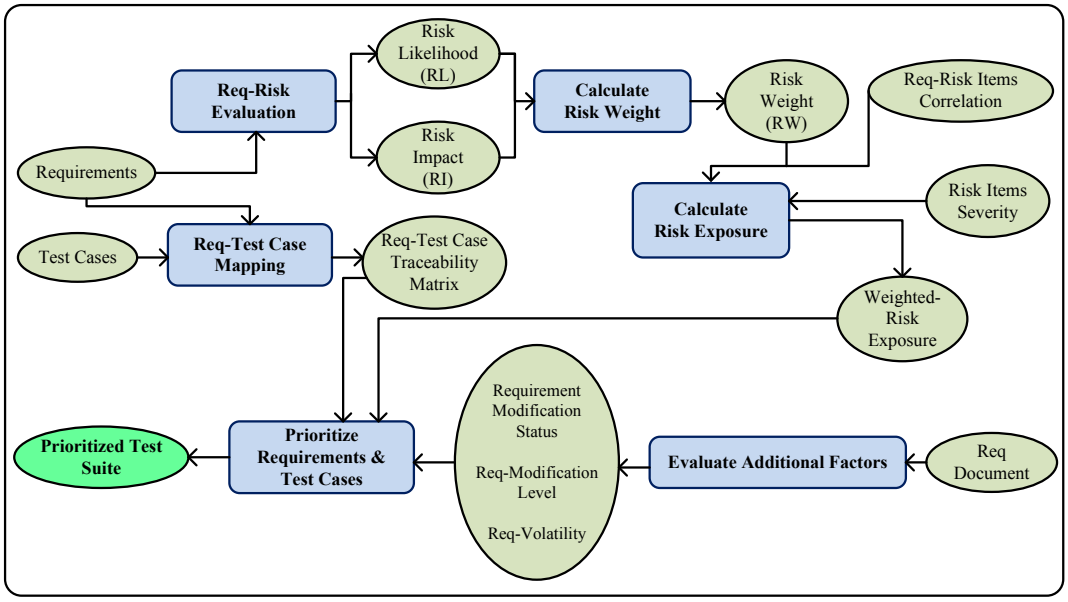

Fig. 1. Overview of Requirements Risk-based Approach

TABLE I
RISK WEIGHTS OF REQUIREMENTS

Requirements	RL	RI	RW
	0.4	0.6	
UC1S1	21	23	22.2
UC2S1	35	45	38
UC3S4	40	47	44.2
UC6S1	24	30	27.6
UC6S2	32	34	33.2
UC8S1	20	22	21.3
UC9S1	30	32	31.2
:	:	:	:
UC34S6	25	26	25.6

where N is the number of criteria; W_i is the relative weight of the criterion, C_i (In this experiment, C_i denotes RL or RI); and R_{ji} is the performance value of the requirement, Req_j, under criterion C_i.

Table I shows a portion of RW values for *iTrust* requirements (version 1). As shown in Table I, we assigned the weights for RL and RI as 0.4 and 0.6, respectively. When determining requirements risks in terms of RL and RI, RI becomes a key factor to decide the risk level. The software system may cope with risks that have a high probability and low impact, but risks with high impact can cause a serious problem if they occur. Thus, to draw more attention to requirements with a relatively high risk impact and a relatively low likelihood compared to requirements with a relatively high probability and a low risk impact, we assigned a higher ratio to RI. Both RL and RI values ranged from 0 to 50, and have been commonly used by others [15]. A value of 0 indicates no likelihood of risk or impact, whereas 50 indicates the highest likelihood of risk or impact.

In this example, for the UC1S1 requirement ("The health care personnel creates a patient as a new user of the system."), we assigned 21 as the RL because this requirement has an average probability of being a potential risk for the *iTrust* system. This RL was quantified by considering the system's current security measures, implementation simplicity, and language robustness. For the RI of this requirement, we assigned 23 because the requirement has potential risks related to the new user's confidentiality and privacy, and it might also cause inconsistencies in the system, unless the relevant data (e.g., patient's ID) are properly handled in the system. Thus, the RW value for UC1S1 was obtained by applying equation 1 as follows:

$$\text{RW}_{(UC1S1)} = (0.4*21) + (0.6*23) = 22.2$$

Because both RL and RI values range from 0 to 50, RW values have the same range. A value of 0 means no risk, and 50 means the highest risk. The RW for UC1S1 is 22.2, which indicates a medium-low risk. In Table I, UC3S4 has the highest RW (44.2), and UC8S1 has the lowest RW (21.3). In fact, the UC3S4 requirement allows a licensed health care professional (LHCP) to access laboratory results of patients, to see upcoming appointments, and to reject or accept comprehensive reports. There is a relatively high risk of misusing a patient's confidential information, such as laboratory results, by LHCP or a hacker, so we assigned a high RL value. If the confidentiality of patients' sensitive data are breached or patients' privacy is violated, severe consequences such as monetary and reputation losses may occur. Thus, a high RI value was assigned.

C. Calculate Risk Exposure Values

So far, we explained how to calculate RW values for the requirements. Although the RW values of the requirements illustrate how risky each requirement is in terms of the risks identified for the requirements themselves, RW does not provide information about whether the requirements are associated with potential defect types of the system. (Note: One particular requirement can be associated with multiple defect types, which indicates that such a requirement has a high potential to expose several defects.) Using this information, we can identify the requirements that are related to common failures of a software product, thus we can assign high priorities, which will be used in prioritizing test cases to such requirements.

TABLE II
SOFTWARE PRODUCT RISK ITEMS

	Risk Item	Abbreviation
i	Input Problem	IP
ii	Output Problem	OP
iii	Calculation	Calc
iv	Interactions	Inac
v	Error Handling	ErHa
vi	Startup/ShutDown	St/Sh

To obtain such information, we followed the process defined by Yoon et al. [11]: (1) Identify risk items (RiIM) as shown in Table II. Risk items are potential defect types of a particular system [12]. The input problem is one example. During system operation, input data can cause several problems. For instance, a system many crash or perform erroneously if the input data are not validated before they are executed or if the data are beyond the valid boundary. (2) Calculate risk exposure values for risk items. Here, risk exposure (RE) values quantify the risk level for the risk item.

To find the probability of fault occurrence, we considered the association of risk items and requirements. When a particular risk item associated with more requirements, its probability of fault occurrence will increase. In order to determine the cost of risk items, the risk weight (RW) of requirements is employed. The cost of risk items increase when its associated requirements have high risk weights. Table III shows a matrix to determine the risk exposure (RE) values for risk items and the weighted RE values for requirements.

The matrix lists requirements, risk weights for requirements, and a set of risk items. Each risk item ($RiIM_x$) has a severity

TABLE III
RISK EXPOSURE AND WEIGHTED RISK EXPOSURE MATRIX

		Risk Items				
Requirements	Risk Weights	RiIM1 (SV1)	... ...	RiIMx (SVx)	RiIMn (SVn)	Weighted-RE
Req_1	RW(Req_1)	C_{11}		C_{1x}	C_{1n}	Weighted-RE(Req_1)
...	...	...	...	...	...	...
Req_y	RW(Req_y)	C_{y1}	...	C_{yx}	C_{yn}	Weighted-RE(Req_y)
Req_m	RW(Req_m)	C_{m1}	...	C_{mx}	C_{mn}	Weighted-RE(Req_m)
		RE(RiIM1)	...	RE(RiIMx)	RE(RiIMn)	

value (SV_x) that indicates how risky the item is. The severity values are defined in Table IV. If a risk item is associated with a certain requirement, the C_{mx} value is 1; otherwise, the value is 0. Again, multiple associations between risk times and requirements can exists. The last row in the matrix shows RE values for risk items, and RE values are calculated using Equation 2; the last column shows weighted RE values that are calculated by incorporating RE values with severity values of risk items for each requirement. The final outcome of this step, the weighted RE values, is used to prioritize requirements.

Equations 2 and 3 are used to calculate the risk exposure values for risk items and the weighted risk exposure values for requirements.

$$RE(RiIM_x) = \sum_{i=1}^{m} (RW(Req_y) * C_{yx}) \tag{2}$$

where m is the number of requirements, $RW(Req_y)$ is the risk weight of Req_y, and C_{yx} is 1 when requirement Req_y is associated with the risk item $RiIM_x$ or 0 otherwise.

$$Weighted - RE(Req_y) = \sum_{i=1}^{n} (RE(RiIMx) * C_{yx} * SV_x) \tag{3}$$

where n is the number of risk items for the system; SV_x is the severity value of the risk item, $RiIM_x$; and C_{yx} indicates the correlation between the requirement, Req_y, and the risk item, $RiIM_x$.

TABLE IV
SEVERITY OF RISK ITEMS

Severity Value	Description
1	Slightly critical risk item
2	Moderately critical risk item
3	Very critical risk item
4	Most critical risk item

Table V shows a sample data set collected from this experiment. In this example, we can see that the output problem's risk item is associated with all requirements except for UC6S2. Thus, the risk exposure value of the output problem's risk item can be calculated by using Equation 2 as follows:

RE(OP) = (22.2*1) + (22.2*1) + (38*1) + (38*1) +...+ (33.2*0) +...+ (25.6*1) = 3530.2

Because the output problem has a high RE value compared to other risk items, it implies that the output problem is a highly risky area for this product.

After calculating RE values, we calculate the weighted RE values for each requirement by following Equation 3. For instance, we obtain the weighted RE value for UC1S1 as follows.

Weighted-RE(UC1S1) = (3530.2*1*4) + (3206.2*1*1) + (3185.4*0*3) + (2516.6*1*4) + (881.6*0*3) + (503.4*0*2) = 27393.4

After calculating the weighted RE values for requirements, we prioritize requirements by their weighted RE in descending order. Table VI shows a portion of our data. From the table, we can see that each requirement has one or more corresponding test cases (the last subsection explains how to map these two) and requirements are appeared by their weighted RE in descending order. When multiple requirements have the same value (e.g., UC33S1 and UC26S2 have the same value, 40601.2, in the table), we need to decide which one should be picked first among others. To aid that process, we consider additional factors for prioritization, and the next section explains them.

D. Evaluate Additional Factors to Prioritize Requirements

We utilize the weighted RE values as a primary factor to prioritize requirements, but if multiple requirements have the same value, we need to have a way to break the tie. One simple way is to randomly choose a candidate and repeat the process until all remaining requirements with the same value are chosen. In this study, however, we consider three factors that can be potentially effective in finding error-prone requirements [6], [7], [16], [17]: requirement modification status, level of modification, and level of requirement volatility.

Requirement modification and its modification level were determined by comparing two consecutive versions. For new requirements, we assigned the highest level of modification because new requirements can cause serious source code modifications and can introduce new faults to the system [6]. Volatile requirements are susceptible to have faults in later versions of the system. However, the impact of volatile requirements in terms of introducing system faults is relatively low compared to requirement modifications. We used modification status as a second factor for prioritization, followed by the level of

TABLE V
EXAMPLE OF RISK EXPOSURE AND WEIGHTED RISK EXPOSURE OF ITRUST

REQS	TC ID	RW	OP	ErHa	Inac	IP	Calc	Sh/St	Weighted-RE
Risk Items Severity Values			4	1	3	4	3	2	
UC1S1	TC1	22.2	1	1	0	1	0	0	27393.4
UC1S1	TC2	22.2	1	1	0	1	0	0	27393.4
UC2S1	TC5	38	1	1	1	1	0	0	27393.4
UC2S1	TC6	38	1	1	1	1	0	0	36949.6
UC3S4	TC7	44.2	1	0	1	0	0	0	23677
UC3S4	TC8	44.2	1	0	1	0	0	0	23677
UC6S1	TC9	27.6	1	0	0	0	0	0	14120.8
UC6S1	TC10	27.6	1	0	0	0	0	0	14120.8
UC6S2	TC11	33.2	0	0	1	0	0	0	9556.2
UC6S2	TC12	33.2	0	0	1	0	0	0	9556.2
UC8S1	TC13	21.2	1	1	0	1	1	0	30038.2
UC9S1	TC14	31.2	1	1	1	0	0	0	26883.2
UC9S1	TC15	31.2	1	1	1	0	0	0	26883.2
:	:	:	:	:	:	:	:	:	:
UC34S6	TC122	25.6	1	1	1	1	0	0	36949.6
Risk Exposure (RE)			3530.2	3206.2	3185.4	2516.6	881.6	503.4	

TABLE VI
EXAMPLE OF PRIORITIZED TEST SUITE - ITRUST

Req	TC-ID	Weighted-RE	RM	RML	VL
UC33S1	TC117	40601.2	1	10	6
UC26S2	TC94	40601.2	0	0	6
UC21	TC30	39594.4	1	10	7
UC21	TC80	39594.4	1	10	7
:	:	:	:	:	:
UC12	TC26	26480.4	0	0	3
UC30S3	TC100	23835.6	1	10	6
UC30S1	TC99	23835.6	1	4	6
UC3S3	TC43	16414.0	0	0	3
UC24	TC91	15917.4	0	0	7
UC25	TC92	15917.4	0	0	6
:	:	:	:	:	:
UC2S2	TC77	10066.4	0	0	2

modification and the level of requirement volatility. If the requirements are still tied after applying all the factors, they are ordered randomly.

Table VI shows the values for these three factors applied to each requirements. RM, requirement modification status, has two values to indicate whether the requirement is modified (1) or not (0). RML indicates the level of modification. VL indicates the level of requirements volatility. These values ranges from 0 to 10. The value 0 indicates the lowest level and 10 indicates the highest level. Again, for the first two requirements, UC33S1 and UC26S2, in the table, because they have the same weighted RE value, their priority is decided by comparing the next factor, RM. UC33S1 has 1 and UC26S2 has 0 for RM, so UC33S1 is picked first for prioritization.

E. Prioritize Requirements and Test Cases

After collecting all the values we described, we prioritize requirements using the weighted RE values and the additional factors. Next, to obtain prioritized test cases, we need a traceability matrix that shows mapping relationships between requirements and test cases. In our case, we used the traceability matrix that came with *iTrust*. (The developers created requirements and tests mapping information).

Often, multiple test cases are created for a single requirement, meaning that the prioritization values for all tests with the same requirement are the same. Thus, after mapping test cases to their corresponding requirements, we randomly order tests with the same priority. For *iTrust*, only one or two test cases are associated with the same requirement.

III. EMPIRICAL STUDY

In this study, we investigate the following research questions:

RQ1: Can requirements risk-based test case prioritization improve the rate of fault detection for test suites?

RQ2: Can requirements risk-based test case prioritization find more faults in the risky components early?

A. Objects of Analysis

TABLE VII
EXPERIMENT OBJECTS AND ASSOCIATED DATA

Ver.	Size (KLOCs)	Use Cases	Req.	Test Cases	Mutation Faults	Mutation Groups
v1	24.42	28	91	122	54	13
v2	25.93	32	105	142	71	12
v3	26.70	34	108	157	75	12

An open source application *(iTrust)* was utilized for this experiment. The *iTrust* program is a patient-centric electronic health record system which was developed by the RealSearch Research Group at North Carolina State University. Four versions of the *iTrust* system (version 0, 1, 2, and 3) were used in our experiment. Each version has more than 28 user cases. The test cases used in this study were functional test cases associated with requirements and written by *iTrust* system developers. Each version's metrics are listed in Table VII. The

metrics for *v0*, which is the base version of *iTrust*, are not listed in the table because regression testing starts with the second release of the system. We, however, use information from *v0* to obtain mutants for *v1*. Our experiment requires faults in the program, so we utilize mutation faults created from our previous study [6].

B. Variables and Measures

Independent Variable: Our study manipulated one independent variable, test case prioritization technique. We considered four control techniques and one heuristic prioritization technique as follows:

- Control Techniques
  - Original (*Torig*): The object program provides the testing scripts. *Torig* executes test cases in the order in which they are available in the original testing script.
  - Code metric (*Tcm*): This technique uses a code metric that we defined in our previous study [6]. The code metric is calculated using three types of information obtained from source code, Line of Code (LOC), Nested Block Depth (NBD), and McCabe Cyclomatic Complexity (MCC), which are considered good predictors for finding error-prone modules [18], [19].[1]
  - Requirements-based clustering: We consider two requirements-based clustering techniques proposed by Arafeen and Do [6] as follows:
    - Tcl-orig-prior (*Tcop*): This technique uses the original test case order for prioritization and the prioritized cluster order for selection.
    - Tcl-cm-prior (*Tccp*): This technique uses the code metric for prioritization and the prioritized cluster order for selection.

    The previous study [6] used several cluster sizes, but in this study, to simplify the comparisons, we chose cluster sizes of 10 and 20 which showed moderate and the best results, respectively.
- Heuristic (*Trrb*): The heuristic technique uses requirements and their risks to prioritize test cases as described in Section II.

Dependent Variable and Measures: We considered two dependent variables as follows:

- Average Percentage of Fault Detection (APFD): APFD [20], [21] is the measurement of averaging the percentage of fault detection during the execution of a test suite. The APFD value ranges from 0 to 100, and the higher the value is, the better the technique is. (See [20] for the formal definition of APFD.)
- Percentage of Total Risk Severity Weight (PTRSW): PTRSW shows how effective the test suite is for finding more faults in the system's risky components as early as possible. The PTRSW value ranges from 0% to 100%. If we run all test cases, the PTRSW value reaches 100%. If we execute a portion of the test cases by stopping at a certain point, the PTRSW value is under 100% and the higher the value is, the better the technique is. The following equation shows how to calculate PTRSW:
  PTRSW = (TRSW/GTRSW)*100%

  The Total Risk Severity Weight (TRSW) and the Grand Total Risk Severity Weight (GTRSW) are explained in detail in the following section.

C. Experimental Setup and Procedure

The requirements risk-based approach requires several types of information, such as requirements-tests mapping information, requirements modification information, risk exposure (RE) and weighted risk exposure (W-RE) values, requirements-classes/methods association information, and fault-test association information.

The requirements-tests mapping information came with *iTrust*. To obtain requirements modification information, we manually inspected the requirements across all versions. Risks associated with the requirements were identified and documented as explained in Section II.

The risk exposure (RE) values for the requirements and the weighted risk exposure (W-RE) values for risk items were obtained using the steps explained in Section II. To obtain fault data for this experiment, a set of mutation faults created in the previous study [6] was used. We created mutant groups by randomly choosing n mutation faults (between 1 and 10) from those available with that particular version of the program. We repeated this process for each version, and we obtained 13, 12, and 12 mutant groups for v1, v2, and v3, respectively. Because we need to know the locations for the mutation faults to measure PTRSW, we used the ByteMe mutation analysis tool to identify the classes and methods altered by mutation faults.

To locate the requirements affected by the modification, using the mutation analysis tool, we built a requirements-classes/methods trace file that shows which classes and methods were used to implement a particular requirement. This information, along with the requirements risks, was used to determine risks for system classes. In this way, a risk caused by a particular mutation fault in a system component was determined, and the Risk Severity Weight (RSW) value was assigned to the mutation fault. The RSW value ranges between 0 and 10. RSW is 0 if a mutation fault does not cause any risk, and it becomes 10 if a mutation fault causes severe risks in a critical component.

A mutation fault is detected by one or more test cases. When multiple test cases detect the same mutation fault, all those test cases are assigned the same RSW value. Also, a single test case can detect multiple mutation faults. In that case, all RSW values for the associated mutation faults are added to obtain the Total Risk Severity Weight (TRSW) for that test case. The TRSW values for test cases are summed to get the Grand Total Risk Severity Weight (GTRSW) for the test suite. Table VIII

[1] $Tcm = \frac{NBD}{Max(NBD)} + \frac{MCC}{Max(MCC)} + \frac{LOC}{Max(LOC)}$

shows sample RSW, TRSW, and GTRSW values calculated for the original test order of *iTrust* Version 2. Columns 2, 3, and 4 contain RSW values of sample test cases. For example, when we execute 12.5%, 25%, 50%, and 75% of the original order of test cases, the PTRSW values are 0.83%, 3.94%, 46.68%, and 57.26%, respectively. As we can see from the results in the next section, these values are relatively low compared to other techniques, which means the original order of test cases is not able to detect faults in the risky components early.

After collecting all the required data, we performed prioritization techniques. We calculated the APFD and PTRSW values for each test case obtained from the techniques.

D. Threats to Validity

Construct Validity: The risk identification is subject to human judgment (in our case, a graduate student). Therefore, the results can be biased by the personnel's knowledge and experience.

Internal Validity: In this study, we assumed that there are no project or process risks associated with the requirements. We only considered the software product risk category when evaluating the RE of risk items. This assumption potentially influences the effectiveness of the technique but can be minimized by considering more product risk items.

External Validity: Our experiment used the *iTrust* system, a mid-size, open source application which has requirements documents. This limitation to generalization can be addressed by using different sizes of industrial applications with future studies.

IV. Data and Analysis

In this section, we present the results of our study and data analyses for each research question. (We discuss further implications of the data and results in Section V.)

A. The effectiveness of requirements risk-based prioritization for improving the rate of fault detection of test cases (RQ1)

Our first research question (RQ1) considers whether a requirements risk-based approach can help improve the effectiveness of test case prioritization. To answer this question, we compare techniques based on the results shown in Table IX. The first column in the table lists the control techniques. CS-10 and CS-20 in the cluster-based techniques denote the cluster size that we explained in Section III-B. The second column represents the average APFD value of each control technique for version 1, and the third column shows the improvement rates of our requirements risk-based approach over the control techniques. Subsequent columns repeat the same format (APFD values of the controls and the heuristic's improvement rates over the controls) for versions 2 and 3.

The results in Table IX indicate that requirements risk-based prioritization outperformed the first two control techniques (*Torig* and *Tcm*) across all versions; the improvement rates ranged from 24% to 96%. In the case of cluster-based control techniques, the results varied. At a cluster size of 10 (CS-10), the heuristic outperformed the controls for the first two versions, but for version 3, it was not better than the controls. At a cluster size 20 (CS-20), the heuristic did not yield improvement over the controls.

To visualize our results, we illustrate them in boxplots as shown in Figure 2, which presents the APFD values for all techniques and versions. The horizontal axis corresponds to the techniques while the vertical axis represents the APFD values. Each boxplot for version 1 has 13 data points, and for versions 2 and 3, 12 data points are presented.

Examining the boxplots, we observe similar trends from the tables (average values). For versions 1 and 2, the heuristic outperformed the control techniques except for two cases, and for version 3, the heuristic only outperformed two control techniques. The boxplots for version 1 show a wider distribution than the other two versions, and the distance between the minimum and maximum values is very wide for *Tcop-20* and *Trb*. For version 2, the data distributions across techniques are similar, and they do not show extremely low or high values unlike the results for version 1. For version 3, the control techniques that used clustering with a cluster size of

TABLE VIII
Example Test Cases, TRSW and Associated Data for Original Test Order-Version 2

Test cases (Orig v2)	Mutations			TRSW	Percentage of Total Risk Severity Weight (PTRSW)
	M1	...	M54		
TC1	3	0	1	4	
:	0	0	0	0	
TC18	0	0	0	0	
:	0	0	0	0	
Sub-total of TRSW after 12.5% test case execution				4	0.83%
:	:	:	:	:	
TC36	0	0	0	0	
Sub-total of TRSW after 25% test case execution				19	3.94%
:	:	:	:	:	
TC71	0	0	0	0	
Sub-total of TRSW after 50% test case execution				225	46.68%
:	:	:	:	:	
TC107	0	0	0	0	
Sub-total of TRSW after 75% test case execution				276	57.26%
:	:	:	:	:	
TC142	0	0	0	0	
Grand total after execution the entire test suite (GTRSW)				482	

TABLE IX
APFD COMPARISON AND IMPROVEMENT OVER CONTROLS

Control Technique	$iTrust-v1$		$iTrust-v2$		$iTrust-v3$	
	APFD	Improvement over controls (%)	APFD	Improvement over controls (%)	APFD	Improvement over controls (%)
Torig	43.76	37	47.81	27	28.83	96
Tcm	45.77	31	48.80	24	44.84	26
Tcl-orig-prior($CS-10$)	43.59	38	57.28	6	70.60	-20
Tcl-cm-prior($CS-10$)	39.51	52	57.78	5	75.31	-25
Tcl-orig-prior($CS-20$)	72.37	-17	63.42	-4	72.06	-22
Tcl-cm-prior($CS-20$)	75.45	-20	65.33	-7	72.41	-22

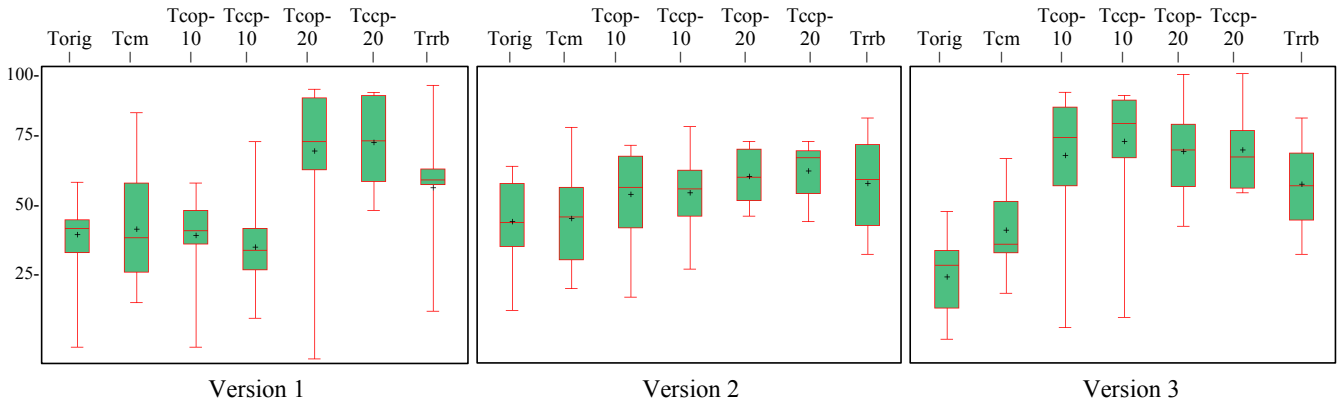

Fig. 2. APFD Boxplots For All Controls And Heuristic

10 show a wider distribution than other techniques. When we examined the *iTrust* requirements, we found that newly added requirements caused a considerable number of code modifications. The cluster based prioritization technique used code modification information and requirement implementation priority as the primary criteria to prioritize requirement clusters. This factor might have influenced the cluster based approach to produce better results over our approach.

B. The effectiveness of requirements risk-based prioritization for finding faults associated with risky components (RQ2)

With RQ1, we examined whether prioritization that utilizes requirements and their risks can be effective at finding faults earlier. The results are encouraging, but we do not know whether the early detected faults are, indeed, the faults that reside in the risky components. Thus, our second research question (RQ2) considers whether the requirements risk-based approach can be more effective in finding faults associated with risky components early compared to the control techniques.

To investigate this research question, we measured the PTRSW values that we described in Section III-B, which show how fast the technique can detect faults in the risky components. For this evaluation, we considered two control techniques: original (*Torig*) and the cluster-based technique with a cluster size of 20 that produced the best results (*Tccp*). Table X shows the results when we foreshorten the test execution process by 12.5%, 25%, 50%, and 75%. This means that, for a 50% cutting cutting ratio, we simulate the effects of having the testing process halted half way through.

The results show that our risk based approach can detect more faults in the risky components early than the controls except for two cases (*Tccp* at 12.5% and 25% execution rates for version 3). In particular, our approach produced relatively high fault detection rates at 50% for version 1 and 75% for version 2. These results indicate that the use of requirements risks during prioritization was effective in locating faults that reveal risks early. Further, even when companies need to cut their testing process short due to their product release schedule, still can still identify and fix more important faults under the limited time budget than otherwise.

Figure 3 shows the results graphically. The horizontal axis shows the percentage of test execution, and the vertical axis represents the PTRSW values. This figure includes the results for a 100% test execution rate. As we observed from the table, our approach outperforms the controls at all percentage levels for versions 2 and 3, but for version 3, our approach is not better than *Tccp* under the 50% test execution rate. Again, we speculate that using code modification information worked better for version 3 because version 3 underwent a major code modification that included a large number of requirements changes. From this observation, we think that combining our approach with code modification information for versions that go through major changes could improve the outcome.

V. DISCUSSION AND IMPLICATIONS

From our results, we drew the following observations. First, our results suggest that the use of requirements and their risks

TABLE X
PERCENTAGE OF TOTAL RISK SEVERITY WEIGHT (PTRSW) FOR DIFFERENT TETS EXECUTION LEVELS

Version	Method	Percentage of Total Risk Severity Weight (PTRSW) (%)			
		Execution Rate (12.50%)	Execution Rate (25%)	Execution Rate (50%)	Execution Rate (75%)
Version 1	Original Order	0.00	0.00	26.49	70.90
	Req-Cluster	7.46	7.46	30.97	71.00
	Req-Risk	21.64	30.22	74.63	82.00
Version 2	Original Order	0.00	0.00	26.49	70.90
	Req-Cluster	7.46	7.46	30.97	71.00
	Req-Risk	21.64	30.22	74.63	82.00
Version 3	Original Order	0.00	0.00	26.49	70.90
	Req-Cluster	7.46	7.46	30.97	71.00
	Req-Risk	21.64	30.22	74.63	82.00

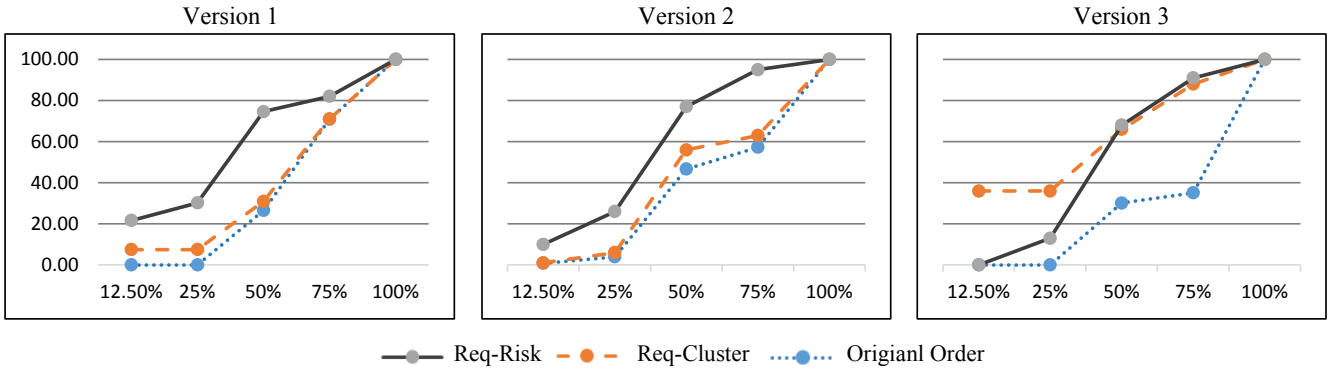

Fig. 3. PTRSW Comparison Graphs For All Versions of *iTrust*

can help improve the effectiveness of test case prioritization. The proposed technique performed better than techniques that used original order and source code information, but for the clustering techniques, the results were not consistent (in particular, for version 3). As we discussed in Section IV, we speculate that the use of prioritization factor and code change pattern in version 3 affected this outcome. The source code of version 3 was considerably affected by requirements changes or new requirements. For this situation, the use of code modification information would work better in identifying test cases that exercise more error-prone components. This observation indicates that we might need to consider different test case prioritization factors under different circumstances.

Second, our study results strongly support the conclusion that the use of requirements and their risks can help early detection of faults that reside in the risky components. The requirements associated with the critical risk items are assigned high priorities and those requirements are further prioritized depending on the degree they correlate with additional requirement-risks related factors (requirements modification and volatility). Hence, test cases associated with requirements having more risky behaviors get top priorities in the test suite, and this mechanism is the primary reason for the successful early detection of faults that reside in the risky components.

The results of the study provide important implications for software industry. Typically, companies that build safety or mission critical systems spend enormous time and effort in testing because the consequences of system failure often involve high costs and can even be life threatening. Also, the majority of modern software systems are web-based and deployed through the internet. This means that software systems are vulnerable to various security attacks, and thus they require thorough testing. For both cases, finding critical faults early plays an important role in producing reliable systems more cost-effectively. Further, as we discussed in the previous section, the proposed technique was able to detect more critical faults early under the limited time slots. Thus, using our approach, companies can manage their production schedule more effectively by fixing costly faults as early as possible.

VI. RELATED WORK

This section discusses the existing work related to test case prioritization and risk-based prioritization.

Test case prioritization techniques reorder test cases to maximize some objective function, such as improving rate of fault detection. Due to the importance in practice, many researchers have proposed and studied various test case prioritization techniques [3]. While the majority of test case prioritization techniques have utilized source code information [4], [22], [23], other types of software artifact have been considered, such as system requirements and design documents [6], [7], [17]. For instance, Srikanth et al. [8] presented a prioritization technique applied at the system level using system requirements and fault proneness of requirements. Krishnamoorthi and Mary [7] attempted to improve the rate of severe fault detection by prioritizing test cases using factors related to

requirement specification, such as customer priority or requirements changes. Arafeen and Do [6] proposed a requirements cluster-based prioritization that uses a text-mining technique that provides a means to cluster relevant requirements. These studies reported that using requirements information improved the effectiveness of prioritization.

Other researchers have used risk information along with requirements or design documents to prioritize test cases so that they can run test cases to exercise code areas with potential risks as early as possible [9], [10], [11]. Chen et al. [9] utilized activity diagrams to derive test cases and prioritized those test cases by using the risk exposure values obtained through risk analysis with fault history information. Stallbaum et al. [10] proposed a RiteDAP (Risk-based test case Derivation And Prioritization) technique which uses activity diagrams to generate test cases and prioritizes the generated test cases based on the risks associated with fault information. Yoon et al. [11] used the relationship between risks in the risk items and test cases to evaluate prioritized test cases. They identified risk items (e.g., handling inputs/outputs or user interactions) and analyzed their risk levels based on requirements risks. While these values were used to evaluate the prioritized test cases, a direct relationship between requirements and test cases was not used for prioritizing test cases. Unlike these studies, in our work, we focus on identifying the risky requirements using a set of risk levels for various risk items. We prioritized test cases through a direct relationship between requirements risks and test cases so that we can identify test cases that can expose potential risk early. Also, we provide an adequate evaluation method for risk-based prioritization.

VII. Conclusions and Future Work

In this paper, we proposed a new test case prioritization method, which is based on requirements risks information, and assessed the approach by comparing other existing techniques. Our results indicated that the requirements risk-based prioritization can be effective in detecting faults early and even better in locating faults with the risky components early. With the proposed approach, software companies can manage their testing and release schedules better by providing early feedback to testers and developers so that they can fix the problems as soon as possible.

As we discussed in Section IV, for the case of *iTrust* version 3 that underwent a large number of code modifications and requirements changes, our approach was not better than the requirements clustering-based approach that utilized code modification information. Therefore, for future work, we plan to develop hybrid techniques that use both requirements risks and code metric information so that we can selectively apply techniques that consider software systems' characteristics. Further, we only used one application in this study, mainly because many open source applications do not have requirements documents, but we plan to seek more applications with requirements including industrial applications, so that we can replicate our study to see our findings can be generalized.

Acknowledgments

This work was supported, in part, by NSF CAREER Award CCF-1149389 to North Dakota State University.

References

[1] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *IEEE TSE*, vol. 28, no. 2, pp. 159–182, Feb. 2002.

[2] A. Srivastava and J. Thiagarajan, " Effectively prioritizing tests in development environment," in *Proceedings of the International Symposium on Software Testing and Analysis*, Jul. 2002, pp. 97–106.

[3] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritisation: A survey," *JSTVR*, pp. 67–120, Mar. 2010.

[4] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel, "The effects of time constraints on test case prioritization: A series of controlled experiments," *IEEE TSE*, vol. 26, no. 5, Sep. 2010.

[5] X. Qu, M. Cohen, and G. Rothermel, "Configuration-aware regression testing: An empirical study of sampling and prioritization," in *ISSTA*, Jul. 2008, pp. 75–86.

[6] M. Arafeen and H. Do, "Test case prioritization using requirements-based clustering." *International Conference of Software Testing, Verification and Validation (ICST)*, Mar. 2013.

[7] R. Krishnamoorthi and S. Sahaaya Arul Mary, "Factor oriented requirement coverage based system test case prioritization of new and regression test cases," *Information and Software Technology*, vol. 51, no. 4, pp. 799–808, 2009.

[8] H. Srikanth, L. Williams, and J. Osborne, "System test case prioritization of new and regression test cases," in *ESE*, Aug. 2005, pp. 64–73.

[9] Y. Chen, R. Probert, and D. Sims, "Specification-based regression test selection with risk analysis," in *Proceedings of the 2002 Conference of the Centre for Advanced Studies on Collaborative research*, Sep. 2002.

[10] H. Stallbaum, A. Metzger, and K. Pohl, "An Automated Technique for Risk-based Test Case Generation and Prioritization," in *Proceedings of the 3rd International Workshop on Automation of Software Test*, May 2008, pp. 67–70.

[11] M. Yoon, E. Lee, M. Song, and B. Choi, "A test case prioritization through correlation of requirement and risk," *Journal of Software Engineering and Applications*, vol. 5, no. 10, pp. 823–835, 2012.

[12] J. Bach, "Risk and requirements-based testing," *IEEE Computer*, vol. 32, no. 6, pp. 113–114, 1999.

[13] D. Wallace and D. Kuhn, "Failure Modes in Medi-cal Device Software: An Analysis of 15 Years of Recall Data," *Reliability, Quality and Safety Engineering*, vol. 8, no. 4, pp. 301–311, 2001.

[14] S. Amland, "Risk-based testing: Risk analysis funda-mentals and metrics for software testing including a financial application case study," *Journal of Systems and Software*, vol. 53, no. 3, pp. 287–295, 2000.

[15] E. Triantaphyllou and K. Baig, "The impact of aggregating benefit and cost criteria in four MCDA methods," *IEEE Transactions on Engineering Management*, vol. 25, no. 2, pp. 213–226, Feb. 2005.

[16] T. Graves, "Predicting fault incidence using software change history," *IEEE TSE*, vol. 26, pp. 653–661, Jul. 2000.

[17] H. Srikanth and L. Williams, "On the economics of requirements-based test case prioritization," in *Int'l Workshop on EDSER*, May 2005, pp. 1–3.

[18] N. Schneidewind and H.-M. Hoffman, "An experiment in software error data collection and analysis," *IEEE TSE*, vol. 5, no. 3, pp. 276–286, May 1979.

[19] W. Zage and D. Zage, "Evaluating design metrics on large-scale software," *IEEE TSE*, vol. 10, pp. 75–81, 1993.

[20] A. Malishevsky, G. Rothermel, and S. Elbaum, "Modeling the cost-benefits tradeoffs for regression testing techniques," in *ICSM*, Oct. 2002, pp. 204–213.

[21] H. Do and G. Rothermel, "On the use of mutation faults in empirical assessments of test case prioritization techniques," *IEEE Transactions on Software Engineering*, vol. 32, no. 9, pp. 733–752, Sep. 2006.

[22] G. Rothermel, R. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE TSE*, vol. 27, no. 10, pp. 929–948, Oct. 2001.

[23] M. Sherriff, M. Lake, and L. Williams, "Prioritization of regression tests using singular value decomposition with empirical change records," in *ISSRE*, Nov. 2007, pp. 81–90.