ABSTRACT
Smart Grid systems are kind of System of Systems with distributed and highly heterogeneous software connected to provide various services. Early knowledge of defect prone parts is useful for improving the safety and maintenance of these kinds of systems. We report the results of an empirical study of using reused components to predict future defective components in a type of open source Smart Grid application (transmission and operation domain of Smart Grid). Our results showed that reused components of this Smart Grid application are strong predictors of future defective components. The model’s best predictors gave an average recall of 0.92 (average precision of 0.406) when tested across three future releases. Which implied that 92% of predicted defective components in the next release turned out to be defective. This model can be employed to tailor quality assurance (QA) efforts in a way that blind spots are avoided in such critical system and QA effectiveness significantly improved.

KEY WORDS
Component; Smart Grid; prediction model; empirical study; Component defect-proneness; Import types; System of Systems.

1. Introduction
Reusing a software component (class, package or library) for performing certain system functions is inevitable in application developments, especially object-oriented based systems. Many empirical studies have been conducted that focused on finding correlation between design metrics of object-oriented (OO) components and their external quality attributes [7, 12-18]. Several studies have shown reuse to increase productivity and quality of software [1-3, 20]. Similarly, some authors showed reused components to have fewer defect densities in comparison to non-reusable components [2, 20]. In contrast, Schroeter et al. [10] demonstrated that a component that reuses another component might become defective because the component it depends on is defect prone.

Additionally, several other properties such as cohesion, complexities and coupling among modules and components are shown through empirical studies [7, 12-18] to be correlated to fault rates. These studies seek to explain the correlational relationship between component structural properties and its defect-proneness.

Furthermore, Schroeter et al. [10] and Duala-Ekoko et al. [11] showed that imported types have high correlation with defect-proneness of the Eclipse software. In this approach, import types (classes, files and packages) between components are analyzed and used to rank and classify components reusing them as defect-prone or not defect-prone. The authors analyzed 2 releases of eclipse project and the best results reported were predictions at the package level. While Schroeter et al. used the equal-blame approach that means the usage type is responsible for the fault reported in the respective component using it, Duala-Ekoko et al. reported that failure-associated type is an imported type referenced within methods modified in order to fix a failure and that this type provides more accurate understanding of type usage and the component failure. These two studies provide measures to predict fault proneness given a particular usage types.

Our hypothesis is that the approach of Schroeter et al. [10] can also be applied to Smart Grid applications with many external dependencies (reused components). Thus, high ranked and consistent predictors across releases of an application can serve as early pointer of defect-proneness to components that depend on them. In addition to the prediction model, this can be used to allocate testing efforts to new components that reuse such components. The significant contributions of our work are: First, the domains of applications. We have applied the approach to an application in Smart Grid system as against Eclipse software used in the previous studies. Hall et al. [24], in a systematic literature review on fault prediction performance reported that context such as application area and programming language is a key factor in model performance. An integrated development environment such as eclipse with mostly internal dependencies within project is different from an application such as Smart Grid software that typically has many external dependencies.

To the best of our knowledge no study has been reported in this domain. As a result of the domain difference, we differ in the notion of good and bad “imports” (reused components). The insight is that, a reused component of good quality can have a high probability of being defect prone to components that
dependent on it simply because of the nature of services or functions it offers or for other reasons. Third, we showed that the trustworthiness of prediction models based on this approach could be established with statistically tested data for normality and measures of central tendencies.

The rest of the paper is organized as follows: Section 2 discusses the background to our work, such as the domain and the clarification of terms used. In section 3, the case study used and data collection approach is presented. The results of the study are presented in Section 4. Section 5 provides discussions of the results and threat to its validity. Finally, the conclusion and future work is given in section 6.

2. Background

2.1 Choice of Case Study

In this study, we performed an empirical study on Smart Grid application, a type of system of systems (SoS) applications. Several domains of Smart Grid exist such as the generation, transmission, distribution, markets, operations, service providers and consumers [5] with different types of software running in these domains (legacy systems and new applications). This software, in addition, performs different functions and provides different services.

A discussion of evolution complexity and challenges of Smart Grid applications as compared to monolithic software is given in Anvari et al. [4]. Our motivation for the choice of this case study is that, as a critical infrastructure, the availability and reliability of the Smart Grid is crucial to its safety and security. Smart Grid represents the injection of Information and Communication Technology (ICT) infrastructure to the electricity grid to allow for bi-directional flow of energy and information [5, 8, 9]. Smart Grid is still in the formation stage, and represents a shift from relatively closed grid to a more complex and highly interconnected systems.

Although, efforts are in place to develop interoperable standard, there is still substantial gap identified as a results of evolving requirements and different implemented software and hardware products [6]. The fact that these systems have to interoperate poses a quality challenge for the entire system. For instance, if software for collecting data from field devices that are designated for monitoring the health or quality of the transmission line fails as a result of software defect, the end operator (human or automated device) is denied real-time data for taking adequate control action. The data collection software could be viewed as less important when compared to substation controller system.

It is common to focus QA efforts on the most important parts of the system, however, as noted by Zhang et al [25], a perceived less important system in a SoS can be ignored for quality assurance, thus leaving blind spots in the system that can compromise such other high assurance systems. A case cited involved a surprise anomaly that resulted into modification of ground software at NASA which is a perceived less important part when compared to the on-board guidance system. The on-board guidance system is typically focused for QA. The lesson is that, although it is not feasible to allocate QA resources to all aspects of software, defect prediction models can be employed to tailor QA efforts in a way that blind spots are avoided.

Thus defect prediction models are needed to support QA focus on different and identified defective parts of the various Smart Grid applications. A defect that results into failure in one part of such critical system can have cascading and fatal effects on the rest of the Smart Grid systems. Since different software will drive different Smart Grid systems and many of the system will interoperate and integrate together, early detection of defect prone areas and components of these systems is important for improving the overall quality of the grid.

2.2 Clarification of Terms

We clarify the following terms that are used in the paper as follows:

- A **component** can either be a class, file, package or an external library.
- An **external library** is a reused component that is commonly regarded as a 3rd party component.
- An **internal library** is a reused component that is internal to the application and is referred as package or module.
- A **defect** can be a bug or fault that is detected in a component. A defect may or may not lead to a system failure.
- A **reused component** in the context of this paper is an import type and can be internal or external libraries. For instance, it may be possible for a component to import N number of components with keyword “using” in C# or “import” in java.

3. Case Study

Our case study is a medium-sized Smart Grid open source software (OSS) named openPDC [19], supported by the Tennessee Valley Authority (TVA). The solution is developed using .NET Framework and mainly with C# programming language. The openPDC is a phasor data concentrator software that is designed to process real time data for user-defined actions and for archival purpose. This type of Smart Grid application is used to monitor the health of the electricity transmission grid in other to take appropriate control actions based on real-time data from the phasor devices [26].

The first stable release of the software was published on Jan. 2010 and eight stable releases have been published till date. Table 1 provides the description of the dependencies. The openPDC solution reuses both the components (libraries) in these dependencies (TVA, Timeseries, Historian and Synchrophasor libraries) and other third party libraries. An example of a third party library is oracle or sql database libraries. In addition, the solution reused components from the .NET framework.

These types of library components begin with the “System” namespace.
3.1 Data Collection

The openPDC project maintains an Issue Tracker repository that allows both developers and users to log various defects that are found in the releases. In addition, it maintains a SVN repository that logs every version that is changed either for corrective (bug fixed), preventive, adaptive or perfective purposes [20]. The reasons for making the changes are sometimes included. Further, every change set can be successfully traced to the class and package that are involved with this change. Developers occasionally include the change set that contains the bug fixed in the comment field in Issue Tracker.

We developed scripts (Figure 1.) that extracted all files, packages and assemblies from the source files and computed for each file, the package dependencies and the type (interface, abstract or concrete). We used simple syntactic analysis to identify class import type such as inheritance, composition, aggregation that is not explicitly declared. Additionally, a database schema was used to store and extract outgoing dependencies of each package and assemblies for further post processing. We also manually looked up each resolved defect’s change set in the SVN repository to identify the classes or files and packages affected by this change. We observed that certain defects captured in the SVN repository are not logged in the issue tracker. Thus, we looked out for keywords such as “fixed”, “corrected”, “bug”, “resolved” and “issue” in the SVN repository and included the result in the analyzed data. For this study, we did not separate pre-release defects from post-release defects.

![Figure 1. Data extraction process from SVN and defect repositories](image)

### 4 Results

#### 4.1 Defects Distribution

Table 2 shows the summary of the release data extracted from the repository. It is medium sized software with approximately 124KLOC and 625 class files as at version 1.4SP2. In v1.3.11, 106 defects are recorded with 120 defective files from the total of 556 files (see Tables 2 and 3). Moreover, in v1.4SP2, 33 files out of 625 (probability of defect = 0.0528) files are responsible for the total of 43 defects in the release. This probability of 0.0528 shows that a random pick of a file in the release has a reliability of 0.0528 out of 1 of being defective. For our study, we used v1.3.11 upward. To further understand the distribution of the defect data, we determined the average of defects produced by the defective components. As pointed out in Zhang et al. [25], learning from defect dense components is useful knowledge in defect prediction modeling. Given that, a defective component can have a certain number of defects in a particular release that is greater or equal to one. Therefore, for all defective components, the average (mean) of defects per defective component is calculated as (see Table 4):

\[
\text{Average of defects per defective component} = \frac{\sum_{i=1}^{n} d_i}{n}
\]

Where: defective component: \(\{c_1, c_2, c_3 \ldots c_n\}\) maps to defect values: \(\{d_1, d_2, d_3 \ldots d_n\}\) ∀ \(d_i > 0\)

<table>
<thead>
<tr>
<th>Release</th>
<th>Date</th>
<th>#Pkgs</th>
<th>#Files</th>
<th>KLOC</th>
<th>#Defect</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1.0</td>
<td>Jan 2010</td>
<td>39</td>
<td>375</td>
<td>56.74</td>
<td></td>
</tr>
<tr>
<td>v1.1</td>
<td>Jun 2010</td>
<td>45</td>
<td>417</td>
<td>74.42</td>
<td></td>
</tr>
<tr>
<td>v1.2</td>
<td>Sept 2010</td>
<td>67</td>
<td>555</td>
<td>86.9</td>
<td></td>
</tr>
<tr>
<td>v1.3.11</td>
<td>Oct 2010</td>
<td>66</td>
<td>556</td>
<td>89.36</td>
<td>106</td>
</tr>
<tr>
<td>v1.4</td>
<td>Mar 2011</td>
<td>73</td>
<td>584</td>
<td>103.17</td>
<td>13</td>
</tr>
<tr>
<td>v1.4 SP1</td>
<td>May 2011</td>
<td>73</td>
<td>585</td>
<td>104.89</td>
<td>108</td>
</tr>
<tr>
<td>v1.4 SP2</td>
<td>Dec 2011</td>
<td>76</td>
<td>625</td>
<td>123.95</td>
<td>43</td>
</tr>
</tbody>
</table>

Table 1

openPDC Solution overview

<table>
<thead>
<tr>
<th>Solution</th>
<th>Description</th>
<th>Assemblies</th>
</tr>
</thead>
<tbody>
<tr>
<td>TVA Framework</td>
<td>It is the &quot;code library&quot; that all other solutions and code depend on.</td>
<td>TVA.Core.dll, TVA.Communication.dll,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TVA.Windows.dll, TVA.ServiceProcess.dll,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TVA.ServiceModel.dll, TVA.Web.dll,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TVA.Security.dll</td>
</tr>
<tr>
<td>TimeSeries-</td>
<td>Core framework used to manage, process and respond to dynamic changes in</td>
<td>TVA.Historian.dll</td>
</tr>
<tr>
<td>Framework</td>
<td>fast moving streaming time-series data in real-time.</td>
<td></td>
</tr>
<tr>
<td>Historian</td>
<td>Contains all code that is used for archiving data within the openPDC;</td>
<td></td>
</tr>
<tr>
<td>Synchrophasor</td>
<td>Contains all code related to the primary system executables of the openPDC</td>
<td>TVA.PhasorProtocols.dll</td>
</tr>
<tr>
<td></td>
<td>as well as the phasor protocol parsing and generating library.</td>
<td></td>
</tr>
</tbody>
</table>

Table 2

openPDC release summary extracted from repositories
The probability that a component that does not reuse C will contain defect is computed as:

\[ P(\text{defect} \mid \neg C) = \frac{N_{pd} - N_{cd}}{N_T - N_c} \]

The proportion data is the probability of defect \( P(\text{defect}|C) \) as explained in the previous data. To justify the use of 2-tailed test for separating significant imports from non-significant ones, we tested the data for normality using the Shapiro-Wilk normality test. The p-value (0.2244) as shown later in Table 11 is greater than the confidence level 0.1; we therefore do not reject the hypothesis that the data is from a normally distributed population. Additionally, both the skewness and kurtosis statistics as shown are not statistically significant. We used the rule of thumb by comparing the absolute values of both the skewness and kurtosis statistics to twice their standard errors. Both values are less than twice their standard errors as observed.

Subsequently, we separated defect-prone components from less defect-prone by using a 2 tailed-tests with 0.05 confidence interval to test whether the computed probability for each import type is significantly higher or lower than the average probability of defect occurrence. The proportion and the p-values of import types in v1.3.11 are reported in Table 5. The results of the classified import types based on the 2 tailed tests of significance are summarized in Table 6. Of a total of 88 import types in v1.3.11, 25 are tested to be statistically significant.

### 4.2 Feature Selection

We separated the defect-prone usage types from non-defect-prone usage types using the approach in Schroeter et al. [10]. First, we computed the probability of finding a defect in components that reused another component C (file or package), that is \( P(\text{defect}|C) \). Second, we compute the probability of a defect when not reusing component C. In order words, the likelihood of a component becoming defective when C is not used. Lastly, for every probability \( P(\text{defect}|C) \), we tested with two t-tests at 0.05 confidence interval whether they are significantly higher or lower than the average probability of defect. The average probability of a defect occurring in a component is calculated as:

\[ P(\text{defect}) = \frac{\text{Total no of defective components}(N_d)}{\text{Total no of components}(N_T)} \]

The probability of a component becoming defective by reusing another component C is given as:

\[ P(\text{defect}|C) = \frac{\text{Total no of defective components that reused } C \ (N_{cd})}{\text{Total no of components that reused } C \ (N_c)} \]
4.3 Classification Model

In order to answer our research question, we built a classification model by using the statistically tested import types that are significant as potential predictors or explanatory variables for the model. The import types of every component are filtered through the significant import types of the trained model (Figure 2). In other words, if we create a model on a release, then the set of explanatory variables or predictors are those import types that are tested significant for this release. As a result, the output of the filter block are set of predictors $P_1...P_n$ that take on binary values 0 or 1 as previously discussed. The output of the first model is either category “Yes”, that means the component is defective or “No” that implies not defective. For model type 2, the output are three categorical variables: High indicates components with defect number higher than the average of defects per defective component, Low indicates components with defect number lower than the average of defects per defective component and No indicates no defect. We trained our model with both NaiveBayes and Random Forest algorithms in Weka [21]. The main objective is to test if reused components can be employed to predict defective components in this Smart Grid application.

NaiveBayes is a probabilistic supervised learning algorithm [22] that uses the method of maximum likelihood. It uses probability inference on high dimensional data by computing the conditional probabilities along attributes axes and classifying the target attribute based on the maximum likelihood of each attribute. Random Forest [23] is a supervised classification algorithm that ensembles many decision trees to predict a target class based on a set of input variables. The final prediction is the most frequent component voted by all trees in the forest. In other words, the final predicted component represents the component with the most votes. To train our model, a previous release is analyzed and the significance of the import types statistically tested for defect-proneness. We trained our model by using cross validation method on the transformed dataset. In this approach, a training dataset is divided into 10-folds and the model is trained on each fold and the result cross-validated on the rest folds in each iteration. The objective is to increase the model’s ability to generalize well on future unseen dataset.

The quality of the prediction model is measured by the recall, accuracy and precision metrics.

**Recall** is computed as the ratio of correctly predicted defect-prone components to the total defect-prone components in the release.

$$\text{Recall} = \frac{\# \text{correctly predicted defective components}}{\# \text{all defective components}}$$

**Precision** measures the percentage of correctly classified defect-prone components to the total number of components that are predicted as defect-prone.

$$\text{Precision} = \frac{\# \text{correctly predicted defective components}}{\# \text{all predicted defective components}}$$

**Accuracy** provides the overall prediction power of the model. It measures the percentage of correctly predicted defect-prone components and correctly predicted non defect-prone components to the total number of all components.

$$\text{Accuracy} = \frac{\# \text{correctly predicted components}}{\# \text{all components}}$$

We have trained a classification model using three different releases and tested at each time on the future releases. We considered only forward predictions, because backward predictions appear not to be useful in practice.

We report the results of 2 different classification models trained under the same experimental conditions that are necessary for our research goals. By experimental conditions, we mean the same type of algorithms and algorithm parameters. The results showed the recall and precision values for defect-prone component classification and the accuracy value for correct classification of both defect-prone and non defect-prone components.

1. **Model Type 1** – Predicting components as defective and not defective (Yes/No)
2. **Model Type 2** – Predicting components using defect density categories (High, Low and No defect)

Tables 7 and 8 report some of the results of the 2 model types. In these models, we predicted for both files and packages. As shown from the results, there is a recall of over 90% from test performed on some future releases of models trained on earlier releases. For instance, **Model Type 1** trained on v1.3.11 package, produced a recall of 1.0 (100%) when tested on v1.4 and v1.4SP2. A look at the confusion matrix in table 9 revealed that all the 12 defective packages in v1.4SP2 are correctly classified as defective with model trained on v1.3.11. Likewise, 20 out of 24 defective files in v1.4 are correctly classified as defective with the model trained on v1.3.11.
The results for Model Type 2 show recall values as high as 1.0 (100%) for most defect dense components in some of the tests performed. For example, the model trained on v1.3.11 at the file level when tested on v1.4 and v1.4SP1 gave a recall of 1.0 on component category that is classified as “High” (Defect dense components). The same effect is observed at package level with recall value sometimes as high as 100%. The presented confusion matrix in table 10 shows for instance, all the 6 most defect dense file components correctly classified in v1.4 with model trained on v1.3.11. Also, all the 4 most defect dense packages are correctly classified in v1.4 with model trained on v1.3.11.

Model Type 2 may be preferred if Model Type 1 yields recall value of below 1.0 (100%). The reason is that, the misclassified defective components in Type 1 could be the most defect dense components in the release. Take for instance the 4 misclassified components in v1.4 could as well be part of all the 6 most defect dense components in the release.

Table 9
Confusion matrix Model Type 1: Random Forest:

<table>
<thead>
<tr>
<th>Actual</th>
<th>Predicted</th>
<th>Y</th>
<th>N</th>
<th>Y</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package Train– v.1.3: Test v.1.4</td>
<td>20</td>
<td>4</td>
<td>12</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>File Test v.1.4</td>
<td>80</td>
<td>480</td>
<td>22</td>
<td>42</td>
<td></td>
</tr>
</tbody>
</table>

Table 10
Confusion matrix Model Type 2: Random Forest:

<table>
<thead>
<tr>
<th>Actual</th>
<th>Predicted</th>
<th>H</th>
<th>L</th>
<th>N</th>
<th>H</th>
<th>L</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package Train– v.1.3: Test v.1.4</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>File Test v.1.4</td>
<td>8</td>
<td>6</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

5. Discussions

These results clearly demonstrate that reused components can be focused as predictors of defective components in future projects of this Smart Grid application. In practical use, defect predictors are useful for adjusting quality assurance (QA) budgets to focus on blind spots especially for critical infrastructure [25]. In addition, this approach can be used to indicate early warning signs of any possible defect for dependent components. The possible set of predictors (import types) provides the possibility of defect in the newly dependent components. For example, TVA.Data has a likelihood of 1.0 in v1.3.11. This means that all the components that have reused TVA.Data are found with defects. In v1.4, it has a probability of about 0.6, which indicates that 60% of all components that reused it are found with defects. Since, reusability promotes stability, a shortfall of this modelling method is therefore the possibility of the model to underperform when the internally reused components mature overtime.

5.1 Model Performance

The proportion data for predictors of each release we used are tested for normality. As shown in Table 11, only the proportion data for v1.3.11 (files and packages) and v1.4SP1 (packages) tested positive for normality. We compare the results of models trained with these versions to the rest. From observations, the recall performance of the models seem consistent with the measures of their central tendencies (mean, median and 3rd quartiles) in addition to testing positive to normality. The best model recorded is trained on v1.3.11 for package component with recalls of 1.0, 0.76 and 1.0 when tested on v1.4, v1.4SP1 and v1.4SP2 respectively. Model trained on v1.3.11 for file prediction also showed strong recall performance of 0.917, 0.714 and 0.697 when tested on v1.4, v1.4SP1 and v1.4SP2 respectively.
5.2 Good and Bad Imports

In the work reported by Duala-Ekoko et al. [11], the authors argued against the equal-blame approach by Shroeter et al. [10]. They showed that not all imports by a component could be blamed for the component’s defect. By associating the change performed on the component as a result of the defect to the actual import, it is possible to know the source of the defect. The likelihood performed in [10] on an eclipse project categorized imports as “good” when they have lower likelihood of producing defective components when compared to the average likelihood of defect occurrence and “bad” when their likelihood of producing defective components is higher.

Our study revealed that such categorization could be misleading when dealing with application. For instance, “System.Data” or “System.Net” are .NET framework components (external libraries) reused by the application. Based on the equal-blame approach, these components have significant high likelihoods of producing defective components, not because they are “bad” imports, but possibly because of the nature of services they provide for this type of application that can make components depending on them prone to defects. Thus reused components may have high quality, but still have high probability of producing defective components due to the nature of its functions, the way it is implemented and for many other reasons.

5.3 Threats to Validity

We have performed analysis and evaluation of a Smart Grid system using data from a single environment. Thus, we cannot claim that this kind of pattern or related will be visible in other Smart Grid systems. Although, the application that we studied is open sourced, yet it is stable and consists of stable libraries that have been used over time by credible institutions and subsequently released to the general public. However, as it is with most case studies, we cannot generalize these results across all Smart Grid systems. Our model and assumptions are constructed and validated for this system alone. Further studies will be necessary to compare results across several Smart Grid domains.

As the case is with an evolving system, frequent changes and refactoring are inevitable. Smart Grid systems are in this category. Changes such as additions, deletions and modifications are frequent within the system, thus packages or even classes can be refactored with different names in the later releases. This can be a threat to the accuracy of the prediction model, since this type of model makes use of the components names as the independent variables.

We extracted the defect data as accurately as possible. We mapped the defects found in the issue tracker to the actual version that consist the change. Where the change set number is not included, the defect is traced using the date of change and the content in the description. This as it may, we observe that some discrepancies may still exist in terms of some defect data being left not captured or capturing wrong data such as enhancement as defect. Some defects that are fixed are not captured in the issue tracker; therefore we relied on the description recorded by the developer that performed the change. Furthermore, for some releases, the volume of the defect dataset is rather small. We believe, the more the better for our analysis and modeling, although, the less the better for the software stakeholders.

6. Conclusion and Future work

In this study, we have empirically evaluated a kind of Smart Grid applications. Our result showed that reused components could be employed as good predictors of future defective components that depend on them. Thus in addition to saving quality assurance budget and testing efforts for system testers, this approach can be used by system developers as early warning indicator for new components that depend on highly defect-prone reusable components. A disadvantage of this modeling approach is that the model’s performance can decline as the internally reused components become more stable.

Clearly, the fact that some components failed as a result of reusing another component do not mean this reused component is problematic as shown in this study. We further clarified the notion of “good” and “bad” imports to be context dependent.

In the future, we would investigate how defective components can be associated to the actual reused component(s) that triggers this defect. The hypothesis is that by associating the components to the source of the defects, it is possible to improve the model and make it significantly reliable.
Furthermore, we would like to investigate similar and other types of Smart Grid applications further both in the open source community and commercial industry. Currently, we are studying the defect data of some commercial Smart Grid applications and for the future compare the results of this study.

In addition, we seek to understand the reasons for the defect in certain part of the Smart Grid applications. Some of the questions we will pursue are for example: Is it due to the kind of architecture employed? Does it have relationship with the evolution of requirements, especially that the requirement is changing and not fully known at the moment? Are there design variables as shown in other empirical studies [7, 12-18] that could show correlational relationships? Is our result domain specific or programming language specific, for example across different frameworks (e.g., .NET or Java)?

Finally, we like to also explore how this approach can be combined with other types of prediction modeling (e.g. using structural properties) to produce a competitive defect prediction models. We will investigate the above across many Smart Grid systems and different domains.

References


