Prest: An Intelligent Software Metrics Extraction, Analysis and Defect Prediction Tool

Ekrem Kocagüneli¹, Ayşe Tosun¹, Ayşe Bener¹, Burak Turhan², Bora Çağlayan¹

¹Software Research Laboratory (Softlab), Computer Engineering Department, Boğaziçi University, Turkey
²National Research Council (NRC), Canada

ekrem.kocaguneli@boun.edu.tr, ayse.tosun@boun.edu.tr, bener@boun.edu.tr, Burak.Turhan@nrc-cnrc.gc.ca, bora.caglayan@boun.edu.tr

Abstract

Test managers use intelligent predictors to increase testing efficiency and to decide on when to stop testing. However, those predictors would be impractical to use in an industry setting, unless measurement and prediction processes are automated. Prest as an open source tool aims to address this problem. Compared to other open source prediction and analysis tools Prest is unique that it collects source code metrics and call graphs in 5 different programming languages, and performs learning based defect prediction and analysis. So far Prest in real life industry projects helped companies to achieve an average of 32% efficiency increase in testing effort.

1. Introduction

The role of software measurement becomes increasingly important to understand and control mature software development practices and products [1]. Software measurement helps to evaluate the software quality by measuring error-proneness of software modules, since residual defects in the software affects the final quality. However, measurement programs cannot be easily employed in software companies [2]. There has to be a tool support to analyze the quality of the software using various source code metrics [3,7]. Many researchers also have been working on building predictive models: defect prediction, cost/effort estimation. These models need raw data (i.e. regular measurement of software attributes) [4,26,24,25,27,28]. These research have significant implications in practice as well. Predictive models support practitioners to take critical decisions under uncertainty. Automated tools help researchers and practitioners to measure software artifacts seamless to coders [4,24,25,26,27,28]. Current software development environment, on the other hand, is complex such that multiple platforms (i.e. hardware and software) as well as multiple programming languages have to co-exist. Therefore any automated code measurement and analysis tool should address the issue of heterogeneity in software systems.

There exist several measurement and analysis tools, which are provided either as commercial of-the-shelf (COTS) [5, 22, 23] or as open source tools [6, 15, 19, 20, 21]. There are COTS tools, which provide extensive set of metrics and functionalities; however, they are not always affordable. Furthermore, their output formats cannot be easily integrated with other measurement and analysis tools. Open source tools, on the other hand, are easily accessible and their functionalities may be tailored to meet specific needs. However, open source tools have certain deficiencies: a) they can extract only a limited number of static code attributes from a limited number of programming languages, b) they do not include a learning based prediction support and c) they lack multiple output formats [8].

In this paper, we introduce an intelligent open source, software metrics extraction, analysis and defect prediction tool, called Prest [16]. The need for Prest has emerged during our collaborative research with industry partners from various domains (i.e. telecommunication [13], embedded systems [11] and healthcare [12]) over the past four years. Our aim in developing Prest was to extract static code attributes from software programs and build a learning-based defect predictor, which would highlight defect-prone parts of new projects, using code attributes and defect data from past projects. Prest is capable of extracting 28 static code attributes and generating call graphs by using five different language parsers. It also provides output in various formats that are compatible with popular toolkits like Weka [10]. Our industry partners have been using Prest for two years. The project
managers have been able to detect problems in coding practices and testing, and they take corrective actions on a timely manner.

2. Functionality

Prest is developed as a one-stop-shop tool that is basically capable of:

- Extracting common static code metrics from C, C++, Java, JSP and PL/SQL languages
- Presenting output via GUI components and in *.xml, *.csv, *.xls and *.arff file formats
- Generating call graphs in class and method level
- Defining new metrics or thresholds on extracted metrics
- Applying machine learning methods for analysis and defect prediction.

Each of these functionalities will be further described using a sample code (Figure 1). We also placed the sample code of Figure 1 and an executable jar of Prest in the Prest repository [16] for self trial. More complex analysis including defect prediction will be provided as a demo in Section 4.

2.1. Parsing and saving a project

Prest can parse all files that are written in different programming languages by using different parsers at the same time. Similar tools, on the other hand, can parse only one language at a time while ignoring other files. Once a project, such as the sample code in Figure 1, is parsed, the metrics are presented via GUI components in a structured manner and outputs are placed under the related project folder within the repository. The outputs are presented in several formats: *.csv, *.xls,*.arff and *.xml. In Figure 2, only one metric, i.e. cyclomatic density, is presented as the static code attributes that can be extracted by Prest, since we have page limitations. However, Table 1 provides full set of attributes.

Figure 1. Sample code

```java
public class Trial1 {
    public void trial1Func1()
    { trial1Func2(); }
    public void trial1Func2()
    { System.out.println("Trial1 class, function #2"); }
}

public class Trial2 {
    public static void main(String args[]) {
        Trial1 starter = new Trial1();
        Trial2 testClass = new Trial2();
        starter.trial1Func1();
        testClass.trial1Func2();
    }
}
```

Figure 2. GUI Overview of Prest

2.2. Call graph generation

Prest introduces a new and simple call graph feature for all supported languages. It extracts this information to better illustrate dependencies between functions/classes and the complexity of software systems. Basically, a function call graph represents the encodings of caller-callee relations between functions in a structured manner (Figure 3). Using Prest, each function in Figure 3 is treated as a potential caller and a unique ID is assigned to each function. Therefore, all the functions are listed under the column CALLER NAME and their ID's are listed under the column CALLER ID in an excel file. The second CALLEE ID column keeps the ID's of the called function(s) that were called by the caller function. Generated call graph matrix of Prest can be seen in Figure 4.

Figure 3. Function calls of sample code

<table>
<thead>
<tr>
<th>CALLER NAME</th>
<th>CALLER ID</th>
<th>CALLEE ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trial1.trial1Func1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Trial1.trial1Func2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Trial2.main</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>Trial2.trial2Func2</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>Trial2.trial2Func1</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4. Call graph matrix of sample code
2.3. Data Analysis and Prediction

Data analysis and prediction are particular features of Prest, which provide analysis and prediction via Naïve Bayes (NB) and Decision Tree (DT) algorithms. Given actual defect data of a project, in which bugs are matched with functions; Prest can analyze the given data via Naïve Bayes and Decision Tree algorithms and make predictions for a future release, which has yet not been tested. Then, it pinpoints defect-prone modules to increase the testing efficiency considerably. This feature has drastically helped our industry partners by reducing the testing effort by 32\% [11]. Its architecture and a detailed tutorial will be explained in Section 3, 4.

2.4. Threshold and New Metric Definition

Certain values of metrics or a combination of those may be indicator of error proneness. Prest provides users the ability to define certain conditions (thresholds) on the extracted metrics and apply color coding according to user-defined thresholds, i.e. metrics of defect-prone modules are colored with red on the GUI, whereas the defect-free ones are painted as green. Furthermore, Prest lets users to define new metrics by combining existing metrics via mathematical operators. In Figure 5, definition of a new metric using “/ DIVIDE” operator, cyclomatic_complexity and lines_of_code metrics is illustrated.

Figure 5. Defining a new metric

3. System Architecture

Prest architecture has four main components: Language parser, metric extractor, analysis and prediction component and GUI components.

3.1. Language Parser

A parser is responsible for parsing code into tokens depending on its type such as operand, operator etc. Currently, Prest consists of C, C++, Java, JSP and PL/SQL parsers.

3.2. Metrics Extractor

Once the language parser is done with parsing the code into tokens, the metric extraction component starts to execute and it produces logical results depending on the output of the language parser. Those logical results are used to calculate static code metrics, listed in Table 1. Prest collects 28 static code attributes (Table 1) and none of the other open source metrics extraction tools [18] were able to extract all metrics.

Table 1. Static code metrics extracted by Prest

<table>
<thead>
<tr>
<th>Metric Name</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total loc</td>
<td>Blank LOC</td>
</tr>
<tr>
<td>Comment LOC</td>
<td>Code Comment LOC</td>
</tr>
<tr>
<td>Executable LOC</td>
<td>Unique Operands</td>
</tr>
<tr>
<td>Total Operands</td>
<td>Total Operators</td>
</tr>
<tr>
<td>Halstead Vocabulary</td>
<td>Halstead Length</td>
</tr>
<tr>
<td>Halstead Volume</td>
<td>Halsted Level</td>
</tr>
<tr>
<td>Halstead Difficulty</td>
<td>Halstead Effort</td>
</tr>
<tr>
<td>Halstead Error</td>
<td>Halstead Time</td>
</tr>
<tr>
<td>Branch Count</td>
<td>Decision Count</td>
</tr>
<tr>
<td>Call Pairs</td>
<td>Condition Count</td>
</tr>
<tr>
<td>Multiple Condition Count</td>
<td>Cyclomatic Density</td>
</tr>
<tr>
<td>Cyclomatic Complexity</td>
<td>Decision Density</td>
</tr>
<tr>
<td>Design Complexity</td>
<td>Design Density</td>
</tr>
<tr>
<td>Normalized Cyclomatic Complexity</td>
<td>Formal Parameters</td>
</tr>
</tbody>
</table>

3.3. Analysis and Prediction Component

Analysis and prediction component of Prest significantly differentiates itself from similar open source tools, since none of them provides a learning based defect prediction component [18]. Unlike other open source metric extraction tools, Prest can perform analysis and predictions regarding the defect-proneness of software by utilizing machine learning methods. We have benefited from Weka libraries [10] to implement two classifiers, Naïve Bayes and Decision Tree, for this component. However, new methods may be included either by implementing it from scratch or by calling Weka libraries.

3.4. GUI Component

GUI component is responsible for interacting with the user and presenting the results. We paid particular
attention to GUI component and analyzed various tools such as Eclipse [6], Predictive [5] and WEKA [10], before designing it. We aimed to keep the usage simple, while providing full range of features, such as project repository, easy switch between metric extraction and analysis tabs, defining thresholds on static code metrics, filtering results depending on defined thresholds, applying color codes, and defining new metrics.

4. Demo

In Section 2, we have analyzed a sample Java code to discover the functionalities of Prest. In this section, we have analyzed a large software system from one of our industry partners. This software system has been implemented in Java and JSP languages. We took two versions of the same system (version 11 for training and 12 for testing) and extracted static code attributes from both Java and JSP files with Prest. Then, we matched actual defect data with the files whose static code attributes were extracted by Prest and fed them to analysis and prediction component. We have used Naïve Bayes classifier to predict defect-prone files in version 12. Finally, we have measured the performance of the prediction component of Prest when only Java files, only JSP files and both of them are used. In Figure 6, probability of detection rates (pd) and the balance rate (bal) have been increased when both Java and JSP files are used. Furthermore, probability of false alarm rates (pf) has been decreased.

Those results have been encouraging in the sense that extracting static code attributes from all the languages of a software project can increase the prediction performance. In addition, Prest, as a single tool, is able to conduct a thorough analysis in large software systems, thereby reducing the need for multiple tools for different languages and machine learning tools.

5. Development Methodology

Prest has been developed by MS and PhD students in SoftLab during the last three years. At various lengths of involvement (from 6 months to three years), a total of 12 students and a faculty member worked as the developers and designers of Prest. We used a formal waterfall approach where we took the requirements from our industry partners, reviewed them and used existing tools. Then, we designed the architecture, coded Prest, and conducted alpha and beta tests with our industry partners. Also, a senior architect has been guiding us for the current and future architecture of the tool. All development stages are well documented and we have used a versioning system as well as an automated bug tracking system. Current members of SoftLab carry out implementation of new parsers and they provide maintenance of Prest.

6. Current Usage & Benefits

Early versions of Prest were used by a local white-goods manufacturer, who wanted to measure code quality to reduce defect rates and to effectively manage their testing resources [11]. Using Prest, we collected static code metrics attributes from C codes at function level. Then, we analyzed the defect-prone parts of the software using data analysis component of Prest and found that testing effort could be reduced by 32% while catching 76% of defective modules [11].

Recently, we have conducted a metrics program in a large telecommunication software system [13]. In this project, we collected static code metrics with Prest in Java source file-level. Then, we matched those files with actual defect data and used Naïve Bayes classifier to predict defective files of the software. We have also used call graph information in method level and applied the Naïve Bayes classifier to predict defect-prone files in the system. Results show that prediction model in Prest has been capable of detecting 84% of defective files by inspecting only 31% of the code [13].

In addition to our local industry partners, currently Prest has been in use in a multi-national company in the UK. Since Prest [16] is designed as an open source tool, it is available via Google Code [19] to review, download or further develop and integrate.

7. Support

The development team of Prest provides support to users [14]. Once a development activity is performed and a stable version is elicited new code is committed to the Prest repository in Google Code [16]. Therefore,
the code that users can access is always the latest stable version of Prest. Any failure or problem in the system can be directly entered into the issue management system of Google Code in order to track the status of each problem on the web.

8. Related Work

There exist a considerable number of software metrics tools available either as open source [6, 15, 19, 20, 21] or as commercial [5, 22, 23]. Since Prest is developed as an open source tool, we focus on non-commercial tools for comparison of Prest and other tools. We acknowledge that there is no ultimate criterion to compare different tools and conclude that one is certainly better than the other. However, a set of criteria may be defined while assessing different metric tools: Number of languages that are supported, number and nature of metrics extracted, type of output formats and analysis and prediction components. Those functionalities are also examined by other researchers [7, 18]. Thus, they are also critical for our future extensions in Prest. We have presented this comparison with CCCC [19], Chidamber-Kemerer Java Metrics [20], Dependency Finder [21], Eclipse Metrics Plug-in version 1.3.6 [6] and CyVis[15] tools in Table 2. From Table 2, we can see that Prest is more extensive than other open source tools with respect to languages it parses, number of extracted metrics, output formats and its analysis and prediction component. However, this does not make Prest the finest and the ultimate tool, since there has been significant effort behind each tool and we still lack some properties such as simple and precise graphical representation of dependencies in Eclipse plug-in or saving extracted metrics in an html file. Nevertheless, we have managed to provide an all-in-one tool for software practitioners by saving their time and effort for searching multiple tools for various needs and dealing with various output formats. Moreover, we have benefitted from Prest in our research studies by extracting static code attributes and doing predictions for all experiment settings.

9. Conclusion and Future Work

Prest has been in use in three large software systems (locally and internationally). It has also been used in various SoftLab empirical research studies at different companies [11, 12, 13]. Prest in practice with its prediction capability has so far successfully guided project managers to take decisions under uncertainty and has considerably increased testing efficiency.
Going forward, Prest will be constantly adding new parsers as well as more learning algorithms. Currently, we are in the process of migrating Prest tool to cloud computing in order to serve larger communities better, to share data and foster reproduction of empirical experiments.

10. Acknowledgment

This research is funded in part by Tubitak EEEAG108E014. We would also extend our gratitude to Mr. Turgay Aytaç, senior architect, for his guidance as well as A.D.Oral, E.G. Isik, C. Gebi, H. Izmirlioglu, O. Bozcan and S. Karagulle for their efforts in the development of this tool.

11. References

[14] Software Research Laboratory (Softlab), available at www.softlab.boun.edu.tr