Predicting Defects with Program Dependencies

Thomas Zimmermann
Microsoft Research
Redmond, WA, USA
tz@acm.org

Nachiappan Nagappan
Microsoft Research
Redmond, WA, USA
nachin@microsoft.com

Abstract

Software development is a complex and error-prone task. An important factor during the development of complex systems is the understanding of the dependencies that exist between different pieces of the code. In this paper, we show that for Windows Server 2003 dependency data can predict the defect-proneness of software elements. Since most dependencies of a component are already known in the design phase, our prediction models can support design decisions.

1. Introduction

In any large-scale software development effort, teams work independently on different components in the architecture. Even though teams work independently, software dependencies will always exist between components and they are often divided across development teams. Therefore developers need to understand existing dependencies and how it affects (or should affect) their development process. There has been scant empirical evidence of this very common problem in the software development industry.

In this paper, we replicate a study by Schröter et al. [9] and show that the dependency relationships among the binaries of Windows Server 2003 can predict the defect-proneness of binaries.[1] Since many dependencies are decided during design or early in the implementation phase, such prediction models can be used to estimate the risk of failure and select between design alternatives. Our hypotheses are the following:

H1: Considering the dependencies of a single binary (which we call *problem domain*), we can predict its defect-proneness.

H2: Considering the dependencies between all binaries, we can predict the most defect-prone binaries.

[1] Defect-proneness is the probability that a particular software element has a defect that will lead to a failure.

2. Related work

This work is a replication of a study by Schröter et al. [9] who showed that import dependencies can predict defects for Eclipse. Unlike Schröter et al. who considered only import dependencies, we analyze a comprehensive set of dependencies, which includes call dependencies, data dependencies, and Windows specific dependencies such as shared registry entries. By replicating the study on closed source, we contribute towards building an empirical body of knowledge in the field of defect prediction by showing how dependencies predict post-release defects.

Earlier work on dependencies at Microsoft [6] showed that code churn and dependencies can be used as efficient indicators of post-release defects. Other work at Microsoft predicted defects for subsystem with the complexity of dependency graphs [12] or for binaries with network analysis on dependency graphs [11]. Compared to this work, we study the actual dependencies and not metrics computed from dependency data.

Most research on fault-proneness captures software complexity with metrics and builds models that relate these metrics to fault-proneness. For an overview of these studies, we refer to a recent survey by Catal and Diri [3]. Nagappan et al. [7] compared five large subsystems of Microsoft Windows and found that for each subsystem, there were metrics that worked reasonably well, but that no single metric worked well for every subsystem to predict failures. Nagappan's result further demonstrates the importance of replication.

3. Data collection

The data collection consists of two parts: first, we compute dependency data for Windows Server 2003 and next, we collect all defects reported within six months after the release point. All data is collected on binary level, as this unit is typically used for program analysis within Microsoft. Here binary refers to the

executable files (COM, EXE, etc.) and dynamic-link files (DLL) that are shipped with Windows. Binaries are assembled from several source files and typically form a logical unit, e.g., user32.dll provides programs with functionality to implement graphical user interfaces. Our case study involved analysis of 2252 binaries of Windows Server 2003 including their dependencies.

3.1. Program Dependencies

A program dependency is a directed relationship between two pieces of code (such as expressions or methods). There exist different kinds of dependencies: *data dependencies* between the definition and use of values and *call dependencies* between the declaration of functions and the sites where they are called. Microsoft has an automated tool called MaX [10] that tracks dependency information at the function level, including calls, imports, exports, RPC, COM, and access to the Registry. MaX generates a system-wide dependency graph from both native x86 and .NET managed binaries. Within Microsoft, MaX is used for change impact analysis and for integration testing [10].

For our analysis, we use MaX to generate a system-wide dependency graph at the binary level. Formally, we define a dependency graph as a *directed* graph $G = (V, E)$ where V is a set of nodes (=binaries) and $E \subseteq V \times V$ is a set of edges (=dependencies). In the dependency graph, an edge (A, B) means that binary A depends on binary B. Note that even though the graph allows self-edges, i.e., a binary can depend on itself, we ignored self dependencies for the experiments in this paper. From the graph, we get for every binary B a list of dependencies $[X_1, \ldots, X_n]$. This list implicitly describes what we call the *problem domain* of a binary.

3.2. Post-Release Defects

Microsoft records all problems that are reported for Windows Server 2003 in a database. In this study, we investigate post-release defects—that is, defects leading to failures that occurred in the field within six months after the initial release of Windows Server 2003. We collect all problem reports that were classified as non-trivial (in contrast to enhancement requests) and for which the problem was fixed in a later product update. The location of the fix gives us the location of the post-release defect.

Since each post-release defect in our dataset was uncovered by at least one post-release failure, predicting the likelihood of a post-release defect in a binary is equivalent to predicting the likelihood of at least one post-release failure associated with this binary.

4. Predicting defects

In this section, we demonstrate that by just using the dependencies $[X_1, \ldots, X_n]$of a binary B, we can make predictions about the presence of defects in B. In particular, we address two problems:

1. **Classification.** Can we predict which binaries will have defects?
2. **Ranking.** Can we predict which binaries will have the most defects?

Both are important questions for resource allocation. During testing, software managers want to focus on the parts that have defects, preferably the parts that have most of them. Furthermore, since many dependencies are known early in the development process and our models do not require the presence of source code, we can make early predictions—even at design time. This can be help developers to make better design decisions.

For example, assume that a developer can choose between two designs where a binary either depends on m.dll, i.dll, and k.dll (Design 1) or on i.dll and e.exe (Design 2). To make a better decision about which design to choose, the developer would predict for each design the likelihood of defects. In the example, Design 2 would be predicted as defect-prone, while Design 1 would be predicted as defect-free and thus is the primary choice. If for some reason, Design 1 has to be selected, the developers would still be aware of the increased risk of defects. In addition, by considering the weights of the input features, they can identify which dependency is most likely to cause defects.

4.1. Experimental Setup

For the experiments in this section we built prediction models that take the targets of the dependencies $[X_1, \ldots, X_n]$ of a binary B as input, and return as output a *classification* of whether the binary would contain a defect or not (1 or 0) or a probability of its defect-proneness (*ranking*).

Because of the high dimensionality of the input data (every dependency target is considered as one dimension), classical regression models would be doomed to overfit the data. Instead we rely on Support Vector Machines (SVMs) [2, 8]. This decision is also supported by Schröter et al. [9]. On a similar dataset, they also used linear regression, regression trees, and ridge regression, but achieved the best results for SVMs, possibly because SVMs are less prone to overfitting.

For our experiments we used the SVM implementation by John Platt [8]. We performed several initial studies to choose the best SVM configuration. We tried linear, polynomial, and radial basis function (RBF) kernels. We also fit sigmoids into SVMs by using 70/30 splits and three-fold cross-validation. To our surprise linear kernels yielded the best results, which is why we report results with linear kernels in this paper. Training and testing a linear SVM took on average less than a second per experiment (measured on a 3.2 GHz Pentium 4 HT with 1 GB RAM running Windows XP).

To evaluate the predictive power of our dependency data, we use data splitting [5]. That is, we randomly pick two-thirds of all binaries to build a prediction model and use the remaining one-third to measure the efficacy of the built model. Overall we performed 1000 random splits experiments to ensure the stability and repeatability of our results. Throughout the experiments, the input dimensions remained unchanged as the 2252 dependency targets (binaries) of Windows Server 2003 to ensure comparability across splits.

4.2. Classification

To compare *predicted* against *observed* behavior of elements, we build a contingency table and observe two kinds of errors: false negatives and false positives.

		Observed	
		Has defects	Defect-free
Predicted	Has defects	A	B
	Defect-free	C	D

- **False negatives** are binaries that were classified (using a binary classification of defect-free or not) to be defect-free but actually had defects, i.e., they were missed by the prediction model. The number of false negatives corresponds to Cell *C* in the above table. The *recall A/(A+C)* measures how many of the binaries with defects were actually classified correctly as defect-prone.
- **False positives** are binaries that were classified as to have defects but had no defects, i.e., they were flagged incorrectly. In table above the number of false positives is *B*. The *precision A/(A+B)* measures how many of the binaries predicted as to have defects actually had defects.

Both precision and recall should be as close to the value 1 as possible (=no false negatives and no false positives). However, such values are difficult to realize since techniques that reduce the number of false positives often increase the number of false negatives and vice versa.

In the 1000 random splits, the precision ranged between 0.58 and 0.73 with a median of 0.64—roughly two out of three binaries classified to have defects, actually have defects. The recall ranged between 0.66 and 0.81 with a median of 0.75—three out of four binaries that were observed to have defects, are found by our model. Other models for the Windows Server 2003 dataset used complexity metrics and logistic regression with principal component analysis to classify binaries as defect-prone [11]. They had a comparable precision (around 0.7), while having a lower recall (close to 0.6 vs. 0.75 for dependency data). Using SVMs and a radial basis function as kernel on import relationships, Schröter et al. [9] obtained similar precision values (0.67) and recall values (0.69) for the Eclipse project than we did with SVMs for Windows Server 2003.

4.3. Ranking

In order to assess the rankings predicted by our SVMs we use the Spearman's rank correlation coefficient. The Spearman rank correlation is a commonly-used robust correlation technique [4] because it can be applied even when the association between elements is non-linear. In contrast, Pearson correlation requires data to be distributed normally and the association between elements to be linear. Positive correlations result in a value of 1 and negative correlations in -1. For no correlation between the elements, the value is 0. In particular, a high positive value for Spearman means that two rankings are similar (or identical for a value of 1).

For the 1000 random split experiments the Spearman correlation values between the predicted probability of defects (defect-proneness) and actual observed defects ranged between 0.45 and 0.60 with a median of 0.52. All correlations were consistent across the random splits and significant at 99% confidence. These statistically significant and positive correlations between the predicted and the observed rankings indicate the efficacy of the predicted rankings. Put another way, with an increase in the predicted rank there is an increase in the actual rank and vice-versa, thereby quantifying the sensitivity of the predictions. Other models used complexity metrics and linear regression with principal component analysis to predict the most defect-prone binaries for Windows Server 2003. They had Spearman correlation values below 0.5 vs. between 0.45 and 0.6 for dependency data. Using SVMs and a radial basis function as kernel on import relationships, Schröter et al. [9] also observed a positive, though lower, Spearman correlation (0.30) for Eclipse data.

We showed that by using solely dependency data, we can make appropriate predictions for both classification and ranking. This supports our first and second hypothesis that *considering all dependencies of a single binary, we can predict its defect-proneness (H1)* and *considering the dependencies between all binaries, we can predict the most defect-prone binaries (H2).*

5. Threats to validity

As stated by Basili et al., drawing general conclusions from empirical studies in software engineering is difficult because any process depends on a potentially large number of relevant context variables. For this reason, we cannot assume a priori that the results of a study generalize beyond the specific environment in which it was conducted [1]. However, we are confident that dependency data also has predictive power in other projects, especially since similar results were found for the Eclipse project [9].

The analysis in this paper was performed as a post-mortem operation, i.e., all the data points are from the same version. It is possible that these results might be valid only for the current version. We intend to address this issue in our future work by incorporating our approach into the development of the next generation of Windows operating systems.

6. Conclusion and consequences

Software development benefits from early estimates regarding the quality of products since quality assurance can be allocated accordingly. In this paper, we replicated a study by Schröter et al. [9] and studied the *problem domain* as implicitly defined by program dependencies. Our findings on Windows Server 2003 show that the dependency information can assess the defect-proneness of a binary. Since many dependencies are decided during design or early in the implementation phase, one can use our prediction models to estimate defect-proneness and select between design alternatives. This also helps developers avoid creating dependencies on problem prone areas of the system and to explore design alternatives.

Acknowledgments. Many thanks to the reviewers for valuable and helpful suggestions on earlier revisions of this paper.

7. References

[1] V. R. Basili, F. Shull, and F. Lanubile, "Building Knowledge Through Families of Experiments," *IEEE Transactions on Software Engineering,* vol. 25, pp. 456 - 473, 1999.

[2] B. E. Boser, I. Guyon, and V. Vapnik, "A Training Algorithm for Optimal Margin Classifiers," in *Proceedings of the Fifth Annual ACM Conference on Computational Learning Theory (COLT 1992)*, Pittsburgh, PA, USA, 1992, pp. 144-152.

[3] C. Catal and B. Diri, "A Systematic Review of Software Fault Prediction Studies," *Expert Systems with Applications,* vol. 36, pp. 7346-7354, 2009.

[4] N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*, 2nd ed.: Course Technology, 1998.

[5] J. Munson and T. Khoshgoftaar, "The Detection of Fault-Prone Programs," *IEEE Transactions on Software Engineering,* vol. 18, pp. 423-433, 1992.

[6] N. Nagappan and T. Ball, "Using Software Dependencies and Churn Metrics to Predict Field Failures: An Empirical Case Study," in *International Symposium on Empirical Software Engineering and Measurement*, 2007, pp. 364-373.

[7] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *International Conference on Software Engineering*, 2006, pp. 452-461.

[8] J. C. Platt, "Fast Training of Support Vector Machines using Sequential Minimal Optimization," in *Advances in Kernel Methods - Support Vector Learning*, B. Schölkopf, C. J. C. Burges, and A. J. Smola, Eds.: MIT Press, 1998, pp. 185-208.

[9] A. Schröter, T. Zimmermann, and A. Zeller, "Predicting Component Failures at Design Time," in *International Symposium on Empirical Software Engineering* Rio de Janeiro, Brazil, 2006.

[10] A. Srivastava, T. J., and C. Schertz, "Efficient Integration Testing using Dependency Analysis," Microsoft Research-Technical Report, MSR-TR-2005-94, 2005.

[11] T. Zimmermann and N. Nagappan, "Predicting Defects using Network Analysis on Dependency Graphs," in *International Conference on Software Engineering* Leipzig, Germany, 2008, pp. 531-540.

[12] T. Zimmermann and N. Nagappan, "Predicting Subsystem Defects using Dependency Graph Complexities," in *International Symposium on Software Reliability Engineering* Trollhättan, Sweden, 2007.