# Predicting Subsystem Failures using Dependency Graph Complexities

Thomas Zimmermann[+]
*University of Calgary*
*Calgary, Alberta, Canada*
*tz@acm.org*

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

## Abstract

*In any software project, developers need to be aware of existing dependencies and how they affect their system. We investigated the architecture and dependencies of Windows Server 2003 to show how to use the complexity of a subsystem's dependency graph to predict the number of failures at statistically significant levels. Such estimations can help to allocate software quality resources to the parts of a product that need it most, and as early as possible.*

## 1. Introduction

Software dependencies are often spread across binaries that are developed by different teams. These teams have to be aware of existing dependencies and how they affect (or should affect) their development process. More dependencies generally result in more complex code that is harder to manage. There has been scant empirical evidence of this very common problem in the software development industry. Further, with recent trends in the global nature of software development teams, it becomes more crucial to understand software dependencies to make sound design and business decisions.

In this paper, we use dependency and system architecture data to identify their relation to failures. More specifically, we focus on the level of subsystems (as defined by the system's architecture) and compute the complexity of the subsystem's dependency graphs using concepts adapted from classical graph theory. We hypothesize that these complexities correlate with failures—as code complexity metrics do. (We use the IEEE definition of a failure as the *inability of a system or component to perform its required functions within specified performance requirements* [13].) We also show how to use such graph complexities adapted from graph theory to predict the number of failures. Having reliable predictions of failures supports the following two tasks in software engineering.

**Resource allocation.** Software quality assurance consumes a considerable effort in any large-scale software development. To raise the effectiveness of this effort, it is wise to spend more attention on the components that are more likely to fail and need quality assurance most.

**Decision making.** Predictions on the number of failures can also support other decisions such as choosing the correct requirements or design. However, for this case, one needs early indicators of failures. Many dependencies are known early in the development, while code metrics cannot be used until implementation begins or until a substantial part of the code is written.

We studied the dependency data of the Windows Server 2003 operating system which is a large commercial software project, with an analyzed code base of 28.3 MLOC comprising 2252 compiled binaries.

The outline of this paper is as follows: We motivate the relevance of dependencies with two observations and state our research hypotheses in Section 2. Next, we summarize related work in Section 3. In Section 4 we discuss our data collection, i.e., how we obtained the architecture and dependency data and how we computed complexity. Section 5 presents several experiments to support our hypothesis. We conclude the paper with our plans for future work in Section 6.

## 2. Motivation

When we analyzed failure data and dependency graphs for the binaries of Windows Server 2003, we made the following observations.

**Cycles had on average twice as many failures.** We investigated whether dependency cycles have impact on failures. A simple example for a dependency cycle is a mutual dependency, i.e., binaries X and Y depend on each other; for this experiment, we considered cycles of any size, but ignored self-cycles such as X

[+] Tom Zimmermann was an intern with the Software Reliability Research Group, Microsoft Research in the summer of 2006 when this work was carried out.

depends on X. Based on whether binaries are part of a cycle, we divided them into groups. Binaries that were part of *cycles had on average twice as many failures* as the other binaries, at a significance of 99%.

**The larger a clique the more failure-prone are its binaries.** A clique is a set of binaries for which between every pair of binaries (X, Y) a dependency exists—we neglect the direction, i.e., it doesn't matter whether X depends on Y, Y on X, or both. Figure 1 shows an example for an undirected clique; a clique is maximal if no other binary can be added without losing the clique property. We enumerated all *maximal undirected cliques* in the dependency graph of Windows Server 2003 with the Bron-Kerbosch algorithm [7]. Next we grouped the cliques by size and computed the average number of failures per binary. Figure 2 shows the results, including a 95% confidence interval of the average. We can observe that the average *number of failures increases with the size of the clique a binary resides in*. Put another way, binaries that are part of more complex areas (cliques) have more failures.

These observations suggest that certain properties of dependency graphs (such as the presence of cycles and cliques) correlate with failures. In this paper, we will therefore investigate whether dependency data predicts failures. Rather than using code complexity metrics for individual binaries, we will compute complexity measures for the dependency graphs of whole subsystems. By using graph theoretic properties we can take the interaction between binaries into account. Formally, our research hypotheses are the following.

| | |
|---|---|
| **H1** | *For subsystems, the complexity of dependency graphs positively correlates with the number of post-release failures*—an increase in complexity is accompanied by an increase in failures. |
| **H2** | *The complexity of dependency graphs can predict the number of post-release failures.* |
| **H3** | *The quality of the predictions improves when they are made for subsystems that are higher in the system's architecture.* |

## 3. Related work

In this section we discuss related work; it falls into three categories: software architecture and dependencies, complexity metrics, and historical data.

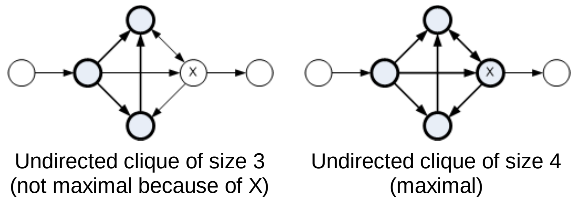

X
X
Undirected clique of size 3
(not maximal because of X)
Undirected clique of size 4
(maximal)

**Figure 1. Undirected cliques.**

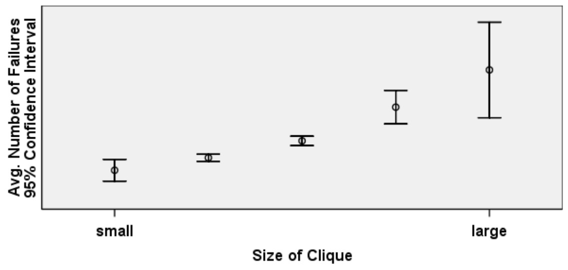


**Figure 2. Failure-proneness of cliques.**

### 3.1. Architecture and dependencies

There are many different ways to describe software architecture: relationships and properties of architectural elements [2, 26], *architectural style* which means a set of design rules combined with local or global constraints [29], and of course architectural description languages [17]. Pinzger et al. [27] integrated information on the evolution of software architecture from the source basis of a project and from the release history data such as modification and problem reports. The integrated architectural views show intended and unintended couplings between architectural elements. This information can be used to highlight to software engineers the locations in the system that may be critical for on-going and future maintenance activities.

Schröter et al. [28] showed that the actual import dependencies (not just the count) can predict defects. Earlier work on at Microsoft [21] showed that code churn and dependencies can be used as efficient indicators of post-release failures. The basic idea is that churn often will propagate across dependencies. Suppose that component A has many dependencies on component B. If the code of component B changes (churns) a lot between versions, we may expect that component A will need to undergo a certain amount of churn in order to keep in synch with component B. Together, a high degree of dependence plus churn can cause errors that will propagate through a system, reducing its reliability.

### 3.2. Complexity metrics

Typically, research on failure-proneness captures software complexity with metrics and builds models that relate these metrics to failure-proneness [9]. Basili et al. [3] were among the first to validate that OO metrics predict defect density. Subramanyam and Krishnan [31] presented a survey on eight more empirical studies, all showing that OO metrics are significantly associated with defects.

Our experiments focus on post-release failures since they matter for the end-users of a program. Only few studies addressed post-release failures: Binkley and Schach [5] developed a coupling metric and showed that it outperforms several other metrics; Ohlsson and Alberg [24] used metrics to predict modules that fail during operation. Additionally, within five Microsoft projects, Nagappan et al. [23] identified metrics that predict post-release failures and reported how to systematically build predictors for post-release failures from history. In contrast to their work, we develop new metrics on dependency data from a graph theoretic point of view.

### 3.3. Historical data

Several researchers used historical data for predicting defect density: Khoshgoftaar et al. [15] classified modules as defect-prone when the number of lines added or deleted exceeded a threshold. Graves et al. [12] used the sum of contributions to a module to predict defect density. Ostrand et al. [25] used historical data from up to 17 releases to predict the files with the highest defect density of the next release. Further, Mockus et al. [18] predicted the customer perceived quality using logistic regression for a commercial telecommunications system (of size seven million lines of code) by utilizing external factors like hardware configurations, software platforms, amount of usage and deployment issues. They observed an increase in probability of failure by twenty times by accounting for such measures in their prediction equations.

## 4. Data collection

In this section we explain how we collect hierarchy information and software dependencies and how we measure the complexity of subsystems. For our experiments we used the Windows Server 2003 operating system which is decomposed into a hierarchy of subsystems as shown in Figure 3. On the highest level are *areas* such as "Multimedia" or "Networking". Areas are further decomposed into *components* such as "Multimedia: DirectX" (DirectX is a Windows technology that enables higher performance in graphics and sound when users are playing games or watching video on their PC) and *subcomponents* such as "Multimedia: DirectX: Sound". On the lowest level are the *binaries* to which we can accurately map failures; we considered post-release failures because they matter most for end-users. Since failures are mapped to the level of binaries, we can aggregate the failure counts of the binaries of a subsystem *(areas, components, subcomponents*) to get its total subsystem failure count.

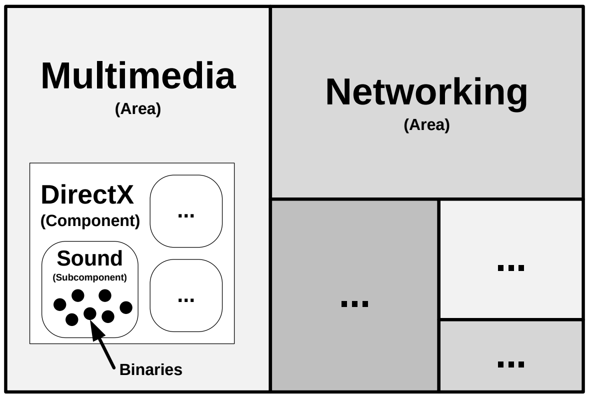


**Figure 3. Example architecture of Windows Server 2003**

We first generate a dependency graph for Windows Server 2003 at the binary level (Section 4.1). Then we divide this graph into different kinds of subgraphs using the area/component/subcomponent hierarchy (Section 4.2). For the subgraphs, we compute complexity measures (Section 4.3) which we finally use to predict failures for subsystems. We placed our analysis on the level of binaries for two reasons: (1) Binaries are easier to analyze since one is independent from the build process and other specialties such as preprocessors. (2) Defects were collected on binary level; mapping them back to source code is challenging and might distort our study.

### 4.1. Software dependencies

A software dependency is a directed relation 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 [30] that tracks dependency information at the function level, including calls, imports, exports, RPC, COM, and Registry accesses. MaX generates a system-wide dependency graph from both native x86 and .NET ma-

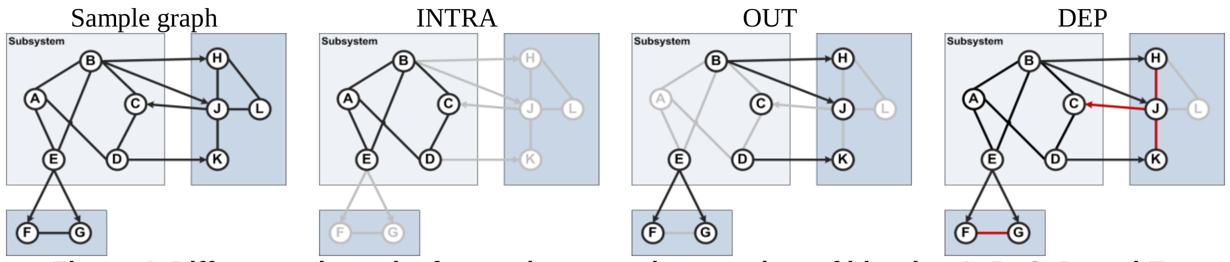


**Figure 4. Different subgraphs for a subsystem that consists of binaries A, B, C, D, and E: intra-dependency (INTRA), outgoing dependency (OUT), and combined dependency graph (DEP).**

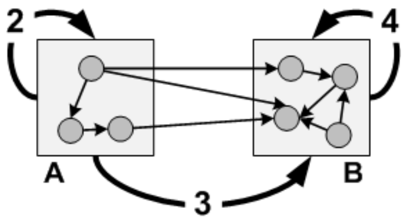


**Figure 5. Lifting up dependencies. The edges are labeled by the multiplicity of a dependency**

naged binaries. This graph can be viewed as the low-level architecture of Windows Server 2003. Within Microsoft, MaX is used for change impact analysis and for integration testing [30]. There are freely available tools like Dependency Finder or JDepend (for Java) and MakeDep (for C++) which can be used to repeat our study for other projects.

For our analysis, we use MaX to generate a system-wide dependency graph at the function level. Since we collect failure data for binaries, we lift this graph up to binary level in a separate post-processing step. Consider for example the dependency graph in Figure 5. Circles denote functions and boxes are binaries. Each thin edge corresponds to a dependency at function level. Lifting them up to binary level, there are two dependencies within A and four within B (represented by self-edges), as well as three dependencies where A depends on B. We refer to these numbers as *multiplicity* of a dependency/edge.

As a result of this lifting operation there may be several dependencies between a pair of binaries (like in Figure 5 between A and B), which results in several edges in the dependency graph. For our predictions, we will consider both regular graphs (where only one edge between two binaries is counted) and multigraphs (where every edge between two binaries is counted).

Formally (for our experiments), a dependency graph is a directed multigraph $G = (V, A)$ where

- $V$ is a set of nodes (binaries) and
- $A = (E, m)$ a multiset of edges (dependencies) for which $E \subseteq V \times V$ contains the actual edges and the function $m: E \to \mathbf{N}$ returns the multiplicity (count) of an edge.

The corresponding regular graph (without multiedges) is $G' = (V, A)$. We allow self-edges for both regular graphs and multigraphs.

## 4.2. Dependency subgraphs

We use hierarchy data from Windows Server 2003 to split the dependency graph $G=(V,A)$ into several subgraphs; for a subsystem that consists of binaries $B$, we compute the following subgraphs (see also Figure 4):

**Intra-dependencies (INTRA).** The subgraph $(V_{intra}, E_{intra})$ contains all *intra-dependencies*, i.e., dependencies $(u,v)$ that exist between two binaries $u,v \in B$ within the subsystem. This subgraph is induced by the set of binaries $B$ that are part of the subsystem.

$$V_{\text{intra}} = B$$
$$E_{\text{intra}} = \{(u,v) | (u,v) \in E, u \in B, v \in B\}$$
$$A_{\text{intra}} = (E_{\text{intra}}, m)$$

**Outgoing dependencies (OUT).** The subgraph $(V_{out}, E_{out})$ contains all *outgoing* inter-dependencies $(u,v)$ that connect the subsystem with other subsystems, i.e., $u \in B$, $v \notin B$. This subgraph is induced by the set of edges that represent outgoing dependencies. We focus on outgoing dependencies because they are the ones that can make code fail.

$$E_{\text{out}} = \{(u,v) | (u,v) \in E, u \in B, v \notin B\}$$
$$A_{\text{out}} = (E_{\text{out}}, m)$$
$$V_{\text{out}} = \{u | (u,v) \in E_{\text{out}}\} \cup \{v | (u,v) \in E_{\text{out}}\}$$

**Subsystem dependency graph (DEP).** The subgraph $(V_{dep}, E_{dep})$ combines the intra-dependencies and the outgoing dependencies subgraphs. Note that we additionally take edges between the neighbors of the subsystem into account.

$$V_{\text{dep}} = V_{\text{intra}} \cup V_{\text{out}}$$
$$E_{\text{out}} = \left\{(u,v) \middle| (u,v) \in E, u \in V_{\text{dep}}, v \in V_{\text{dep}}\right\}$$
$$A_{\text{out}} = (E_{\text{out}}, m)$$

**Table 1. Complexity for a multigraph G=(V,(E,m)) and its underlying graph G'=(V,E).** The set of *weakly* connected components is P; in(v) returns the ingoing and out(v) the outgoing edges of a node v.

| | Regular graph | Multigraph | Aggregation |
|---|---|---|---|
| **Number of NODES** | $\|V\|$ | $\|V\|$ | Not necessary |
| **Number of EDGES** | $\|E\|$ | $\sum_{e\in E} m(e)$ | Not necessary |
| **COMPLEXITY** | $\|E\| - \|V\| + \|P\|$ | $\sum_{e\in E} m(e) - \|V\| + \|P\|$ | Not necessary |
| **DENSITY** | $\frac{\|E\|}{\|V\| \cdot \|V\|}$ | $\frac{\sum_{e\in E} m(e)}{\|V\| \cdot \|V\|}$ | Not necessary |
| **DEGREE of node v** | $\|\mathrm{in}(v) \cup \mathrm{out}(v)\|$ | $\sum_{e\epsilon \|\mathrm{in}(v)\cup \mathrm{out}(v)\|} m(e)$ | Over nodes $v\epsilon V$ using min, max, avg. |
| **ECCENTRICITY of node v** | $\max\{\mathrm{dist}(v,w)\|w\epsilon V\}$ | $\max\{\mathrm{multidist}(v,w)\|w\epsilon V\}$ | Over nodes $v\epsilon V$ using min, max, avg. |
| **MULTIPLICITY of edge e** | 1 | $m(e)$ | Over edges $e\epsilon E$ using min, max, avg. |

Considering different subgraphs allows us to investigate the influence of internal vs. external dependencies on post-release defects. We compute the dependencies across all the three subsystem levels (area, component, and subcomponent).

## 4.3. Graph-Theoretic Complexity Measures

On the subgraphs defined in the previous section, we compute complexity measures which we will later use to predict post-release failures. The complexity measures are computed for both regular graph and multigraphs with the main difference being the number of edges $|E|$ and $\sum_{e\in E} m(e)$ respectively. Some of the measures are aggregated from values for nodes and edges by using minimum, maximum and average. The formulas are summarized in Table 1 and discussed below.

**Graph complexity.** Besides simple complexity measures such as the *number of nodes* or *number of edges*, we compute the *graph complexity* and the *density* of a graph [32]. Although the graph complexity was developed for graphs in general, it is well known in the software engineering community for its use on control flow graphs (McCabe's cyclomatic complexity).

**Degree-based complexity.** We measure the number of ingoing and outgoing edges (degree) of nodes and aggregate them by using minimum, maximum, and average. These values allow us to investigate whether the aggregated number of dependencies has an impact on failures.

**Distance-based complexity.** By using the Floyd-Warshall algorithm [8], we compute the shortest distance between all pairs of nodes. For regular graphs, the initial distance between two connected nodes is 1. For multigraphs, we assume that the higher the multiplicity of an edge *e*, the closer the incident nodes are to each other; thus we set the initial distance to *1/m(e)*. From the distances we compute the *eccentricity* of a node *v* which is the greatest distance between v and any other node. We aggregate all eccentricities with minimum (=radius), maximum (=diameter), and average. With distance-based complexities we can investigate if the propagation of dependencies has an impact on failures.

**Multiplicity-based complexity.** For multigraphs, we measure the minimum, maximum and average multiplicity of edges. This also allows us to investigate the relation between number of dependencies and failures.

## 5. Experimental analysis

In this section, we will support our hypotheses that complexity of dependency graphs predicts the number of failures for a subsystem, with several experiments. We carried out the experiments on three different architecture levels of Windows Server 2003: subcomponents, components, and areas. Most of this paper will focus on the *subcomponent* level: we start with a correlation analysis of complexity measures and number of failures (Section 5.1) and continue with building regression models for failure prediction (Section 5.2). Next, we summarize the results for the *component* and

**Table 2. Correlation between failures and complexity measures (on subcomponent level)**

| | | Pearson | | | | Spearman | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | INTRA | OUT | DEP | | INTRA | OUT | DEP | |
| NODES | | .325(*) | .497(*) | .501(*) | **O3** | .338(*) | .579(*) | .580(*) | **O3** |
| EDGES | | .321(*) | .454(*) | .485(*) | | .353(*) | .586(*) | .567(*) | |
| COMPLEXITY | | .319(*) | .322(*) | .481(*) | | .346(*) | .387(*) | .564(*) | |
| DENSITY | **O2** | -.312(*) | -.292(*) | -.418(*) | | -.294(*) | -.506(*) | -.519(*) | |
| DEGREE_MIN | | .168(*) | .054 | .014 | | .182(*) | .030 | .145(*) | |
| DEGREE_MAX | | .332(*) | .409(*) | .496(*) | | .347(*) | .533(*) | .569(*) | |
| DEGREE_AVG | | .386(*) | .377(*) | .366(*) | | .332(*) | .516(*) | .526(*) | |
| ECCENTRICITY_MIN | | .293(*) | .164(*) | .009 | | .314(*) | .305(*) | .079 | |
| ECCENTRICITY_MAX | | .307(*) | .201(*) | .094(*) | | .323(*) | .337(*) | .370(*) | |
| ECCENTRICITY_AVG | | .303(*) | .193(*) | .099(*) | | .317(*) | .471(*) | .527(*) | |
| MULTI_EDGES | **O4** | **.728(*)** | .432(*) | .393(*) | **O4** | **.667(*)** | .671(*) | .524(*) | |
| MULTI_COMPLEXITY | | **.728(*)** | .432(*) | .393(*) | | **.667(*)** | .671(*) | .524(*) | |
| MULTI_DENSITY | | .290(*) | .116(*) | -.108(*) | | .455(*) | .282(*) | -.138(*) | |
| MULTI_DEGREE_MIN | | .376(*) | .006 | .177(*) | | .296(*) | -.298(*) | .045 | |
| MULTI_DEGREE_MAX | | **.637(*)** | .395(*) | .356(*) | | **.643(*)** | .654(*) | .511(*) | |
| MULTI_DEGREE_AVG | | .538(*) | .247(*) | .148(*) | | .597(*) | .597(*) | .364(*) | |
| MULTI_MULTIPLICITY_MIN | | .300(*) | .005 | -.020 | | .201(*) | -.355(*) | -.328(*) | |
| MULTI_MULTIPLICITY_MAX | | **.640(*)** | .389(*) | .249(*) | | **.640(*)** | .634(*) | .418(*) | |
| MULTI_MULTIPLICITY_AVG | | .454(*) | .178(*) | .013 | | .571(*) | .505(*) | .102(*) | |
| MULTI_ECCENTRICITY_MIN | | .267(*) | .136(*) | -.010 | | .311(*) | .313(*) | .015 | |
| MULTI_ECCENTRICITY_MAX | | .267(*) | .141(*) | -.010 | | .312(*) | .346(*) | .060 | |
| MULTI_ECCENTRICITY_AVG | | .267(*) | .137(*) | -.010 | | .311(*) | .302(*) | .016 | |

*area* level and discuss the influence of granularity (Section 5.3). Finally, we present threats to validity.

## 5.1. Correlation analysis

In order to investigate our initial hypothesis H1, we determined the Pearson and Spearman rank correlation between the dependency graph complexities measures for each subcomponent (Sections 4.2 and 4.3) and its number of failures. For Pearson correlation to be applied the data requires a linear distribution, Spearman rank correlation can be applied even when the association between values is non-linear [11]. The closer the value of correlation is to –1 or +1, the higher two measures are correlated—positively for +1 and negatively for –1.

The results for subcomponent level of Windows Server 2003 are shown in Table 2. The table shows the complexity measures in the rows (Section 4.3) and the different kinds of dependency graphs in the columns (Section 4.2). Correlations that are significant at 0.99 are indicated with (*); note that the *Multi_Edges* and *Multi_Complexity* measures were strongly inter-correlated, which resulted in almost the same correlations with the number of failures. For space reasons we omit we the inter-correlations between the complexity measures; the correlation for the area and component level can be found in our technical report [33].

In Table 2 we can make the following observations.

O1 For most measures the correlations are significant (indicated by *) and positive. This means that with an increase of such measures there an increase in the number of failures, though at different levels of strength.

O2 The only notable negative correlation is for *Density*, which means that with an increase in the density of dependencies there is a decrease in the number of failures. This effect is strongest for DEP graphs, but vanishes when taking multiedges into account (*Multi_Density*).

O3 When we neglect multiplicity and consider *only presence of dependencies*, we obtain the highest correlations for subgraphs that additionally contain the neighborhood of a subsystem (DEP).

O4 When we take *multiplicity* of dependencies into account the correlations are highest for subgraphs that contain only dependencies within the subsystem (INTRA).

O5 The correlations were highest for *Multi_Edges*, and the inter-correlated *Multi_Complexity*, and for *Multi_Degree_Max* and *Multi_Multiplicity_Max* (highlighted in **bold**). All of these measures consider multiedges, suggesting that the number of dependencies matters and not just the presence.

To summarize we could observe significant correlations for most complexity measures, and most of them were positive and high (O1, O5). This confirms our initial hypothesis that *the complexity of dependency graphs positively correlates with the number of post-release failures (H1).* The only exception we observed was the density of a dependency graph (O2). This is surprising, especially since cliques tend to have a high failure-proneness (see Section 2) and a high density at the same time. One possible explanation for the poor correlation of density might be that normalizing the number of dependencies $|E|$ by the squared number of binaries $|V|\cdot|V|$ is too strong. This is supported by the *Degree_Avg* measure which normalizes $|E|$ only by $|V|$ and has a rather high positive correlation (up to 0.527 for Spearman).

The different results for complexity measures with and without multiplicity (O3 and O4), might suggest that one should consider both, the multiplicity of dependencies and the neighborhood of a subsystem—however, dependencies across subsystems should be weighted less. In our future work, we will investigate whether this actually holds true.

## 5.2. Regression analysis

So since complexity of dependency graphs correlates with post-release failures, can we use complexity to predict failures? To answer this question, we build multiple linear regression (MLR) models where the number of post-release failures forms the dependant variable and our complexity measures form the independent variables. We build separate models for every type of subgraph (INTRA, OUT, and DEP) and a combined model that uses all measures from Table 2 as independent variables (COMBINED). We carried out 24 experiments: one for each combination out of two kinds of regression (linear, logistic), three granularities (areas, components, subcomponents,) and four different sets of complexities (INTRA, OUT, DEP, COMBINED.

However, one difficulty associated with MLR is multicollinearity among the independent variables. Multicollinearity comes from inter-correlations such as between the aforementioned *Multi_Edges* and *Multi_-Complexity.* Inter-correlations can lead to an inflated variance in the estimation of the dependant variable. To overcome this problem, we use a standard statistical approach called *Principal Component Analysis* (PCA) [14]. With PCA, a small number of uncorrelated linear combinations of variables are selected for use in regression (linear or logistic). These combinations are independent and thus do not suffer from multicollinearity, while at the same time they account for as much sample variance as possible—for our experiments we selected principal components that account for a cumulative sample variance greater than 95%. We ended up with 5 principal components for INTRA, 7 for OUT, 6 for DEP, and 14 for the COMBINED set of measures (for the composition of components we refer to our technical report [33]). The principal components are then used as the independent variables.

To evaluate the predictive power of graph complexities we use a standard evaluation technique: *data splitting* [20]. 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. For every experiment, we performed 50 random splits to ensure the stability and repeatability of our results—in total we trained 1200 models. Whenever possible, we reused the random splits to facilitate comparison of results.

We measured the quality of *trained* models with:

- The **$R^2$ value** is the ratio of the regression sum of squares to the total sum of squares. It takes values between 0 and 1, with larger values indicating more variability explained by the model and less unexplained variation—a high $R^2$ value indicates good explanative power, but *not* predictive power.
- The **adjusted $R^2$ measure** also can be used to evaluate how well a model fits a given data set [1]. It explains for any bias in the $R^2$ measure by taking into account the degrees of freedom of the independent variables and the sample population. The adjusted $R^2$ tends to remain constant as the $R^2$ measure for large population samples.

Additionally, we performed **F-tests** on the regression models. Such tests measure the statistical significance of a model based on the null hypothesis that its regression coefficients are zero. In our case, every model was significant at 99%.

For *testing*, we measured the predictive power with the Pearson and Spearman correlation coefficients. The Spearman rank correlation is a commonly-used robust correlation technique [11] because it can be applied even when the association between elements is non-linear; the Pearson bivariate correlation requires the data to be distributed normally and the association between elements to be linear. For completeness we compute the Pearson correlations also. As before, the closer the value of a correlation is to –1 or +1, the higher two measures are correlated—in our case we are correlating the predicted number of failures with the actual number of failures (for MLR); and failure-proneness probabilities with actual number of failures (logistic regression), thus values close to 1 are desirable. In Figures 6 to 8, we report only correlations that were significant at 99%.

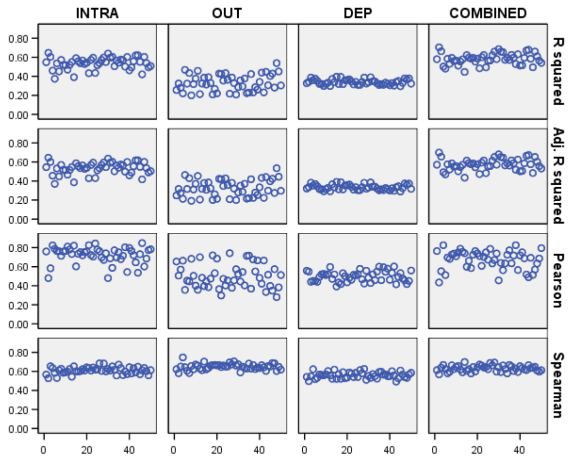


**Figure 6. Linear regression.**

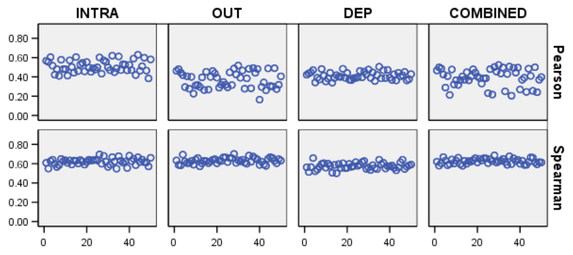


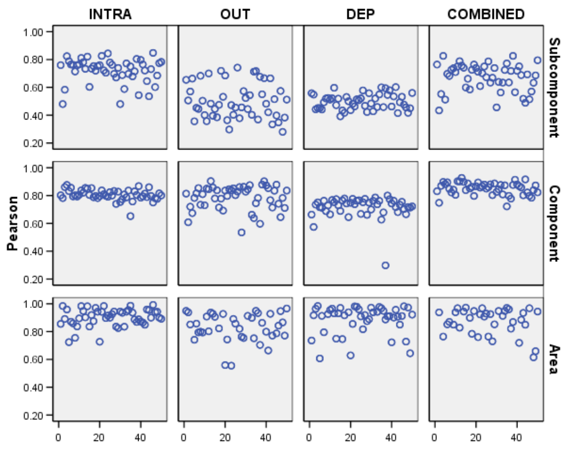
**Figure 7. Logistic regression results.**

### Linear regression

Figure 6 shows the results of four experiments on subcomponent level for linear regression modeling, each of them consisting of 50 random splits. Except for OUT graphs, we can observe the consistent $R^2$ and adjusted $R^2$ values. This indicates the efficacy of the models built using the random split technique. The values for Pearson are less consistent, still we can observe high correlations, especially for INTRA and COMBINED (around 0.70). The values for Spearman correlation (0.60) are very consistent and highest for OUT and COMBINED subgraphs. These values indicate the sensitivity of the predications to estimate failures—that is an increase/decrease in the estimated values is accompanied by a corresponding increase/decrease in the actual number of failures.

### Binary logistic regression

We repeated our experiments with the same 50 random splits using a binary logistic regression model. In contrast to linear regression, logistic regression predicts a value between 0 and 1. This value can be interpreted as failure-proneness, i.e., the likelihood to contain at least one failure. Figure 7 shows the results of



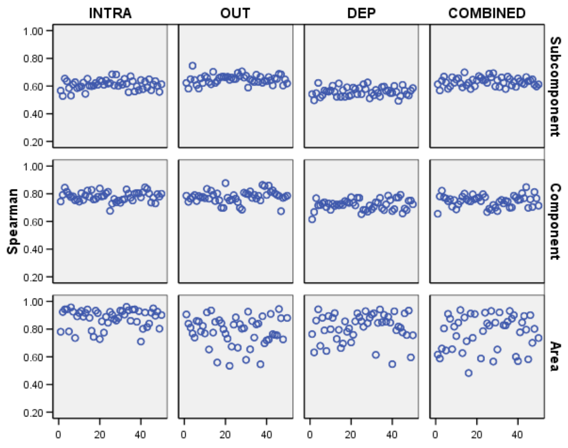


**Figure 8. Correlations for different levels of granularity (subcomponent/component/area)**

our random split experiments. All results are consistent, except the Pearson values. Compared to linear regression, the Pearson correlations are lower because the relation between predicted failure-proneness and actual number of failures is obviously not linear. Thus, using logistic regression did not make much difference in our case. Still, the results for both linear and logistic regression support our hypothesis, that *the complexity of dependency graphs can predict the number of post-release failures (H2).*

## 5.3. Granularity

The previous results were for subcomponent level. Figure 8 shows how the results for linear regression change when we make predictions for component and area level. We can observe that for both the maxima of correlation increases: for Pearson up to 0.927 (components) and 0.992 (areas); for Spearman up to 0.877 (components) and 0.961 (areas). While for component level the results are stable, we can observe many fluctuations for area level.

To summarize, the results for component level show that *the quality of the predications improves when they are made for subsystems that higher in the system's architecture (H3)*—the results for area level also support this hypothesis, however, they additionally demonstrate that the gain in predictive power can come with a decreased stability. Thus it is important to find a good balance between the granularity of reliable predictions and stability.

## 5.4. Threats to validity

In this section we discuss the threats to validity of our work. We assumed that fixes occur in the same location as the corresponding failure. Although this is not always true, this assumption is frequently used in research [10, 19, 23, 25]. 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 [4]. For this reason, we cannot assume a priori that the results of a study generalize beyond the specific environment in which it was conducted.
Since this study was performed on the Windows operating system and the size of the code base and development organization is at a much larger scale than many commercial products, it is likely that the specific models built for Windows would not apply to other products, even those built by Microsoft. This threat in particular is frequently misunderstood as a criticism on empirical studies. However, data on defects is rare and a common empirical research practice is to carry out studies for one project and replicate them on others. However, we are confident that dependency data has predictive power for other projects—we will repeat our experiments for other Microsoft products and invite everyone to do the same for other software.

## 6. Conclusion and consequences

We showed that for subsystems, one can use the complexity of dependency graphs for predicting failures. This helps for resource allocation and decision making. With respect to this, our lessons learned are as follows.

- ➢ Most dependency graph complexities can predict the number of failures (Sections 5.1 and 5.2).
- ➢ Validate any complexity measure before using it for decisions (Section 5.1).
- ➢ Find a balance between the granularity, reliability, and stability of predictions (Section 5.3).

We do not claim that dependency data is the sole predictor of post-release failures—however, our results are another piece in the *puzzle of why software fails*. Other effective predictors include code complexity metrics [23] and process metrics like code churn [22]. In our future work, we will identify more predictors and work on assembling the pieces of the puzzle. Also we plan to look at more non-linear regression and other machine learning teachniues. More specifically, we will focus on the following topics.

**Evolution of dependencies.** We will combine code churn and dependencies. More precisely, we will compare the dependencies of different Windows releases to identify *churned dependencies* and investigate their relation to failures.

**Development process.** How can we include the development process in our predictions? There are many *different characteristics* to describe the process, ranging from size of personnel to criticality, dynamism, and culture [6]. How much difference do agile and plan-driven development processes make with respect to failures? And how much impact has global development?

**The human factor.** Last but not least, humans are the ones who introduce failures. How can we include the *human factor* [16] into predictions about future failures? This will be a challenge for both software engineering and human computer interaction—and ultimately it will reveal why programmers fail and show ways how to avoid it.

## Acknowledgments

We would like to thank Thirumalesh Bhat for his support on the MaX tool. We would also like to thank Tom Ball, Vic Basili, Marc Eaddy, Madan Musuvathi, Adrian Schröter, and Andreas Zeller for discussions on the relationship between dependencies and failures.

## References

[1] F. B. e. Abreu and W. Melo, "Evaluating the Impact of OO Design on Software Quality", Proceedings of Third International Software Metrics Symposium, Berlin,1996.

[2] R. Allen, Garlan, D., "A Formal Basis for Architectural Connection", *ACM Transactions on Software Engineering and Methodology (TOSEM),* 6(3), pp. 213-249, 1997.

[3] V. R. Basili, L. C. Briand, and W. L. Melo, "A Validation of Object Orient Design Metrics as Quality Indicators", *IEEE Transactions on Software Engineering*, 22(10), pp. 751-761, 1996.

[4] V. R. Basili, F. Shull, and F. Lanubile, "Building Knowledge Through Families of Experiments", *IEEE Transac-*

*tions on Software Engineering*, 25(4), pp. 456 - 473, 1999.

[5] A. B. Binkley, Schach, S., "Validation of the coupling dependency metric as a predictor of run-time failures and maintenance measures", Proceedings of International Conference on Software Engineering, pp. 452 - 455, 1998.

[6] B. Boehm and R. Turner, *Balancing Agility and Discipline: A Guide for the Perplexed*. Boston, MA: Addison Wesley, 2003.

[7] C. Bron and J. Kerbosch, "Algorithm 457: finding all cliques of an undirected graph ", *Commun. ACM*, 16(9), pp. 575-577, 1973.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed: The MIT Press, 2001.

[9] G. Denaro, Morasca, S., Pezze., M, "Deriving models of software fault-proneness", Proceedings of International Conference on Software Engineering Knowledge Engineering, pp. 361-368, 2002.

[10] N. Fenton, Ohlsson, N., "Quantitative analysis of faults and failures in a complex software system", *IEEE Transactions in Software Engineering*, 26(8), pp. 797 - 814, 2000.

[11] N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*: Brooks/Cole, 1998.

[12] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting Fault Incidence Using Software Change History", *IEEE Trans. Softw. Eng.*, 26 (7), pp. 653-661, 2000.

[13] IEEE, "IEEE Standard 982.2-1988, IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliabile Software", 1988.

[14] E. J. Jackson, *A Users Guide to Principal Components*. Hoboken, NJ: John Wiley & Sons Inc., 2003.

[15] T. M. Khoshgoftaar, E. B. Allen, N. Goel, A. Nandi, and J. McMullan, "Detection of software modules with high debug code churn in a very large legacy system", Proceedings of Seventh International Symposium on Software Reliability Engineering, White Plains, NY, pp. 364-371, 1996.

[16] A. J. Ko and B. A. Myers, "A framework and methodology for studying the causes of software errors in programming systems", *Journal of Visual Languages & Computing*, 16(1-2), pp. 41-84, 2005.

[17] N. Medvidovic, Taylor, R.N., "A Classification and Comparison Framework for Software Architecture Description Languages", *IEEE Transactions in Software Engineering*, 26(1), pp. 70-93, 2000.

[18] A. Mockus, Zhang, P., Li, P., "Drivers for customer perceived software quality", Proceedings of International Conference on Software Engineering, pp. 225-233, 2005.

[19] K.-H. Möller and D. J. Paulish, "An empirical investigation of software fault distribution", Proceedings of Proceedings First International Software Metrics Symposium, pp. 82-90, 1993.

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

[21] N. Nagappan, Ball, T., "Explaining Failures Using Software Dependences and Churn Metrics," Microsoft Research Technical Report MSR-TR-2006-03, 2006.

[22] N. Nagappan, Ball, T., "Use of Relative Code Churn Measures to Predict System Defect Density", Proceedings of International Conference on Software Engineering (ICSE), pp. 284-292, 2005.

[23] N. Nagappan, Ball, T., Zeller, A., "Mining metrics to predict component failures", Proceedings of International Conference on Software Engineering, pp. 452-461, 2006.

[24] N. Ohlsson, Alberg, H., "Predicting fault-prone software modules in telephone switches", *IEEE Transactions in Software Engineering*, 22(12), pp. 886 - 894, 1996.

[25] T. Ostrand, Weyuker, E., Bell, R.M., "Predicting the location and number of faults in large software systems", *IEEE Transactions in Software Engineering*, 31(4), pp. 340 - 355, 2005.

[26] D. E. Perry, Wolf, A.E., "Foundations for the Study of Software Architecture", *ACM SIGSOFT Software Engineering Notes*, 17(4), pp. 40-52, 1992.

[27] M. Pinzger, Gall, H., Fischer, M., "Towards an Integrated View on Architecture and its Evolution", *Electronic Notes in Theoretical Computer Science*, pp. 183-196, 2005.

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

[29] M. Shaw, Clemants, P., "Toward boxology: preliminary classification of architectural styles", Proceedings of the Second international software architecture workshop and international workshop on multiple perspectives in software development, pp. 50 - 54, 1996.

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

[31] R. Subramanyam and M. S. Krishnan, "Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects", *IEEE Transactions on Software Engineering*, 29(4), pp. 297-310, 2003.

[32] D. B. West, *Introduction to Graph Theory*, 2nd ed: Prentice Hall, 2001.

[33] T. Zimmermann and N. Nagappan, "Predicting Subsystem Failures using Dependency Graph Complexities," Microsoft Research Technical Report MSR-TR-2006-126. 2006.