Abstract: Tools and methods in the context of Model-Driven Engineering have to be evaluated and tested. Unfortunately, adequate test models are scarcely available in many application domains, and available models often lack required properties. Test model generators have been proposed recently to overcome this deficiency. Their basic principle is to synthesize test models by controlled application of edit operations from a given set of edit operation definitions. If test models are created by randomly selecting edit operations, then they become quite unnatural and do not exhibit real-world characteristics; generated sequences of edit operation should rather be similar to realistic model evolution. To this end, we have reverse-engineered a carefully selected set of open-source Java projects to class diagrams and computed the differences between subsequent revisions in terms of various edit operations, including generic low-level graph edit operations and high-level edit operations such as model refactorings. Finally, we statistically analyzed the distribution of the frequency of these edit operations. We have checked the fitness of 60 distributions in order to correctly represent the statistical properties. Only four distributions have been able to adequately describe the observed evolution. The successful distributions are being used to configure our model generator in order to produce more realistic test models.

1 Introduction

Model Driven Engineering has gained a prominent role in the context of software engineering and many tools and methods have been proposed in the last couple of years. These tools and techniques have to be evaluated with regard to aspects such as efficiency, quality of results and scalability. Examples of such tools are model transformations engines, model search engines, model checkers and model versioning systems.

In many application domains real test models are barely available. Real models available often lack characteristics that are required for testing purposes. In fact, the requirements on test models depend a lot on the tool being tested. Some tools, e.g. differencing algorithms, need pairs or sequences of models where the evolution is precisely known while other
tools just need large test models to assess their efficiency and scalability. As discussed in [PSYK11], artificial models and their evolution should hence meet the following general requirements: (a) They should be correct according to their meta model. (b) They should satisfy extra constraints, e.g. multiplicities of references. (c) In the case of changes, the changes should contain atomic as well as high-level edit operations. (d) The changes and evolution should be “realistic” in the sense that they should mimic the statistical properties observable in real models.

The SiDiff Model Generator (SMG) [PSYK12] meets these requirements. It can be configured to create “realistic” test models if the statistical properties of the evolution in real models are observed and known. The SMG modifies a given base input model by applying edit operations, which are based on the meta-model of the input model. The application process is statistically controlled by a component called Stochastic Controller. The configuration for the stochastic controller contains random variates of distributions for different edit operations.

State-of-the-art approaches to understand the evolution of models of software systems are based on software metrics and similar static attributes; the extent of the changes between revisions of a software system is expressed as differences of metrics values, and further statistical analyses are based on these differences. Unfortunately, such approaches do not reflect the dynamic nature of changes well. For instance, considering the static metric Number of Methods (NOM) of classes: If we observe an increase of one in this metric between two subsequent revisions, the actual amount of change might be much larger, e.g. 5 existing methods deleted, 6 new methods added and 3 methods moved to another class.

This error can be avoided by first computing a precise specification of all changes between two revisions, i.e. a difference, and then computing difference metrics [Wen11]. In our above example we would use the difference metrics NOM-Deleted, NOM-Added and NOM-Moved in which we get 14=(5+6+3) changes in total rather than an increase by 1 in the static metric NOM. In other words, we have to count the occurrences of edit operations that have been applied between subsequent revisions of a system.

While this approach seems obvious, it remains to be shown that it can be successfully implemented. The most important question is how differences could be defined. Obviously, textual differences consisting of insertions and deletions of lines of source code will not be a basis for computing meaningful difference metrics. Thus, we reverse-engineered design-level class diagrams from a set of carefully selected open-source Java systems. These class diagrams were compared using advanced model comparison techniques in order to compute changes between revisions on two different levels of abstraction, i.e. based on two different sets of edit operation definitions. The first set of edit operations consists of low-level graph modifications. The second set of edit operations contains high-level edit operations including model refactorings which are applicable to class diagrams from a user’s point of view. Details of our approach to modeling and gathering changes of Java software are presented in Section 2. We investigated the fitness of 60 distributions to represent the statistical properties of the observed frequencies, both on low-level edit operations and high-level ones.

The rest of this paper is organized as follows: Section 3 discusses which software repos-
itories could be regarded as candidates for this investigation and justifies our choices. We then introduce the four successful distributions\(^1\) briefly in Section 4. More information for these statistical models can be found on the accompanying website of the paper [SYPKK12]. Section 5 then presents the results of the fitting of the successful distributions to the computed differences. Threads to validity of our work are discussed in Section 6; related work is discussed in Section 7. The paper ends in Section 8 with a summary and a conclusion.

\section{Modeling Changes of Java Software}

As we are interested in design-level changes, the source code of each revision of a Java software system must first be reverse-engineered into an appropriate model. This is accomplished by a parser that analyzes the structure of the code and creates a class diagram of the Java system. The simplified core of the meta model is depicted in Figure 1, while the complete meta model consists of 15 different element types and is presented in [SYPKK12].

The root element of every model is a project (JProject). Each project can contain a number of packages (JPackage), which in turn can form nested hierarchies. Packages can contain classes (JClass) and interfaces (JInterface). Interfaces can contain only methods (JMethod) and constants (JConstant), whereas classes can additionally contain attributes (JField). Naturally, methods can have parameters (JParameter).

The seven element types that are omitted in Figure 1 represent constructs which are specific to the Java programming language: Primitive types of Java are modeled as simple types (JSimpleType), arrays are represented as special elements (JArrayType). The concept of generics in the Java programming language is modeled by three element types (JGenericType, JTemplateBinding and JTemplateWrapper). Finally, enumerations are represented by two different element types (JEnumeration and JEnumerationLiteral).

Having the appropriate class diagrams at hand, a meaningful difference between two revi-

\(^{1}\)To be precise: Discrete Pareto, Yule, Warring and Beta-Negative Binominal distributions.
sions can be obtained by model comparison technology. A survey on approaches to model comparison can be found in [KRPP09]. Because of the lack of persistent identifiers in reverse-engineered models, we decided to use the SiDiff model differencing framework [KKPS12] in our analysis. We carefully adapted the matching engine to the comparison of the design-level class diagrams. Finally, the changes between revisions are reported on two different levels of abstraction which can be best explained by having a look at the processing pipeline of the SiDiff model differencing framework which is shown in Figure 2.

In the initial matching phase corresponding elements are identified. Based on this matching a low-level difference can then be derived. Generally, five different kinds of low-level edit operations are defined between two subsequent revisions of $R_n$ and $R_{n+1}$: Additions: An element is inserted in $R_{n+1}$. Deletions: An element is removed from $R_n$. Moves: An element is moved to a different position, i.e. the parent element is changed in $R_{n+1}$. Attribute Changes: An element is updated in $R_{n+1}$, e.g. its name or visibility is changed. Reference Changes: A reference of an element is changed, e.g. a method now has a different return type.

![Figure 2: Coarse-grain structure of model comparison tools](image)

Difference metrics for low-level changes can be computed for each element type and for each kind of edit operations by counting their occurrences in a difference. Thus, we obtain a total number of 75 difference metrics, i.e. 5 difference types times 15 element types. Low-level changes can be semantically lifted to high-level changes which usually comprise a set of low-level changes. For example, the same result of the low-level changes adding an element of type JField and subsequently setting its JType can be achieved by one high-level operation which takes the JType of the JField to be created as an additional argument and achieves both low-level changes together (see Figure 1). High-level operations such as refactorings can comprise even larger sets of low-level changes. The semantic lifting engine that we have used in our study is presented in [KKT11]. Obviously, the set of high-level operations to be detected has to be defined individually for each modeling language. The operations defined for the class diagrams can be found at the accompanying website [SYPKK12]. In sum, we identified and defined a total number of 188 high-level edit operations in class diagrams, including 12 refactorings. Quantitative measurements of high-level changes can be easily obtained by counting the occurrences of such edit operations.
3 Selection of the Sample Projects

In the previous section, we showed that difference metrics between class diagrams are a new, more fine-grained description of changes in software systems. An investigation of the statistical properties of these difference metrics must be based upon a set of representative sample projects. This section describes and justifies our selection of the sample projects and sketches the technical preparations which were necessary before the statistical analysis could be performed.

We applied the following constraints in the selection of projects: First, only real, non-trivial software systems are to be considered. Secondly, the projects must be developed over a long period in order to let us study their evolution. Thirdly, the selected projects must be typical Java software systems.

We found out that the projects reported in the Helix Software Evolution Data Set (HDS) [VLJ10] fulfill these three requirements. The projects stem from different application domains, have been under development for at least 18 months, have all more than 15 releases and contain at least 100 classes.

We randomly selected nine projects from the HDS, the website of the paper [SYPKK12] provides basic information about the selected projects and their model representations. We checked out all revisions from the respective repositories. After purifying the data from any non-source files, a parser created a class diagram for each revision of each project; totally 6559 models were created. Models of successive revisions were then compared by the SiDiff model differencing framework and the differences between each pair were computed according to the procedure described in Section 2. Finally for each project, the values of the 75 low-level and the 188 high-level difference metrics have been computed serving as input for our statistical analysis. Hence, for each project $p$ and each difference metric $m$, our data set $S_{p,m}$ is the calculations of $m$ between all subsequent revisions of $p$. Additionally, because the models are obtained by reverse engineering the source code of the projects, there are cases that between two subsequent revisions only parts of the system were changed which do not influence the model representation. Therefore if for a given project $\hat{p}$ all of its computed difference metrics are zero between two subsequent revisions, the computed data is excluded from the corresponding data sets $S_{\hat{p},m}$. We tried to fit each distribution on every $S_{p,m}$.

4 Statistical Models for Describing Changes

The previous section described how the sample projects were selected and how difference metrics were computed. Our goal was to find statistical models, i.e. distributions, which correctly model the changes observed in our sample data sets. The main challenge for such distributions are large changes: they do happen, but their probabilities are quite small. Suitable distributions must therefore be skewed and asymmetric with heavy tails.

Many continuous and discrete univariate distributions are known [JKB94, JKK05, WA99].
We tested 60 distributions\(^2\). Only four discrete distributions with heavy tails performed acceptable, although with different levels of success (see Section 5). These four are the discrete Pareto distribution (DPD) of the power law family and the Yule distribution (YD), the Waring distribution (WD) and the Beta-Negative Binomial distributions (BNBD) from the family of hypergeometric distributions.

These successful distributions are briefly introduced here; the inclined reader can find the detailed information of these distributions on the website of the paper [SYPKK12].

**Power Law and Discrete Pareto Distribution** Considering the function \(y = f(x)\), it is said that \(y\) obeys **Power Law** to \(x\) when \(y\) is proportional to \(x^{-\alpha}\). Such relations, which have different applications, have been observed in linguistics, biology, geography, economics, physics and also computer science, e.g. the size of computer files, grid complex networks, the Internet and web pages hit rates (see [New05, Mit04, IS10, AH02]).

The discrete Pareto distribution that is used throughout this paper is of power law and is based on the Riemann Zeta function ([EMOT55, GR07, OLBC10]); it is obtained from the General Lerch distribution ([ZA95, WA99, JKK05]). It takes a real value \(\rho > 0\) as shape parameter.

**Yule, Waring and Beta-Negative Binomial Distributions** The Yule distribution, which has applications in taxonomy of species in biology, has just one parameter \(b\) which is a positive real. The Waring distribution, which yields the Yule distribution as its special case has two real parameters, \(b > 0\) and \(n \geq 0\) ([WA99, JKK05]).

Both of the Yule and Waring distributions have been generalized to a hypergeometric distribution called the Beta-Negative Binomial distribution\(^3\) [Irw75]. The distribution has three parameters, \(\alpha, \beta\) and \(n\) and is usually denoted by \(\text{BNB}(\alpha, \beta, n)\). Its parameters are positive reals [WA99, JKK05]. This distribution can be obtained from the Negative Binomial distribution when the probability \(p\) of its Bernoulli trials has the Beta distribution.

### 5 Analysis of the Data Set and Results

As discussed in Section 3, our data sets \((S_{p,m})\) were computed considering 263 (=75+188) difference metrics for each of the 9 projects through its life span. This section discusses how the four proposed distributions fit to the observed data.

To decide whether or not a distribution \(\mathcal{D}\) fits to the data sets \(S_{p,m}\), the null and alternative hypotheses, i.e. \(\mathcal{H}_0\) and \(\mathcal{H}_1\), are defined as follows: \(\mathcal{H}_0\): The data set obeys the distribution \(\mathcal{D}\). \(\mathcal{H}_1\): The data set does not obey the distribution \(\mathcal{D}\).

Different methods exist for fitting distributions and estimating parameters. Two commonly used are the method of moments and the maximum-likelihood estimation method (MLE). The former tries to estimate the parameters using the observed moments of the sample, by equating them to the population moments and solving the equations for the parameters.

---

\(^2\)See the accompanying website of the paper at [SYPKK12] for the full list of the tested distributions.

\(^3\)In the literature also referred to as the Generalized Waring distribution and the Beta-Pascal distribution.
The MLE method estimates the parameters by trying to maximize the logarithm of the likelihood function. In this paper the MLE method is employed and the calculations are done using Wolfram Mathematica® 8.0.4 computational engine.

Due to the discrete nature of the difference metrics and the four distributions, the Pearson’s Chi-Square test was used. The significance level was set to 0.05. At first the parameters of the desirable distribution $D$ were estimated, then the p-value of the Pearson’s chi-square statistic was calculated in order to decide whether to reject $H_0$ in favor of $H_1$ or not.

For the 60 distributions that were initially tested, totally $40500 (60 \times 9 \times 5 \times 15)^4$ fittings were considered for low-level operations and $101520 (60 \times 9 \times 188)^5$ for high-level ones. From those, just the results for the four proposed distributions are covered in detail here, separately for low-level and high-level operations.

In the rest of this section only successfully accomplished fittings are reported, i.e. when we were able to decide whether to reject $H_0$ or not; our summaries of the results are based on such successfully accomplished fittings. There were cases where the computed difference metrics were zero for all revisions and no fittings were possible; they are not considered in our analysis.

Since the number of high-level operations are too many (188), we are unable to fully publish the detailed results. Hence, only the summary of our findings is provided here. The more detailed results are available on the accompanying website of the paper [SYPKK12].

5.1 Fitting the Discrete Pareto Distribution

Since the support of the discrete Pareto distribution consists of positive integers, the fittings are done on the shifted data which are obtained by adding +1 to members of our data sets. The shift brings the data in the domain of this distribution.

**Low-Level Operations** Totally 294 successful fittings were performed for low-level operations for the DPD. $H_0$ was not rejected 157 times, so the non-rejection ratio is about 53%.

The DPD is most successful in describing changes of packages and interfaces, but with lower success rate for additions of new packages. It has generally a moderate rate of success in describing changes of classes and performs worse when fields are considered. The DPD is not successful in describing changes of methods and parameters due to a success rates of under 30%. Changes of array types could be fully modeled by this distribution. Additions and deletions of other element types could also be described with moderate success. Figure 3 shows one probability plot of the observed and the fitted probabilities for the JFreeChart project for the difference metric additions of methods. The plot is near to the ideal dashed line, so we constitute a good approximation here although $H_0$ is rejected.

**High-Level Operations** For the high-level operations $H_0$ was not rejected 69% of the times; a considerable improvement for the DPD compared to its application on low-level operations.

---

460 distributions, 9 projects, 5 low-level operations, 15 model element types.

560 distributions, 9 projects, 188 high-level operations.
operations. We conclude that the DPD serves better to describe high-level changes.

5.2 Fitting the Yule Distribution

**Low-Level Operations**  From the 294 fittings of the YD, $H_0$ was rejected in favor of $H_1$ 180 times, giving a non-rejection rate of almost 39% which makes this distribution the least successful one.

The YD was fully successful in describing moves, reference change and update on packages and interfaces; but for additions and deletions this rate drops to less than 50%. For classes, fields, methods and parameters it performs weakly most of the time, rarely reaching 50% of success. Describing additions of elements is only moderately successful, while deletions of elements are better modeled compared to additions. Nevertheless, the YD performs worse than the DPD for both kinds of edit operations. Figure 4 shows the CDF plot of the observed probabilities and the fitted model, for reference changes of methods in the HSQLDB project. There are large differences between the observed probabilities (blue lines) and those who are obtained by the fitted distribution (red lines), which indicates that the fitting is bad and $H_0$ is strongly rejected.

**High-Level Operations**  Here, $H_0$ was not rejected at the rate of 49% for the YD. Although this shows an improvement of 10% compared to low-level operations, the YD also performs the worst for both low-level and high-level operations.

![Figure 3: The probability plot of JFreeChart: adding of methods, discrete Pareto distribution.](image1)

![Figure 4: The CDF plot of HSQLDB: reference change of methods, Yule distribution.](image2)

5.3 Fitting the Waring Distribution

**Low-Level Operations**  For the WD, $H_0$ was not rejected 252 out of 294 times, which gives a very good non-rejection rate of 86%.

The WD was fully successful in describing changes of packages and interfaces. For classes

---

6 CDF: Cumulative Distribution Function.
it was successful almost 70% of the times or more and this rate is even higher with 90% and more when fields are considered. For changes on methods we get good success rates between 45% to 90%. Figure 5 shows the probability plot for the Maven project considering additions of methods. For changes on parameters this distribution performs also well. The exceptions are reference updates for which it has only a success rate of 25%. It was fully successful in describing changes of array types, constants, simple types, generic types and the other elements (see Section 2).

Figure 7 shows the p-value plot of the DataVision project which shows that the distribution was almost fully successful in describing all kinds of changes on all element types except for reference changes of parameters (row=4, column=6). The black cells emerge from two possibilities: Either there was no data discovered by our change detection tool, i.e. the difference metric value is 0, or the distribution could not be fitted because either the data did not fulfill the requirements of the distribution or parameters could not be estimated. It should be mentioned that the first case happens most of the time, while the second case occurs very rarely. White cells indicate that the calculated p-value was less than the specified significance level, i.e. $H_0$ was rejected. Finally, when the calculated p-value was above the significance level, i.e. $H_0$ was not rejected, the cell is colored. The more intense the color of the cell, the higher the p-value.

**High-Level Changes** The non-rejection ratio of the null hypotheses is at almost 93%, which is obviously a good result. The WD performs very well in describing the change behavior observed in the models almost for all defined high-level operations. This distribution is performing almost 25% better compared to the DPD.

**Low-Level Operations** For the BNBD, $H_0$ was rejected 34 out of 294 times in favor of $H_1$, yielding an 88% non-rejection rate, which is a slight improvement to the WD.

The performances of the BNBD and the WD are almost identical for the difference metrics over the 15 element types. Like the WD, the BNBD is not successful in modeling reference

5.4 Fitting the Beta-Negative Binomial Distribution
changes for parameters with a success rate of only about 25%. Figure 6 depicts the CDF plot for the JFreeMarker project considering deletions of fields. It can be seen that the predicted and observed probabilities completely overlap. Figure 8 shows the p-value plot of the Struts project which is almost fully statistically modeled by the BNBD. In this particular example, only reference changes of interfaces could not be modeled (row=4, column=3).

**High-Level Operations** The non-rejection ratio for the null hypotheses is more than 94% which is quite similar to the WD when high-level operations are considered. Hence, this distribution performs also very well in describing the evolution of class diagrams based on high-level operations.

![Figure 7: P-Value plot of the whole DataVision project when the Waring distribution is used.](image1)

![Figure 8: P-Value plot of the whole Struts project when the BNB distribution is used.](image2)


### 5.5 Conclusion of Fittings

Considering low-level operations, as discussed in Sections 5.1 and 5.2, the discrete Pareto distribution (DPD) is only to some extent successful in describing the observed changes. The Yule distribution is not generally recommended due to its low success rates.

The DPD is generally good in describing changes on packages and interfaces. It is moderately suitable for classes and fields and only very limited suited for methods and parameters. For the rest of the element types, it performs generally good. When high-level operations are taken into account, the DPD performs much better (near 70%) in describing the changes.

Since the DPD is of power law family, we additionally conclude that the **Power Law** is observable to some extent in low-level changes between class diagrams of open-source Java systems and its presence is more apparent when high-level operations are considered; actually based on the shifted data (see Section 5.1).

Considering low-level operations, the Waring distribution (WD) and the Beta-Negative Binomial distribution (BNBD) show much higher success rates than the other two distributions. Both of them perform equally well at explaining the observed difference metrics.
for almost all element types. Despite their successes, they are not successful in predicting reference changes of parameters and therefore should be used with caution in this case.

For the high-level operations the success rates of the WD and the BNBD even increases to almost 94%; making them capable of statistically modeling almost any high-level edit operations in addition to low-level ones.

Although the BNBD is an extension to the WD and has one additional shape parameter, this does not add any benefit to its predictive powers. Furthermore, estimating the parameters of the WD needs less effort and is less time consuming.

Comprehensive information about our tests is provided on the website [SYPKK12].

6 Threats to Validity

In this section we discuss threats to the validity of the presented results.

Accuracy One threat is based on the way differences between class diagrams were computed. Model comparison algorithms can produce differences which are generally considered sub-optimal or wrong. [Wen11] has analyzed this error for class diagrams and the SiDiff differencing framework [KKPS12]; the total number of errors was typically below 2%. This very low error rate cannot have a significant effect on the results of our analysis. The second threat is how accurate the high-level operations are recognized. If model elements were matched based on persistent identifiers, the operation detection could be guaranteed to deliver correct results [KKT11], i.e. all low-level changes were grouped into high-level operations and no low-level ones remain ungrouped. As matchings are computed based on similarity heuristics, possible “incorrect” matches can lead to false negatives, i.e. edit operations which have actually been applied but which were not detected. We calculated the rate of ungrouped low-level changes which was below 0.3%, thus both of the difference derivation and semantic lifting engines in our pipeline (see Figure 2) performed quite well and the results are not distorted.

External Validity Another important question is whether our results are generalizable. Our test data set consists of medium-sized, open-source Java software systems. It is highly probable that our results also hold for large Java software systems as our preliminary studies show. It is not yet clear whether our results also hold for closed software systems, in particular if company-specific programming styles and design rules are enforced.

It is also less than clear whether our results hold for other object-oriented languages, e.g. C++. The question is whether the data model for class diagrams (see Figure 1) is still appropriate. These questions are subject of further research.
7 Related Work

Vasa et al. [VSN07] studied the evolution of classes and interfaces in open-source Java software based on static metrics for released versions of the systems. They showed that the average metric values are almost stable over the histories, i.e. the average size and popularity of classes does not change much between released versions. They also analyzed which kind of classes tend to change more. For this, correspondences between classes are established based on their fully qualified name. The amount of change is measured based on value changes of 25 static software metrics. They showed that more complex classes are more likely to undergo changes. This research is continued in [VSNW07], where they consider additional static metrics. Here they show that the majority of changes happen on a small portion of classes. They also analyze the history of classes superficially based on a comparison between the static metric values counted in the final version of the system and those counted in preceding versions. In [Vas10] Vasa presented an extended and more detailed version of his research.

All this research mentioned above, is based only on changes of static metric values and does not take the evolution, i.e. the actual changes between two versions, into account. Furthermore, only released versions of the software systems are considered, i.e. the time period between two versions is rather long. In contrast we used much finer time intervals, i.e. revisions, which reflect the changes more accurately. Lastly, no parametric distribution is reported in any of the publications.

All following papers focus on finding occurrences of distributions for static metric values on software systems. They also have in common that they only use single system snapshots as the base for their analysis, i.e. the topic of system evolution is not brought up at all. Additionally, neither of them tried nor proposed the Yule, Waring and Beta-Negative Binomial distributions in their researches.

Concas et al. [CMPS07] studied metrics of VisualWorks Smalltalk and compared them to those observed in Eclipse and the JDK. They showed that the power law and the continuous univariate Pareto distribution are observable in their data.

Baxter et al. [BFN+06] studied the structure of Java software systems based on static software metrics. They report that some, but not all, of the considered metrics follow power law.

Wheeldon and Counsell [WC03] analyzed power law distributions for different forms of object-oriented couplings in Java software. They extracted graph structures from the source code representing the couplings, e.g. inheritance or return type usage and counted static metrics on them. They identified 12 coupling-related metrics that obey power law.

8 Summary and Conclusion

In this paper we thoroughly studied the evolution of reverse engineered class diagrams based on difference metrics of low-level and high-level edit operations.
Nine typical open-source Java projects were initially selected and a parser created design-level class diagram representations of 6559 source code revisions. All subsequent class diagrams were compared by a model differencing framework. On each calculated difference, 75 low-level and 188 high-level difference metrics were counted.

We then addressed the question which statistical model would be the best to describe the observed changes. Sixty continuous and discrete univariate distributions have been tested and only four of them performed acceptable. These are the discrete Pareto, Yule, Waring and Beta-Negative Binomial distributions. The Yule distribution is generally not recommended due to low success rates. The discrete Pareto distribution showed an acceptable performance on low-level changes and describes changes on high-level changes quite well at almost 70% success rate. Additionally we conclude the presence of the Power Law to some extend in the analyzed difference metric values when shifted (see Section 5.1).

The Waring and the Beta-Negative Binomial distributions are the most successful distributions. They can describe almost any type of low-level change for each element type with an success rate near to 90%. The only exceptions are reference changes for parameters. For high-level changes these two distributions perform even better reaching success rates of 93% and 94% respectively. They are capable of modeling almost any high-level changes.

The knowledge of this research is directly used in the SiDiff Model Generator [PSYK12] to create synthetic models emulating realistic evolution of software systems.

References


