Statistical Analysis and Simulation of Design Models Evolution

DISSERTATION

zur Erlangung des Grades

Doktor der Naturwissenschaften (rer. nat.)

vorgelegt von

M.Sc. Hamed Shariat Yazdi

eingereicht bei
der Naturwissenschaftlich-Technischen Fakultät
der Universität Siegen

Siegen, 2015

1. **Gutachter:** Prof. Dr. Udo Kelter (Universität Siegen).

2. **Gutachter:** Prof. Dr. Lefteris Angelis (Aristotle University of Thessaloniki).

Tag der mündlichen Prüfung: 21.08.2015.

I would like to dedicate my dissertation to my parents

Houshang Shariat Yazdi *and* ***Akram Samsami***

and to my brothers

Ali Shariat Yazdi *and* ***Reza Shariat Yazdi***

without whom none of my success would be possible.

Acknowledgements

Many great and wonderful people have supported me to accomplish my PhD education. First of all, this work owes its success to my family. I would like to gratefully acknowledge my parents, Houshang and Akram, as well as my brothers, Ali and Reza. They always have strongly supported me by any means they could.

I would like to thank my PhD supervisors Prof. Dr. Udo Kelter, chair of the Software Engineering Group at the University of Siegen, and Prof. Dr. Lefteris Angelis, chair of Statistics and Information Systems Group and Deputy Head of Department of Informatics at the Aristotle University of Thessaloniki. I am very thankful for their continuous support, constructive feedback and fruitful discussions.

I also would like to thank Prof. Dr. Madjid Fathi, chair of the Institute of Knowledge-Based Systems and Knowledge Management at the University of Siegen, for his support and encouragement during my education in Siegen. He has always offered his help and support generously.

I have had a memorable time in the Software Engineering Group at the University of Siegen. I am very thankful to all of my colleagues and friends who have always supported me in different ways. My thanks to my friend Dr. Stefan Berlik for his support, help and very interesting discussions. I would like to thank my other friends and colleagues: Pit Pietsch, Timo Kehrer, Frank Schuh, Mike Schmidt, Dr. Sven Wenzel, Michaela Rindt, Dennis Reuling, Christopher Pietsch, Viktor Dück and Roswitha Eifler. Timo Kehrer, Pit Pietsch and Frank Schuh have always been ready to help.

Last but not least, I would like to thank my friends Mr. Mehdi Bohlouli and his wife Mrs. Dr. Sanaz Mohammad Hassani for their encouragement and strong support, specially during difficulties.

May, 2015
Siegen, Germany
Hamed Shariat Yazdi

Abstract

Tools, algorithms and methods in the context of Model-Driven Engineering (MDE) have to be assessed, evaluated and tested with regard to different aspects such as correctness, quality, scalability and efficiency. Unfortunately, appropriate test models are scarcely available and those which are accessible often lack desired properties. Therefore, one needs to resort to artificially generated test models in practice.

Many services and features of model versioning systems are motivated from the collaborative development paradigm. Testing such services does not require single models, but rather pairs of models, one being derived from the other one by applying a known sequence of edit steps. The edit operations used to modify the models should be the same as in usual development environments, e.g. adding, deleting and changing of model elements in visual model editors. Existing model generators are motivated from the testing of model transformation engines, they do not consider the true nature of evolution in which models are evolved through iterative editing steps. They provide no or very little control over the generation process and they can generate only single models rather than model histories. Moreover, the generation of stochastic and other properties of interest also are not supported in the existing approaches.

Furthermore, blindly generating models through random application of edit operations does not yield useful models, since the generated models are not (stochastically) realistic and do not reflect true properties of evolution in real software systems. Unfortunately, little is known about how models of real software systems evolve over time, what are the properties and characteristics of evolution, how one can mathematically formulate the evolution and simulate it.

To address the previous problems, we introduce a new general approach which facilitates generating (stochastically) realistic test models for model differencing tools and tools for analyzing model histories. We propose a model generator which addresses the above deficiencies and generates or modifies models by applying proper edit operations. Fine control mechanisms for the generation process are devised and the generator supports stochastic and other properties of interest in the generated models. It also can generate histories, i.e. related sequences, of models. Moreover, in our approach we provide a methodological framework for capturing, mathematically representing and simulating the evolution of real design models. The proposed framework is able to capture the evolution in terms of edit operations applied between revisions. Mathematically, the representation of evolution is based on different statistical distributions as well as different time series models. Forecasting, simulation and generation of stochastically

realistic test models are discussed in detail. As an application, the framework is applied to the evolution of design models obtained from sample a set of carefully selected Java systems.

In order to study the the evolution of design models, we analyzed 9 major Java projects which have at least 100 revisions. We reverse engineered the design models from the Java source code and compared consecutive revisions of the design models. The observed changes were expressed in terms of two sets of edit operations. The first set consists of 75 low-level graph edit operations, e.g. add, delete, etc. of nodes and edges of the abstract syntax graph of the models. The second set consists of 188 high-level (user-level) edit operations which are more meaningful from a developer's point of view and are frequently found in visual model editors. A high-level operation typically comprises several low-level operations and is considered as one user action.

In our approach, we mathematically formulated the pairwise evolution, i.e. changes between each two subsequent revisions, using statistical models (distributions). In this regard, we initially considered many distributions which could be promising in modeling the frequencies of the observed low-level and high-level changes. Six distributions were very successful in modeling the changes and able to model the evolution with very good rates of success. To simulate the pairwise evolution, we studied random variate generation algorithms of our successful distributions in detail. For four of our distributions which no tailored algorithms existed, we indirectly generated their random variates.

The chronological (historical) evolution of design models was modeled using three kinds of time series models, namely ARMA, GARCH and mixed ARMA-GARCH. The comparative performance of the time series models for handling the dynamics of evolution as well as accuracies of their forecasts was deeply studied. Roughly speaking, our studies show that mixed ARMA-GARCH models are superior to other models. Moreover, we discuss the simulation aspects of our proposed time series models in detail.

The knowledge gained through statistical analysis of the evolution was then used in our test model generator in order to generate more realistic test models for model differencing, model versioning, history analysis tools, etc.

Kurzfassung

Im Kontext der modellgetriebenen Entwicklung müssen Werkzeuge, Algorithmen und Methoden bewertet, evaluiert und getestet werden. Dabei spielen verschiedene Aspekte wie Korrektheit, Qualität, Skalierbarkeit und Effizienz eine große Rolle. Problematisch dabei ist, dass geeignete Testmodelle nur spärlich verfügbar sind. Verfügbare Modelle weisen darüber hinaus die für Evaluationszwecke gewünschten Eigenschaften oft nicht auf. Aus diesem Grund muss in der Praxis auf künstlich erzeugte Testmodelle zurückgegriffen werden.

Viele der Funktionalitäten von Modellversionierungssystemen sind motiviert von den Paradigmen der kollaborativen (Software) Entwicklung. Für das Testen derartiger Funktionalitäten braucht man keine einzelnen Modelle, sondern Paare von Modellen, bei denen das Zweite durch Anwendungen einer bekannten Sequenz von Editierschritten auf das Erste erzeugt wird. Die genutzten Editieroperationen sollten dabei die gleichen sein, die bei den typischen Entwicklungsumgebungen angewendet werden, beispielsweise das Hinzufügen, Löschen oder Verändern von Modellelementen in visuellen Editoren. Derzeit existierende Modellgeneratoren sind motiviert durch das Testen von Modelltransformationsumgebungen. Dabei berücksichtigen sie nicht die wahre Natur der (Software) Evolution, bei der die Modelle iterativ durch die kontrollierte Anwendung einzelner Editierschritte verändert werden. Dabei bieten sie nur wenig Kontrolle über den Prozess der Generierung und sie können nur einzelne Modelle, aber keine Modellhistorien, erzeugen. Darüber hinaus werden gewünschte Eigenschaften, beispielsweise eine stochastisch kontrollierte Erzeugung, von den derzeit existierenden Ansätzen nicht unterstützt.

Aufgrund der (blinden) zufallsgesteuerten Anwendungen von Editieroperationen werden keine brauchbaren, (stochastisch) realistischen Modelle generiert. Dadurch repräsentieren sie keine Eigenschaften von Evolutionen in echten Systemen. Leider gibt es wenig wissenschaftliche Erkenntnis darüber, wie Modelle in realen Systemen evolvieren, was die Eigenschaften und Charakteristika einer solchen Evolution sind und wie man diese mathematisch formulieren und simulieren kann.

Um die zuvor genannten Probleme zu adressieren, stellen wir einen allgemeinen Ansatz zur (stochastischen) Generierung realer Testmodelle zur Verwendung in Differenzwerkzeugen und Historienanalysen vor. Unser Generator generiert oder modifiziert Modelle durch geeignete Anwendung von Editieroperationen. Sowohl feine Kontrollmechanismen für den Generierungsprozess als auch die Unterstützung von stochastischen und anderen interessanten Eigenschaften in den generierten Modellen zeichnen den Generator

aus. Zusätzlich kann dieser Historien, d.h. abhängige/zusammenhängende Änderungssequenzen, von Modellen generieren. Unser Ansatz bietet eine methodische Umgebung für das Aufzeichnen, die mathematische Repräsentation als auch das Simulieren von Evolutionen realer Modelle. Die vorgestellte Umgebung kann die Evolution in Form von Editieroperationen, angewandt zwischen Revisionen, erfassen. Die mathematische Repräsentation der Evolution basiert sowohl auf verschiedenen stochastischen Verteilungen als auch unterschiedlichen Modellen von Zeitreihen. Das Vorhersagen, Simulieren und Generieren von stochastisch realistischen Testmodellen wird im Detail diskutiert. Als praktische Anwendung setzen wir unsere Umgebung im Rahmen einer Modellevolution von sorgfältig ausgewählten Java-Systemen ein.

Im Rahmen dieser Arbeit wurde die Evolution von Design Modellen auf Basis von neun Open-Source Java Projekten analysiert. Für jedes Projekt lagen mindestens 100 Revisionen vor, aus deren Quelltexten Design Modelle nachkonstruiert wurden. Die dabei gefunden Änderungen konnten anhand zwei verschiedener Mengen von Editieroperationen beschrieben werden. Die erste Menge besteht aus 75 einfachen Graph-Operationen. Beispiele dafür sind das Hinzufügen, Löschen, etc. einzelner Knoten und Kanten im abstrakten Syntax-Graphen der Modelle. Die zweite Menge enthält 188 komplexe Editieroperationen. Komplexe Editieroperationen haben für Entwickler eine höhere Aussagekraft, da sie auf dem gewohnten Abstraktionsniveau des Entwicklers angesiedelt und oftmals in visuellen Modelleditoren zu finden sind. Typischerweise besteht eine komplexe Editieroperation dabei aus mehreren einfachen Operationen, wobei die Ausführung der komplexen Operation immer als eine einzelne Aktion angesehen wird. Um die schrittweise Evolution, also die Veränderung aufeinanderfolgender Revisionen, zu analysieren betrachteten wir verschiedene statistische Modelle (Distributionen). Von allen betrachteten Distributionen erwiesen sich sechs als sehr erfolgreich dabei die beobachteten Veränderungen und die Evolution der Modelle auf Basis einfacher und komplexer Editieroperationen zu beschreiben. Um die Evolution weiter simulieren zu können, betrachteten wir Algorithmen für die Erstellung von Zufallsvariaten der erfolgreichen Distributionen. Für vier der Distributionen, für die keine derartigen Algorithmen verfügbar waren, wurden die Zufallsvariate indirekt abgeleitet.

Die chronologische (historische) Evolution von Modellen wurde auf Basis von drei Zeitreihen nachgebildet, konkret ARMA, GARCH und einer Mischung aus ARMA-GARCH. Sowohl deren Leistungsfähigkeit, Evolutionsdynamik darstellen zu können, als auch die Genauigkeit von Vorhersagen wurden im Detail analysiert und gegenübergestellt. Grob gesagt zeigen unsere Ergebnisse, dass ARMA-GARCH Modelle besser als die übrigen geeignet sind. Zusätzlich diskutieren wir ausführlich die Simulationsmöglichkeiten aller vorgestellten Zeitreihen.

Die Ergebnisse unserer statistischen Analysen der Evolution haben wir dann in unserem Testmodell Generator eingesetzt. So konnten wir realistische Testmodelle generieren, die für Modelldifferenz-, Versionierungs- und Historienanalysewerkzeuge u.s.w. verwendet werden können.

Contents

List of Figures

List of Tables

List of Algorithms

Part I

Introduction and Preliminaries

CHAPTER 1

Introduction

1.1 Introduction

The development of software systems has always experienced increase in abstraction levels, from handling and modeling of problems to implementing solutions. Historically, the development has shifted from low level machine code to structural programming practices, to object-oriented programming, to domain specific languages, and finally to model-driven approaches [Ameller, 2009]. Model-Driven Engineering (MDE) can be considered the top most in the abstraction hierarchy [Stahl et al., 2006]. In MDE, systems and phenomena of interest are handled and implemented as models. "A model captures a view of a system. It is an abstraction of the system, with a certain purpose. This purpose determines what is to be included in the model and what is irrelevant. Thus the model completely describes those aspects of the system that are relevant to the purpose of the model, at the appropriate level of detail" [OMG, 2012b].

As name suggests, models are the main artifacts in MDE and all tools, algorithms and methods are built upon the concept of models. In addition to have a better way of modeling real world systems, MDE promises to increase productivity through separation of business logic from implementation [Selic, 2003, Stahl et al., 2006]. As models of business logic evolve to accommodate changing requirements of the business, models of implementation independently evolve to cope with technological changes. In such a separation, business models can be transformed to implementation models and the implementation models themselves, can be transformed to executable code. Generally, the models which are independent of any technical or implementation-specific issues are referred to as Platform-Independent Models (PIM). In contrast Platform-Specific models (PSM) are bounded to implementation-related or technological issues. A model transformation can transform a PIM to another PIM or PSM. Similarly, a PSM can be transformed to another PSM or to code. Model transformations are intrinsic nature of MDE which increase productivity [Sendall and Kozaczynski, 2003].

In addition to evolving models resulted from changes in technologies or business logic, the shift of development from the code level to the model level results in versions of models which evolve over time. The development is a repetitive and incremental process in which software artifacts are developed over time. The situation becomes more complex when software developers collaborate to implement MDE solutions based on models. Traditional collaborative development environments are mostly based on repositories and are code-centric. To support collaborative work, such repositories allow developers to compute differences between various versions of code. If conflicts between versions are detected, the conflicts can be resolved and the code can be merged. To bring similar functionality to MDE, in addition to suitable model repositories, proper model differencing and model versioning tools are needed [Mens et al., 2005]. In this regard, much research effort have been put in the fields of model versioning and model differencing [Kolovos et al., 2009, Stephan and Cordy, 2013, Brosch et al., 2012].

In MDE generally, and in the domain of model transformation, model differencing and model versioning specifically, test models are of great importance. Test models are used to assess different aspects of algorithms, tools and methods e.g., correctness, scalability and efficiency. The requirements of test models usually depend on the nature of processes or tools being tested and are not necessarily the same for all tools. For instance model repositories or model search engines, might need large or huge models for testing their efficiency, scalability and performance. As another example, other tools such as model differencing, model versioning, model tracing and history analysis tools need pairs or sequences of models in which changes between models are precisely documented.

Although the use of test models in MDE seems trivial, in practice various problems arise. The biggest obstacle is that test models are quite scarce and meager existing ones often lack desired properties. To overcome these problems, artificial test models have been employed [Wu et al., 2012]. The majority of existing approaches (see Section 3.2) are motivated from the domain of model transformation testing. Roughly speaking, the typical approach in model transformation testing is to generate test models which contain model elements suitable for testing a model transformation engine. Such elements are those whose existence or properties are useful for testing particular aspects of a transformation.

In collaborative development paradigm, in contrast to model transformation, different versions of models are considered. Developers usually edit and modify models in graphical model editors. Typical model editors represent models as abstract syntax graphs in which, generally speaking, nodes represent model elements and edges represent relationships between elements (see Section 2.2). In such a graph, nodes and edges are typed and can contain attributes. The edit operations for such a representation are, in their lowest level, graph edit operations [Kehrer et al., 2013d,a], e.g. adding, deleting or modifying nodes and edges as well as their types or attributes. Although low-level edit operations are correct, they are difficult to comprehend by human developers who are used to working on higher abstraction levels. In addition to low-level graph edit operations, one can consider high-level edit operations which are more developer-friendly and

are usually found in graphical model editors [Kehrer et al., 2011]. A high-level operation can comprise of few to many low-level operations which can be regarded as a whole. For example, in UML class diagrams (see Section 2.1.4), if one deletes a class, all of its attributes and methods will be simultaneously deleted. Such high-level operations are also of special interest in development environments and for developers who are used to working in graphical model editors.

To support collaborative development in MDE, differences of models should be computed, possible conflicts should be resolved, and finally the conflicts should be merged into a new consolidated model. Therefore, typical existing model generation approaches which are motivated from the domain of model transformation are not quite useful for testing of model differencing and model merging approaches, since the generation of models is not based on edit operations (see Sections 3.1 and 3.2). To be useful in the direction of model differencing and model merging, a generator should resemble the natural process of editing of models in which a new version is obtained by modifying an existing model using properly defined edit operations (either low-level or high-level). Similarly, the same argument holds valid when we consider, model tracing, history analysis and difference presentation tools.

Moreover, if such a suitable generator exists, simply generating models based on random application of edit operations is of little value in many applications[1]. For instance, to test model differencing and model merging approaches, kinds, frequencies and order of the applied edit operations as well as where they are applied, are important factors. In this regard, it is quite essential that the generation process is finely under control and the generated test models resemble a realistic modification process that one observes in real world. Unfortunately, little is known about how real software models change and evolve over time and what are their characteristics or mathematical properties [Shariat Yazdi et al., 2013, 2014a]. Therefore, generating realistic test models which resemble the real world model evolution is a challenge [Shariat Yazdi et al., 2014b].

1.2 Research Goals and Contributions

With the sketch of situation presented earlier, we can concretely state the following main problems:

- In the domain of model differencing and model versioning, test models are quite scarce and the meager existing ones often lack desired properties.

[1] As a concrete example, we can mention the QuDiMo porject, Qualitätsoptimierte Differenzen für Modelle (Quality Optimized Differences for Models) [Pietsch and Shariat Yazdi, 2011], which was supported by the German Research Foundation (DFG). QuDiMo aimed at optimizing the existing model differencing algorithms in daily collaborative environments. Optimization needed realistic sets of benchmarks to be available. Such real standard benchmarks were proven to be impossible to obtain and therefore had to be artificially generated. Artificial benchmarks must be realistic in the sense that they must exhibit real (stochastic) properties one observes in models of real software systems. Considering design models of software systems, such properties have not been studied, mathematically formulated or regenerated.

- The existing model generation approaches are motivated from the domain of model transformation, and are not suitable for model differencing, model versioning tools, etc. They do not generate or modify models based on application of edit operations.

- To be useful, the generated test models should resemble realistic evolution (changes) one observes in models of real software systems.

- Little is known about how real software models evolve over time, which characteristics or mathematical properties they have and how one can mathematically formulate and simulate them.

In short, we can say that there is much need for artificial models which are realistic in the sense that they resemble true evolution of real software models.

To address the previous problems we propose a new general approach which generates realistic test models for model differencing, model versioning, history analysis tools, etc. In this regard, we propose a new model generator which generates models using edit operations. The generation process is finely under control. Moreover, the generator supports stochastic and other properties of interest in the generated models and can generate pairs or sequences of models.

To capture, analyze, mathematically formulate and simulate the evolution of design models, we propose a methodological framework. The evolution is captured in terms of applied edit operation between revisions of models and it is mathematically formulated using many statistical distributions as well as different time series models. Forecasting, simulation and generation of stochastically realistic test models are discussed in detail. To test the framework, we applied it to the evolution of design models obtained from sample a set of carefully selected Java systems.

More concretely, the contributions of this dissertation are as follows:

1. We present a model generator which generates models by application of (specified) edit operations. The models generated this way are appropriate for evaluating model differencing, model versioning, model tracing, history analysis tools etc. Contrary to existing approaches, the generator is finely configurable and the generation process is under control. This is achieved by providing different controlling mechanisms which will be described in detail later in Chapter 3. Generation of simple or complex structures is supported by providing appropriate low-level and high-level edit operations. More importantly, in contrast to the existing generators, our generator supports stochastic properties of interest in the generated models. This allows pairs or sequences of models which exhibit stochastic or other properties of interest, be finely generated.

2. In order to study real evolution and generate realistic test models, we captured the changes of typical real software systems at the level of their design models[2]. In this

[2] Simplified class diagrams are also used in the analysis and definition phase of software projects. In this work, we will always refer to the more detailed class diagrams as used in the design phase, which are translated into source code in many MDE methods.

regard, we used complete (revisions) history of nine typical real Java systems and we reverse engineered their design models by parsing their source code. The evolution (changes) was captured by counting occurrences of edit operations which were applied between revisions of design models. The investigated edit operations were not only low-level graph edit operations, but also high-level (developer-friendly) ones, defined on design models. In this way, we modeled the evolution at two abstraction levels. The first level is expressed in terms of 75 low-level graph edit operations and the second level considers the evolution in terms of 188 high-level (developer-friendly) edit operations. These two levels of abstraction are the basis for our analysis of evolution in this dissertation. Our approach can be used and adapted to study other types of models than design models.

3. To better understand the evolution of design models and to mathematically formulate it, we studied the evolution (changes) separately in terms of the applied low-level and high-level edit operations between revisions. For each edit operation, we considered sets of changes between subsequent revisions (pairwise evolution) and we employed many (sixty) statistical models, i.e. distributions, which seemed to be promising in modeling them. We showed that six distributions were able to model both the low-level and high-level changes with very good rates of success (often more than 90%). To regenerate the evolution in artificial models, one needs to know how random variates of these distributions are generated. We studied random variate generation methods of these distributions in detail and we provided the required theoretical background. For four of the distributions, there were no direct method for generation of their random variates. In this regard, we indirectly generated their random variates.

4. The historical evolutions of design models are also deeply studied. The historical evolution chronologically considers the sequence of changes between revisions. To analyze the historical evolution, we again considered the changes both in terms of low-level and high-level edit operations. We employed three categories of time series models. More precisely, we employed ARMA, GARCH and mixed ARMA-GARCH models in this regard. We studied and compared the performance of time series models, both to assess their suitability for capturing the stochastic properties of evolution, and to properly forecast future changes. We also studied how the proposed time series models are simulated, i.e. how valid sequences of them are reproduced, and how the initial conditions affect the generated sequences. The simulated sequences were used to generate histories of test models which have similar stochastic properties as real software systems.

Parts of the contributions of this dissertation which mentioned earlier was achieved as the result of effective teamwork. First, our model generator [Shariat Yazdi and Pietsch, 2011] was collaboratively developed by Hamed Shariat Yazdi, Pit Pietsch, Michaela Rindt, Tim Sollbach, Davy Franck Wanmeni, Thi Minh Hoa Nguyen, Petrissa Roth and Andre Bertels. However, detailed configuration and fine control over the generation process, support for stochastic properties and model histories, mathematical modeling

of the system etc. are the contribution of the author of this dissertation himself. Second, capturing the evolution of design models in terms of low-level and high-level changes is the result of collaborative work of Hamed Shariat Yazdi and Timo Kehrer. In this regard the author used the SiLift semantic lifting engine [Kehrer, 2015]. SiLift was developed under the MOCA project by Timo Keher, Kristopher Born et al., and was funded by the German Research Foundation (DFG) [Kelter et al., 2015].

The author (Hamed Shariat Yazdi) will use the "we" pronoun throughout this dissertation, but all of contributions and materials presented in this work, except the previously mentioned collaborative works, originated from the work of the author solely by himself.

1.3 Dissertation Structure

As we now know the contribution of this dissertation, the contents are organized as follows.

In Chapter 2, we provide the preliminaries and background of Model-Driven Engineering (MDE) and associated important concepts and materials. The beginning parts of this chapter is mainly suitable for new comers into MDE and briefly provides necessary information to understand and follow the presented research in this dissertation. In Section 2.1, we review what MDE is and how it is used. We cover the concepts of models, metamodels and abstraction levels in MDE. We see what we mean by model transformation, model differencing and model versioning. The readers who are familiar with MDE, can skip basic materials there. In Section 2.2, we see that models can be presented as typed graphs. Such a representation is suitable in computation of differences between models which is the topic of Section 2.3. As we told, in this dissertation, differences between revisions of design models are computed at low-level graph edit operations as well as high-level developer-friendly edit operations. How such differences are computed is discussed in detail in Section 2.4 by providing illustrative examples.

Our proposed model generator is presented in Chapter 3. In this regard in Section 3.1, based on the reviewed literature, we propose fundamental criteria that a generator must meet. We reviewed the existing approaches in the field of model generation in Section 3.2, and we investigated whether the existing generators meet our criteria. We show that existing generators do not have fine control mechanisms, they do not support stochastic or other properties of interest in generated models and none are suitable for generation of model histories. In Section 3.3, we present our model generator and its usage scenarios. Moreover, we discuss how models are generated and how edit operations are performed on models. Section 3.4 presents the controlling mechanisms of the generator. There, we discuss how specified properties of interest, such as stochastic properties, are created in the generated models. The chapter ends with an evaluation of the generator in Section 3.5.

Chapter 4 discusses how evolution of software design models should be captured. In Section 4.1, we show that evolution of design models cannot be properly expressed in terms of changes in values of static software metrics. Instead, we will see that evolution

is more precisely expressed in terms of applied edit operations between revisions. In Section 4.2, we introduce a difference derivation pipeline which can properly capture the evolution in terms of applied edit operations. There, we consider low-level graph edit operations as well as high-level developer-friendly operations. Based on these two sets of edit operations, we model the evolution at two abstraction levels. To capture the evolution of design models, we show how our approach is applied to Java systems in Section 4.3. There, we provide our metamodel of Java design models. Providing an explanatory example in Java, we show how changes between design models are computed. Considering low-level edit operations on our metamodel, we show that the evolution of Java design models can be expressed in terms of 75 low-level difference metrics. Employing high-level edit operations, we express the evolution in terms of 188 high-level difference metrics. The sets of computed low-level and high-level difference metrics are the data sets of our analyses in later chapters, where we mathematically model the evolution. How our sample sets of real Java systems are selected for the analyses, is discussed in Section 4.4.

Chapter 5 is devoted to mathematically model and simulate the design model evolutions of Java systems. In this chapter we study the evolution as the amount of changes between subsequent revisions of models (pairwise evolution), without considering chronology of changes. In Section 5.1, we show that histograms of changes are typically skewed with heavy tails. There, we introduce six statistical models (distributions) which were the most successful ones from our long list of sixty candidates. In Section 5.2, we present the results of our analyses on the sets of low-level and high-level difference metrics. We show that the proposed distributions are quite successful in modeling different low-level and high-level metrics with very good success rates. Therefore, they can be employed to model the evolution. To simulate the evolution, one needs to properly generate random variates of the proposed distributions. Section 5.3 is devoted to random variate generation of the distributions. There, we provide the required algorithms in detail. For four of the proposed distributions, there is no tailored algorithm for generating their random variates. Therefore, we indirectly generated their random variates. We employed statistical properties of the distributions in order to generate their random variates. Finally, the related works are presented in Section 5.4.

In Chapter 6, we study the chronological properties of the design model evolution. As in the previous chapter, we consider the evolution separately based on low-level and high-level difference metrics. To chronologically model the evolution, we employed time series models which intrinsically consider time dependency properties of changes. Such properties are quite important when histories of test models should be generated. The histories of test models should resemble the time-dependent properties of real software models. To this aim, we provide the mathematical foundation of our approach in Section 6.1. There, we consider ARMA, GARCH and mixed ARMA-GARCH models, and we discuss how and why such models are suitable. We provide detailed methodology of those models and we discuss their forecasts' accuracies. In Section 6.2, we model the evolution of design models using our proposed time series models. We show that chronological evolution can be modeled by ARMA and mixed ARMA-GARCH models. In Section 6.3,

we compare the two successful categories of models, i.e. ARMA against mixed ARMA-GARCH. In this regard, we study which one better handles the dynamic characteristics of changes and which one better forecasts the future changes. To generate more realistic model histories, we need to simulate sequences of proposed time series models. In Section 6.4, we study the simulation aspects of the proposed time series models and we show how such sequences are used in our model generator to generate histories. The chapter closes with a threat to the validity analysis and a review of the related works in Sections 6.5 and 6.6 respectively.

The dissertation ends with a conclusion and an outlook in Chapter 7.

1.4 Publications Associated to this Dissertation

There are currently eight publications associated to this work. Two of them are submitted to renowned journals (one is in the review process). Others are submitted to either renowned conferences or noted workshops. The fifth publication in our list, i.e. [Shariat Yazdi et al., 2013], received the Best Paper Award of Software Engineering Conference of 2013 (SE2013) in Germany.

1. Hamed Shariat Yazdi, Lefteris Angelis, Timo Kehrer and Udo Kelter. *A Time Series Framework for Simulating the Evolution of Software Design Models Towards Generating Realistic Test Models*, [Submitted to a Journal], 2015.

2. Hamed Shariat Yazdi, Mahnaz Mirbolouki, Pit Pietsch, Timo Kehrer and Udo Kelter. *Analysis and Prediction of Design Model Evolution Using Time Series.* Workshops of the 26th International Conference on Advanced Information Systems Engineering - (CAiSE 2014 Workshop), pages 1-15, Thessaloniki, Greece, June 2014 ([Shariat Yazdi et al., 2014a]).

3. Hamed Shariat Yazdi, Pit Pietsch, Timo Kehrer and Udo Kelter. *Synthesizing Realistic Test Models.* Journal of Computer Science - Research and Development, pages 1-23, Springer, 2014 ([Shariat Yazdi et al., 2014b]).

4. Timo Kehrer, Pit Pietsch, Hamed Shariat Yazdi and Udo Kelter. *Detection of High-Level Changes in Evolving Java Software.* Softwaretechnik-Trends, volume 33, number 2, 2013 ([Kehrer et al., 2013c]).

5. Hamed Shariat Yazdi, Pit Pietsch, Timo Kehrer and Udo Kelter. *Statistical Analysis of Changes for Synthesizing Realistic Test Models.* Best Paper Award of SE2013, In Multi-conference Software Engineering 2013 (SE2013), Gesellschaft für Informatik (GI), pages 225-238, Aachen, Germany, 2013 ([Shariat Yazdi et al., 2013]).

6. Pit Pietsch, Hamed Shariat Yazdi, Timo Kehrer and Udo Kelter. *Assessing the Quality of Model Differencing Engines.* In Comparison and Versioning of Software Models 2012 (CVSM2012), Essen, Germany, 2012 ([Pietsch et al., 2012b]).

7. Pit Pietsch, Hamed Shariat Yazdi and Udo Kelter. *Controlled Generation of Models with Defined Properties.* In Software Engineering Conference 2012 (SE2012), pages 95-106, Berlin, Germany, 2012 ([Pietsch et al., 2012a]).

8. Pit Pietsch, Hamed Shariat Yazdi and Udo Kelter. *Generating Realistic Test Models for Model Processing Tools.* In Proceedings of the 26th IEEE and ACM International Conference on Automated Software Engineering (ASE2011), pages 620-623, Lawrence, KA, USA, 2011 ([Pietsch et al., 2011]).

CHAPTER 2

Preliminaries and Background

The purpose of this chapter is to provide the preliminaries and the required background for this research. In this chapter we focus on relevant material from Model-Driven Engineering (MDE). In this regard, we give an overview of MDE by providing the definitions and describing the necessary concepts and techniques. The presented materials in this chapter is not intended to be either exhaustive or comprehensive. MDE is a mature field with enormous research materials, technologies and tools. There have been numerous books and papers dedicated to each particular aspect of MDE which is out of scope of this dissertation. In this chapter we just concentrate on fundamental requirements which are necessary for this research. The beginning parts of this chapter is purposely written to provide the basic concepts for newcomers to MDE.

The chapter is organized as follows. Section 2.1 provides a glimpse into MDE by providing basic definitions and concepts. We review what a model means and which standards and technologies are available in MDE. We also review model transformation, model versioning and the associated concepts. Section 2.2 explains how models can be principally regarded as graphs in the mathematical sense. There, we review that what are low-level and high-level differences between models and how they relate to the graph representation of models. Once the relationship of models and graphs are explained, Section 2.3 clarifies how models can be compared and their differences can be computed. In the end, in Section 2.4, we explain two tools, SiDiff and SiLift, which we use in this research for computing the low-level and high-level changes respectively.

2.1 A Glimpse into the Model-Driven Engineering World

Since the advent of computer, historically, the way we develop our programs on computers have been changed considerably [Ameller, 2009]. The old days were mostly dominated by the low-level programming languages which were near to the machine language, and were mostly restricted to machine specifications. By the invent of compilers, those were

promoted to high-level programming languages which are usually referred as the third level programming languages[1] e.g. C or Fortran. The high-level programming languages elevated the abstraction levels, eliminated platform-specific considerations and delivered greater flexibility. Developing object-oriented programming features was another step toward more abstraction in this regard. Although the newer technologies in third generation programming languages, e.g. Java EE[2] and .NET, are used widely in practice, there are still many problems which limits their usability. Fast growth of the complexity of a platform which is hard to handle by such languages, and optimally deploying a large-scale system with too many components are two examples of these problems [Schmidt, 2006]. Moreover, using such languages to model concepts of a specific domain was tedious and inefficient.

The move toward more abstraction has continued by the *domain specific languages* (DSL) [Fowler and Parsons, 2010] which are usually referred to as fourth generation languages [Selic, 2008]. The main aim of DSL is providing more productivity by delivering faster and more robust solutions for the domain of interest by providing tailored and customized notions, tools, constructs and algorithms for that domain [Mernik et al., 2005]. Moreover, DSL also simplify and increase the involvement of non-programmer domain experts and promote the quality of solutions and products [Voelter and et al, 2013]. The Structured Query Language (SQL) for data base queries, Matlab [MathWorks Company, 2013] and Mathematica [Wolfram Research Inc., 2014] languages for modeling and mathematical analyses are typical good examples of DSL. Although DSL have many benefits, they also have some drawbacks, for instance their development is hard which requires both the domain and the programming knowledge, causing higher costs [Mernik et al., 2005].

Model-Driven Engineering (MDE) [Stahl et al., 2006] can be considered as the latest approach in the continuous effort for supporting more abstraction. As the title suggests, models are the most centric artifacts in the development process instead of source programs [Selic, 2003]. Models can reflect real world concepts and entities in a simplified manner. They can also be formal specification of a system in a higher abstraction level.

MDE promises to be the solution to the inability of the third generation programming languages to cope with the increasing complexity of systems and to their ineffectiveness with regard to capturing and modeling the domain knowledge [Schmidt, 2006]. Specified models provide more flexibility in the sense that the design of the systems or solutions are independent of the underlying technologies or implementations. Therefore understating, modification and maintenance of specified models can be done more efficiently [Selic, 2003]. MDE brings the benefits of automation in software engineering and boosts productivity [Selic, 2003]. The long time productivity will be also improved by capturing the system through higher level models which are less sensitive to the personnel changes, requirement changes and variation in development or deployment platforms [Atkinson and Kuhne, 2003].

[1] The machine and the assembly languages are usually referred to as the first and the second generations respectively [Ameller, 2009].

[2] Java Enterprise Edition, formerly known as J2EE.

Furthermore, MDE models are not just used for documentation of a system, rather code can be automatically generated out of them. For this purpose, high-level models of a system are typically transformed into platform-specific models which in the end are transformed into executable code. This process requires model transformation approaches and technologies which allow such features. Model transformation is the key element in MDE [Sendall and Kozaczynski, 2003] and we will cover it in more detail in Section 2.1.5.

Ambiguity of Concepts Although MDE aims and goals are more or less well known in the literature, the term itself seems to be mixed up with *Model-Driven Development* (MDD) and *Model-Driven Architecture* (MDA) and they are more or less interchangeably used. "Model-Driven Development is simply the notion that we can construct a model of a system that we can then transform into the real thing." [Mellor et al., 2003]. This definition is quite broad and the development in the third generation languages can be typically regarded as MDD [Ameller, 2009]. MDA is a set of standards from the Object Management Group (OMG) in the field of MDE. OMG is a nonprofit international organization which is formed by many[3] academic institutions, companies, vendors and end-users with the aim of defining global technological standards[4].

The MDA intention is to separate the specification of a system and its functionality from the implementation aspects on a specific platform, letting the specification and functionalists (*a*) to evolve separately from technological changes in that platform and (*b*) to be easily deployed to other platforms [OMG, 2001]. "The MDA set of standards includes the representation and exchange of models in a variety of modeling languages, the transformation of models, the production of stakeholder documentation, and the execution of models. MDA models can represent systems at any level of abstraction or from different viewpoints, ranging from enterprise architectures to technology implementations" [OMG, 2014]. Regarding MDE, "It is important to note that MDE does not define any particular approach; instead, MDE itself is a paradigm that is addressed by [different] approaches, while being independent of language or technology" [Saraiva, 2013]. This gives a broader meaning to MDE comparing to the MDA which is a specific standard offered by OMG. According to [Ameller, 2009], we can therefore consider this loose mathematical relationship between them: MDA $\subset$ MDD $\subset$ MDE, although they are interchangeably used in the literature as mentioned before. In the next section, we give more information about MDA which is a widely used accepted standard.

2.1.1 Model-Driven Architecture

As we discussed earlier, MDA is the OMG standard and approach to MDE. It includes associated technologies and techniques which address those standards, such as UML, mapping functions, executable models etc. MDA is aimed to support the representation of a system, using linked models, at any abstraction level from different viewpoints

[3] At the time of writing this dissertation, more than 280 members `http://www.omg.org/`.
[4] Including: Unified Modeling Language (UML) and XML Metadata Interchange (XMI).

[Mellor et al., 2004]. The models used in MDA can be used for modeling real world entities, system specifications as well as technological artifacts [OMG, 2014]. Moreover, MDA is also very concerned in transformation of models to other models, each capturing different aspects of the system with a different level of abstraction, expressed by possibly different languages.

In the rapidly changing world, business logic and software technologies are independently and quickly evolved and developed. Thus software solutions should be developed in a way that the business solution is quite independent of the implementation platform [Stahl et al., 2006]. In this regard, MDA in its core is formed by three levels of Platform-Independent Models, Platform-Specific Models and the source code [Kleppe et al., 2003]. "[A *Platform-Independent Model* (PIM) is] formal specifications of the structure and function of the system that abstracts away technical details" [OMG, 2001]. It does not consider or include any detail or technical issues of a specific platform such as the implementation language or database technologies. This gives the flexibility of defining and modeling a system independent of any technological issues or implementation specifications. A *Platform-Specific Model* (PSM) is defined in terms of, and bounded to a specific platform [OMG, 2014]. A PSM will use the concepts and mechanisms of the target platform. "A *platform* is a software infrastructure implemented with a specific technology (Unix platform, CORBA platform, Windows platform) on specified hardware technology" [OMG, 2001]. The concept of platform is therefore highly dependent on the context, and a PSM is bounded to the specification and characteristics of the associated platform [Brown et al., 2005].

As mentioned above, separation of PIM as the overall design of the system from PSM which is bounded to technological specifications, can address the parallel evolution of business logic and the runtime/execution platforms. This requires that a PIM can be transformed into a PSM and the PSM is then translated into the executable code. The business solution can be developed independently of any technological or implementation considerations, and then it is deployed into different platforms, e.g. Linux or Windows using Java or C#, employing different database technologies. We cover the transformation of models in more detail in Section 2.1.5.

2.1.2 Models and Meta-Models

As we discussed in Section 2.1, models are the primary artifacts in the MDE world. In MDE, a model might represent a real world entity or a system as in other fields of science or engineering, although it is not only limited to that. In the MDA approach, a *model* is a selective set of information which represent some aspects of the system such as structure, behavior etc. [OMG, 2001, 2014]. The model is then somehow related to the systems either directly or implicitly. A model can be regarded as a set of statements about the system which could be evaluated as true or false and when all of the statements are true, we can regard the model to be correct [Seidewitz, 2003]. There are different benefits, gained by using models in the MDA approach [OMG, 2014, Brown et al., 2005]:

- Models can be used to facilitate team work through well defined terms and nota-

tions, managed and shared semantics of the systems as well as different libraries, rules and reusable processes.

- Automated transformation of the models will help to derive new artifacts and new implementations in a consistent manner. This will increase productivity by reducing the cost of realization, design and maintenance efforts.
- Analytics, statistics and various data analyses can be done on data and semantics which are captured in models.
- Models can be executed and simulations can be performed by realizing the design with minimum technical details.
- Different information such as acquisition of specifications and processes as well as documentation can be derived from models.
- Models provide a structural presentation of knowledge for unstructured information.

Although the above list provides the benefits of using models, the following key characteristics should be satisfactorily met in order a model to be useful in practice [Selic, 2003]:

- *Abstraction.* The model should be a proper abstraction of the system, removing the unnecessary details with respect to the view point of interest.
- *Understandability.* The model should be understandable by hiding unnecessary details and by conveying the complexity of the system with much little information.
- *Accuracy.* This characteristic means that a model should correctly reflect the interesting features or views of the system.
- *Predictiveness.* This means that the model can be correctly used to predict the interesting features of the system either through experiments or some kind of formal analysis.
- *Inexpensiveness.* The constructed model should be more affordable to build and analyzed than the original system.

The above requirements impose that the models should be expressed in a consistent manner. Models which consists of many elements should be defined using a well defined language. The language should state if an element is legitimate within the model. It should also clearly describe the relationship of the model elements with respect to each other. Moreover, it should be suitable for computer processing [Kleppe et al., 2003]. The language should consist of syntax, semantics, notions, terms and structures which are well defined and understood by domain experts and stakeholders [OMG, 2014, Mellor et al., 2003]. Such a language is referred to as a *metamodel*. We say that a model *conforms* to a metamodel when all elements and relationships are legitimate based on

the rules in the metamodel [OMG, 2014]. The relationship of a metamodel to a model can be regarded as the class-instance relationship in which every model can be regarded as an instance of the metamodel [Stahl et al., 2006]. Metamodels define a clear and an unambiguous way of describing the semantics of the models, allowing automation, model transformation as well as analysis [Brown et al., 2005].

2.1.3 Abstraction Layers and Meta Object Facility

As we discussed in Section 2.1.2, a metamodel describes a modeling language, its semantics and syntax. A metamodel is principally a model itself, this requires that the metamodel should also be described by another metamodel, say meta-metamodel. Theoretically, this concept can be expanded in a similar way, but it can not be handled in practice. Therefore, the model to metamodel relationship is quite relative and a model can be a metamodel in the hierarchy [Stahl et al., 2006]. To be practical, OMG has defined four hierarchical layers of modeling which are denoted by M0, M1, M2 and M3, depicted in Figure 2.1 [Atkinson and Kuhne, 2003].

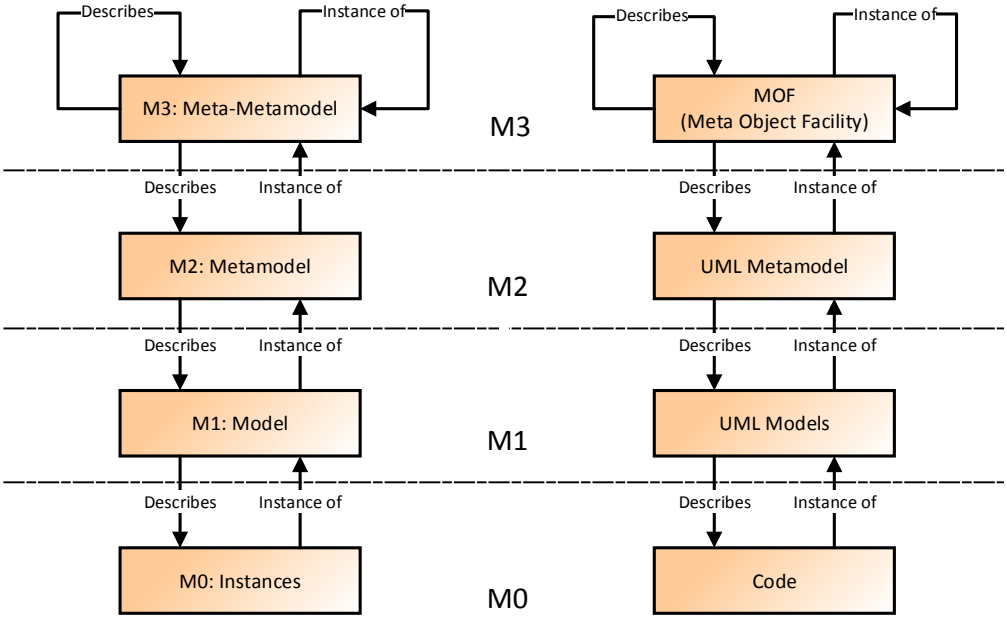


Figure 2.1: The OMG four layer of abstraction in modeling (Adapted from [Ameller, 2009, Stahl et al., 2006]).

Each layer in the hierarchy is an instance of the immediate level above it except the topmost level. At the lowest level, i.e. M0, the *user data* exists which should be processed, handled or modeled by the application, e.g. information in a database or instances of an object-oriented system at run-time [Mellor et al., 2004]. The M1 level holds the models of the user data (user models), the concept at M1 level classifies the instances of M0 level and elements in the M0 level are all valid and feasible instances of the M1 level

[Kleppe et al., 2003]. M1 for instance, can hold the classes of an object-oriented system or tables in a rational database [Mellor et al., 2004]. The M2 level holds the metamodel of the models in the M1 level which is a well defined language that describes the models. As an example, UML Classes and Attributes are defined in the M2 level while instances of them are existing in the M1 level. Finally, M3 is the topmost level, usually referred to as meta-metamodel, which describes the metamodel, semantic and relationships of its elements.

According to OMG standards, the M3 level is designed to be reflexive, i.e. it is described by its own, and is called *Meta Object Facility* (MOF) [OMG, 2005a]. MOF is a standard language which is used to define all other languages and metamodels in MDA [Kleppe et al., 2003]. It clarifies how sets of metamodels should be formed in order to properly define the models of interest. MOF provides semantics which allow various technological mappings. It is a common framework where different technologies can be combined together [Frankel, 2003]. The MOF is also designed to be not only small, but also extendable by the means of inheritance and composition in order to richly support other constructs [OMG, 2005a].

The *Eclipse Modeling Framework* (EMF) is the most famous implementation of MOF. The earlier implementation of EMF employed a subset of MOF which was the most relevant for practical issues. Due to this effort, in the newer OMG standards for MOF, that subset of MOF was classified as Essential MOF or simply EMOF [Stahl et al., 2006]. EMF allows the modeling be practically done through graphical tools and code generation facility. EMF principally unifies Java, XML and UML into a practical framework for code generation. The developers can define their models in any of the aforementioned languages and the other two as well as the implementation classes are generated automatically through EMF [Steinberg et al., 2009]. The meta-metamodel of EMF is referred to as *Ecore* which is self describing and is used to define models in EMF. In EMF, models are used to generate Java code and they can be serialized using XMI standard (see Section 2.1.4).

2.1.4 UML, XMI and OCL

As we discussed in Section 2.1.3, MOF is an OMG standard and language to define the metamodels. MOF is not the only standard of OMG, there are other standards such as Unified Modeling Language (UML), and XML[5] Metadata Interchange (XMI) as well as Object Constraint Language (OCL) which are used frequently in MDA. Here we review them quickly as we refer to them in this dissertation.

UML "The *Unified Modeling Language*, is a general-purpose modeling language with a semantic specification, a graphical notation, an interchange format, and a repository query interface" [OMG, 2005b]. Historically, UML has emerged in the 90's as the successor of object-oriented design and analysis and later standardized by OMG [Fowler, 2004, Frankel, 2003]. UML is the most famous modeling language which is located at the M2

[5] Extensible Markup Language.

level [Kleppe et al., 2003]. The UML metamodel is expressed by MOF which describes the structural and behavioral characteristics of UML [Mellor et al., 2004]. The latest version of the UML has 13 types of diagrams as depicted in Figure 2.2 [OMG, 2012b]. The diagrams are typically divided into two categories of structural and behavioral diagrams. The structural diagrams can be used to model the static structure of objects in a system which are not changing with time while the behavioral diagrams capture the dynamic, i.e. time variant, behaviors of objects in a system, e.g. activities. Class diagrams can be considered among the most popular UML diagrams in the domain of software engineering.

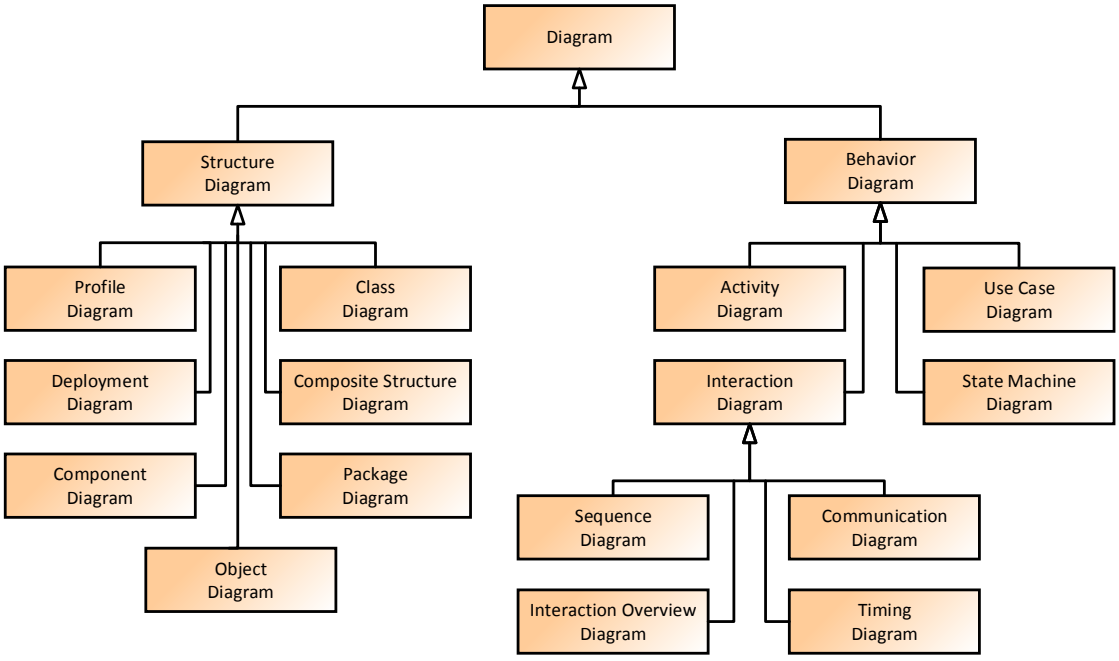


Figure 2.2: Taxonomy of UML diagrams (Adapted from [OMG, 2012b]).

XMI The *XML Metadata Interchange* is another OMG standard for data access which is used for interoperability of metamodels and models between different domains [Mellor et al., 2004]. It allows the definition, interchange and manipulation of different models and metamodels which are conforming to MOF. XMI is based on XML which allows a model to be encoded in an XML document. The advantages of XMI are that, it is (a) based on XML which supports many document types (b) independent of any platform and (c) well documented and has very good tool support [Alanen and Porres, 2005]. Although the platform independence of XMI is an advantage, in some cases it has shortcomings. For example, if two tools are encoding a model in XMI but they use two different versions of UML diagrams, the interchange is not possible [Alanen and Porres, 2014].

OCL The *Object Constraint Language* is a textual modeling language which is bounded to UML models and can add constraints on models and queries regarding models [OMG, 2012a]. Such constraints rely on types of the models (defined in UML) and they enforce data integrity [Warmer and Kleppe, 2003]. OCL is just a pure specification language. In other words OCL constraints have no influence on current state of models, i.e. they can be evaluated but cannot alter the existing state of models directly [OMG, 2012a]. Moreover, OCL is a typed language where each expression conforms to a type and it is not possible to compare expressions with different types, e.g. an integer with a string. As an example of an OCL constraint, a class which contains information about a product requires that its ID is positive. As another example, the class which is used to model a bank transaction for selling that product, requires to check whether the bank account of the seller and the buyer are different. Such constraints which clarify the relationship and expected behavior of the models, cannot be handled by class diagrams alone and are expressed by OCL constraints.

2.1.5 Model Transformations and QVT

As we discussed in Section 2.1.1, MDA is essentially formed by three elements of PIM, PSM and code in its core. A PIM is independent of any platform and models the overall functionality and specification of the system without considering any issues regarding the implementation. Conversely, a PSM is bounded to technical and specific aspects of the platform of interest. The aim of designing a system as a PIM is to increase the productivity and handling the challenges and changes in the real business system by disregarding technical details of the implementation platform [Sendall and Kozaczynski, 2003]. This approach will reduce the time to the market, can use the domain expert knowledge more efficiently by eliminating the implementation considerations, and finally provides grater flexibility by allowing PIMs to be portable.

Although we mentioned that PIM increases productivity, without any implementation they are not fully useful. A PIM should be able to be transformed into code to be executable. To this end, a PIM should first be transformed into a PSM which later, through another transformation, is transformed into executable code on the specified platform [Kleppe et al., 2003]. Transformations play a crucial role in MDE, allowing the realization of a metamodel, i.e. models of a metamodel, be transformed to models conforming to another metamodel. Formally, "a *transformation* is the automatic generation of a target model from a source model, according to a transformation definition. A *transformation definition* is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language. A *transformation rule* is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language" [Kleppe et al., 2003]. Figure 2.3 shows the basic concept of a model transformation [Czarnecki and Helsen, 2006].

Transformation of a model to another one requires that the semantics and syntax of two models are clearly described by the metamodels [Sendall and Kozaczynski, 2003, Mellor et al., 2004]. The tools for transformations, usually allow transformations by:

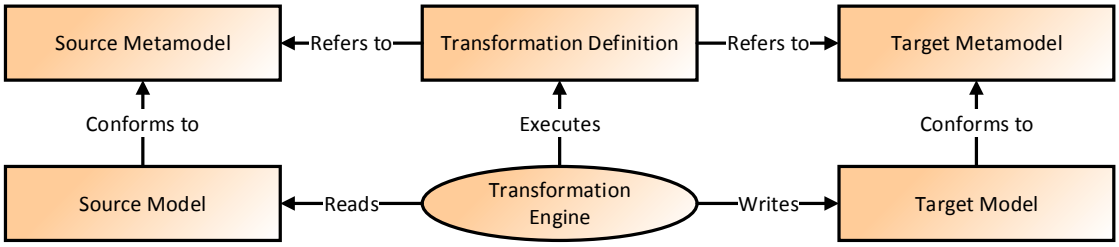


Figure 2.3: Model transformation (Adapted from [Czarnecki and Helsen, 2006]).

(*a*) direct manipulation of models (*b*) transformation via another tool by exporting the model in an intermediate representation, like XMI standard, and (*c*) a transformation language which supports different constructs and rules [Sendall and Kozaczynski, 2003].

Since the programs are also modeling the system at the code level, they can be regarded as models too and model transformations will be considered not only from an abstract model to another abstract one, but also to more concrete ones at the code level [Mens and Van Gorp, 2006, Czarnecki and Helsen, 2006]. Generally, the model transformation approaches can be categorized into model to model (M2M) and model to code (M2C) approaches and M2C can be regarded as a special case of M2M provided that the metamodel of the target language is available [Czarnecki and Helsen, 2003]. The generated code of from a M2C is then used by the associated compiler to be translated into executable program.

If the transformation is performed between two models which are specified in the same modeling language (metamodel) then it is called an *endogenous transformation*, also known as *rephrasing*, otherwise it is an *exogenous transformation* which is also known as *translation*. Transformation of high-level abstract models into concrete ones, e.g. code generation, is of the translation type. Refactoring and optimization of software models for improving their internal structure or their operational qualities are examples of rephrasing.

The other orthogonal classification for transformations is based on the fact that source and target models reside at the same or at different abstraction levels. If both lay on the same level, it is refereed to as a *horizontal transformation*, otherwise it is called a *vertical transformation* [Mens and Van Gorp, 2006, Sendall and Kozaczynski, 2003]. Refactoring of models can also be regarded as a horizontal transformation since it does not affect the abstraction level of models. Refinements, where in each step the model is refined by adding more implementation-oriented issues, can be regarded as a vertical transformation. Table 2.1 shows these two orthogonal classifications.

QVT *Query / View / Transformation* (QVT) is the OMG standard for the transformation of models whose languages are defined with MOF [OMG, 2011]. QVT consists of languages for creating views on the models, querying the models and for writing the model transformations [Kleppe et al., 2003]. A *view* is a particular perspective of the

	Horizontal	Vertical
Endogenous	*Refactoring*	*Formal refinement*
Exogenous	*Refactoring*	*Code generation*

Table 2.1: Orthogonal classifications of model transformations with examples (Adapted from [Mens and Van Gorp, 2006]).

system which is modeled or is a model which is derived from the original model by considering some aspects or elements of interests [Brown, 2004, OMG, 2001].

2.1.6 Model Versioning

The collaborative development of software systems, traditionally, has been highly dependent on *version control systems* (VCS) which allow team development work by (*a*) keeping the log of changes on software artifacts (mostly codes), (*b*) concurrent development, (*c*) branching support, (*d*) attached annotations and comments on the performed changes and (*e*) distributed development by eliminating physical location barrier.

There are two types of VCS, *centralized version control systems* (CVCS) and *distributed version control systems* (DVCS) [Otte, 2009]. In a CVCS there is a central repository which is basically a server that stores histories of software artifacts which are known as revisions. A *revision* is a snapshot of the artifact in time which is additionally labeled with a revision number (typically an integer) and some metadata including the comments on the revision. Typically, a user first checks out the latest or a specific revision out of the repository as a local copy, modifies it and commits it back to the repository. For such activity, the user needs a network connection. In contrast, in a DVCS there is no central repository and each user has a complete local repository which can deploy on the network as a remote repository, letting others to clone it. Once a new user clones it, he/she can develop further, and meanwhile fetches new changes from the first user, or pushes his/her changes to the first repository. Alternatively, the new user can publish his/her own repository as a completely standalone remote repository and a similar story happens again to this new repository in which other interested users might clone and use it. Subversion (SVN) and Git are two typical examples of CVCS and DVCS respectively [Pilato et al., 2008, Loeliger and McCullough, 2012].

Since both CVCS and DVCS support parallel development paradigm, artifacts are highly probable to be modified and touched by different developers in parallel, which often cause conflicts. One solution is to avoid any conflict by locking the artifact until the modification is over by the user, and freeing it for further modification by others. The locking approach has the drawback that other users have to wait till the artifact is released which causes delay in collaborative work specially when the user forgets or, e.g. due to unexpected connection problems, is unable to unlock the artifact. Such approach is usually referred to as the pessimistic approach which is frequent in CVCS. The optimistic approach allows a copy of the artifact to be modified, and then the

changes are merged with another state of the artifact caused by another modification. Therefore, it is prerequisite that the concurrent changes on artifacts are managed and the conflicts are appropriately resolved [Brosch, 2012]. Typically, such a resolution requires that different changes are merged together in a new artifact [Barrett et al., 2008]. The merge process requires that two artifacts are first compared, conflicts are detected and resolved, and finally they are merged into a new artifact [Altmanninger et al., 2009].

Since the focus of the development has shifted from code to models in MDE, it is quite essential that models are also supported by VCS in order to flourish the collaborative work. Such VCS should allow concurrent modification of models by different developers and should support the collaborative development by merging the performed changes into a consolidated final model [Brosch et al., 2010, 2012]. Two dimensions can be considered for merging of the software artifacts [Brosch et al., 2012]. The first dimension is that how the artifacts are presented. Text based representations and graph based representations, which implicitly includes tree based representations, are the most common approaches. The second dimension, which is orthogonal to the first one, is that how changes are identified, represented and finally merged.

Regarding the first dimension, unfortunately, text based versioning systems are not able to handle models appropriately even if they are represented by a textual format like XMI [Kelter et al., 2005]. Such approaches do not consider the semantic and syntax of models, therefore text based merging leads to invalid or awkward results [Barrett et al., 2008]. For instance, when two identical models are stored in different order in their textual presentation, a so-called "pseudo difference" is reported. The only advantages of text based merging are their speed and simplicity. The other approach is to use the graph based merging algorithms which principally considers the software artifacts as a structured presentation, e.g. trees or graphs, and then compares and merges them [Brosch et al., 2012]. Such approaches have more power in detection and merging of changes and can detect both the syntactic and semantic changes [Mens, 2002].

Regarding the second dimension, i.e. how changes are identified and merged, the two possibilities are: state based and operation based merging [Brosch et al., 2012]. In a *state based merging*, two states of a model, i.e. versions, are compared, differences between them are computed, and finally they are merged together into a new model. The comparison of two states requires that a *matching function* first determines which elements in two states can be regarded as *corresponding*, i.e. presenting the same entity in two models, and then based on this information the changes and conflicts are computed and resolved. Identical elements are ideal for establishing a correspondence, although in practice more advanced techniques are used to establish the correspondence for non-identical elements which are similar and can be regarded as corresponding. The accuracy of the matching function has a direct influence on the merging quality afterward. The matching of elements in two models is principally a graph isomorphism problem which is shown to be NP hard [Khuller and Raghavachari, 1996, Jungnickel, 2006]. In the case that non-identical elements are matched, fine grained differences between the correspondent elements are calculated by a *difference function* which is later used to detect and resolve conflicts [Brosch et al., 2012]. For an *operation based merging*, the applied

sequences of operations to the models are important rather their states. Such operations are recorded by the editor and they usually comprise of atomic as well as composite operations. The recorded operations are then purified for removing invalid or unnecessary operations, e.g. an update operation on a model element which will be later removed. Later the purified operations are used in the merging step. The operation based merging generally has a better performance comparing to the state based merging regarding the time and the accuracy, although it is tightly dependent to the corresponding editor.

Regarding the merging, there are three possibilities for merging two states of an artifact. Suppose that a base model M_0 is checked out from a repository and is modified independently in parallel by two developers. If the applied sets of changes are denoted by C_1 and C_2 respectively, M_1 is the first version which is obtained by application of C_1 to M_0 and M_2 is the second version which is obtained by application of C_2. Without loosing the generality, assume that the commit time of these two models are t_1 and t_2 respectively and $t_1 < t_2$. Suppose that at the time of t_2, the commit will trigger a conflict between versions M_1 and M_2 which should be resolved and the applied changes should be merged. There are three possibilities for merging the applied changes in M_1 and M_2: raw merging, two way merging and three way merging [Mens, 2002, Altmanninger et al., 2009].

A *raw merging* will apply the changes C_1 to the M_0 in order to obtain M_1 and then applies C_2 to M_1 to obtain the merged version of M_3 [Barrett et al., 2008]. This process might fail since C_2 might need something which C_1 has already modified. No reordering of changes in sets C_1 and C_2 is devised in raw merging. A *two way merging* just considers the differences of M_1 and M_2 and it disregards the base model M_0. The algorithm then tries to merge changes in sets C_1 and C_2 in a proper way to obtain the merged model of M_3. Disregarding the base model of M_0 in the two way merging approach has the drawback that it is not possible to determine if an element is added e.g. in M_1 or deleted in M_2, or if an element is modified in one version or in both [Altmanninger et al., 2009]. As its name suggests, a *three way merging* additionally takes the base model of M_0 into account. By comparing the base model with the two versions, it is possible to detect more changes and conflicts comparing to the two way merging [Brosch et al., 2012].

Summing up our discussions, model versioning is an expanding domain in collaborative development in the field of MDE and current VCS should satisfactorily support model merging. The graph based merging algorithms are superior to their text based counterparts, since the syntactic and the semantics conflicts can be detected and resolved more efficiently. For graph representation of models, the state based merging approaches usually have to first establish a correspondence relationship between elements of two models which can be regarded as the same entity. Then, a difference function computes the changes, and the conflicts in the computed changes are finally resolved and used in the merging step. Lastly, the three way merging is more robust in detecting the changes and resolving the conflicts than a two way merging.

2.2 Graph Representation of Models

Since the models and metamodels defined by OMG standards have typically elements which are connected to each others through connections, it is a common practice to represent models and metamodels using graphs. Such a graph representation of a model is a typed attributed graph which is known as *abstract syntax graph* (ASG) [Kehrer et al., 2013d,a]. The elements in the metamodel of the given model, define the types of the nodes and edges in the ASG presentation of a model. Moreover, the metamodel defines the well-formedness of models with respect to the legitimate structure and permissible values of the attributes e.g. multiplicity constraints. The following definition clarifies how models are represented as graphs.

Definition 2.1 (Graph Representation of a Model [Wenzel, 2010])**.** Let M be a model, the directed graph representation of M is the ordered list of $G = (V, E, T_V, T_E)$ in which $V = \{v_1, v_2, \ldots, v_n\}$ is the set of typed vertices representing the elements of M, $E = \{e_1, e_2, \ldots, e_m\} \subseteq V \times V \times T_E$ is the set of typed directed edges which express the relationships among the model elements and, T_V, T_E are the sets of types for vertices and edges respectively, where $T_V \cap T_E = \varnothing$.

Principally, the model elements and their relationships are modified by the developers, i.e. not only elements of a model are edited by the developers, but also the relationships between them are modified. For instance, the association or composition relationships as well as their corresponding multiplicities can be modified. This requires that both the elements and their relationships, which are depicted by edges in a model, should be represented as nodes in the graph representation. Moreover, such nodes in the graph representation should also have attributes which principally reflect the attributes of the corresponding elements or relationships in the model, e.g. name, visibility, multiplicity etc. Therefore, the graph representation of a model have nodes which are typed and attributed. In this way, it is legitimate to ask what is the type of a node in the graph representation and what are its attributes. Figure 2.4 depicts the above representation [Wenzel et al., 2007].

Although the previous discussion sketches the big picture for representing a model as a graph, the mapping of model elements to graphs is not unique and should be defined based on the semantics of the metamodel [Wenzel, 2010]. Since model elements may contain other model elements, e.g. a class which contains attributes and operations, one common approach is to consider the "containment type" T_C for edges in the graph representation. The containment edges are directed edges from the container vertex towards the contained vertices. Edges that are not of type containment, are regarded as "reference type" T_R. Additionally, we have $T_E = T_C \cup T_R$ and $T_C \cap T_R = \varnothing$. Another approach is to consider the metaclasses as the vertex types and the metaassociations as the edge types in the graph representation. The attributes in a graph are derived from the attributes of the metaclasses. Generally, the mapping strategy should be unambiguous and reproducible. The interested readers will find a deeper discussion in [Wenzel, 2010].

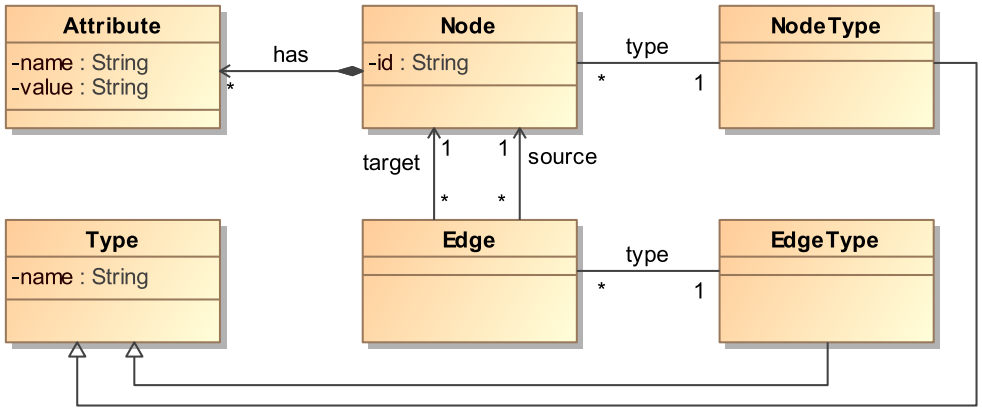


Figure 2.4: Graph representation of models (Adapted from [Wenzel et al., 2007]).

Although, the metamodel describes the relationships between model elements, it does not define how these models should be modified and edited [Kehrer et al., 2013a]. Since the models are represented as graphs, basic graph editing operations can be applied on such a representation. The *basic graph edit operations* are *creating, deleting, moving* and *changing* of elements, attributes and references [Kehrer et al., 2013d]. The computed differences in the state-of-the-art tools, are principally reported in terms of these low-level graph editing operations (see Section 2.4). Although such low-level changes reported on the graph representation of models fully capture the applied changes between two models, they are difficult to comprehend by developers who are used to working on a higher abstraction level when they edit the models through graphical editors. In this regard, such low-level graph edit operations should be semantically lifted to into high-level changes which are easier to understand. To this aim, low-level operations are grouped together in order to form more complex and at the same time more human friendly operations. As an example, a deletion of a class from a package will cause many low-level deletions of other model elements such as attributes and methods within the deleted class, but from a developer's point of view it is just a simple deletion of a class. We investigate this issue more in Section 2.4.

2.3 Model Differencing

As we talked earlier in Section 2.1.6, collaborative software development needs that software artifacts are properly supported by VCS. Since in MDE the main artifacts are models, this imposes that models are properly supported by versioning systems. It means that models should be compared, their conflicts should be derived and resolved and finally they should be merged into a consolidated final model. The process of comparing the models, requires that differences between models are accordingly defined, efficiently computed and properly visualized [Lin et al., 2004]. Model differencing is

not motivated solely by model versioning and model merging. It also plays a crucial role in other activities such as model transformation testing, model comprehension, inconsistency detection and software evolution analysis [Selonen, 2007, Lin et al., 2004, Kolovos et al., 2006].

In model transformation testing, it is desired to test and evaluate a transformation to see if it is correct and if it satisfies its intent and design purpose. In addition to formal proofs and methods, the testing can be accomplished by checking the output of the transformation through computing and analyzing the difference of the output model from the expected ideal model. Model comprehension is usually achieved by transient views on parts of the system which allow investigating different parts of the system from different points of view. Model differencing supports this by showing the changes and differences between such views, highlighting different aspects of the system. Inconsistencies between models or different versions of a model, can be also supported by model differencing in which differences between corresponding model elements can be detected, e.g. abstract class vs concrete class. Evolution of models and tracing of model elements [Wenzel and Kelter, 2008, Wenzel, 2010] through the development life cycle of software systems are other applications of model differencing.

Model differencing is quite mature and there has been much research done on it[6] [Kolovos et al., 2009, Stephan and Cordy, 2013, 2012, University of Siegen, 2014]. The aim of this section is to declare the basic concepts and review the most important aspects of model differencing which we need for better comprehension of the coming sections, without going too deep into the model differencing itself. In Section 2.3.1 we will cover the concepts and definitions in model differencing. Sections 2.3.2 and 2.3.3 review different model differencing approaches

2.3.1 Model Differencing Concepts

Model differencing principally considers two models which should be compared and differences between them should be computed. The models should not necessarily be related to each other, but in practice it is mostly so. It is quite frequent that the models are revisions of each other, i.e. they have a predecessor-successor relationship in a repository or stem form a common ancestor (see Section 2.1.6).

Differences are computed by identifying the common elements between two models. This means that there should be a matching algorithm which establishes a correspondence relationship to the common elements of the two models. An element of the first model M_1 is said to be *corresponding* to an element in the second model M_2, if they represent the same entity in both models [Wenzel, 2010]. Two corresponding elements are not necessarily identical, since they might have been modified. The set of all corresponding elements of two models is referred to as a *matching*.

[6] At the time of writing this dissertation, there is a comprehensive collection of research materials and papers (currently 450 in total) regarding the model differencing and model versioning at [University of Siegen, 2014].

An *asymmetric difference*[7] or simply a *difference* between two models of M_1 and M_2 is the set of changes which are applied to M_1 in order to yield M_2, provided that they are executed without any error [Kelter and Schmidt, 2008, Wenzel, 2010]. The set of changes are expressed in terms of edit operations. *Edit operations* are those operations which are defined or specified to alter models of a specific type. They principally clarify how models of that type should be modified [Pietsch et al., 2012a]. As an example, consider the edit operation of `createState(name)` which is used to create a state with a specified name in statechart diagrams. The edit operations are therefore, type-specific and are not necessarily the same for models of another type. An *edit step* or an *invocation of an edit operation* is an edit operation with concrete arguments. With the above information, an asymmetric difference is a sequence of invocations of edit operations which is also referred to as a *patch*. Therefore, an asymmetric difference, i.e. a patch, can also be *applied* or *executed* on model M_1 in order to get model M_2.

Since we have to first find the matching of two models in order to derive their differences, the elements of two models can be conceptually categorized based on the fact that they are in a matching or not [Kolovos et al., 2006], i.e. if for an element in the first model a corresponding element in the second model exists or not. The element in the first model for which a corresponding element in the second model exists, might conform or might not conform to the second element. We refer to these two types of elements as *corresponding conforming* and *corresponding non-conforming* respectively. Similarly, the elements in the first model for which there is no correspondence in the second model can be divided in two groups: elements that are included in the domain of comparison operations and elements which are excluded from it. We respectively refer to them as *non-corresponding comparison-inclusive* and *non-corresponding comparison-exclusive* elements. Figure 2.5 shows this categorization more clearly. Here conformance is another constraint on correspondences. For instance, if two classes within an already matched packages have the same name but one is abstract and the other is concrete, are corresponding but not conforming. The concept of conformance is quite dependent to the semantics and types of the models.

As [Kolovos et al., 2006] describes, corresponding conforming elements means that the matching algorithm has correctly established a correspondence between models. On the contrary, corresponding non-conforming elements might indicate that the established correspondence is wrong. Non-corresponding comparison-inclusive elements in the first model might indicate that the differencing is incomplete, since they have not been in the calculated matching, either intentionally or unintentionally. They might also indicate that the matching is wrong. Non-corresponding comparison-exclusive elements might indicate that the comparison operation is incomplete or intentionally disregards them.

As discussed earlier, finding the matching between two models is principally a graph

[7] Considering the problem of displaying differences of two models, a common approach is that the differences of two models are created and displayed based on the common parts. In this context, a *symmetric difference* is defined as the matching set and two inserting transformations which allow non-common elements are added to the common parts [Kelter and Schmidt, 2008]. Presentation of differences is not of our interest and is not covered in this dissertation, but the interested readers will find more information in [Kolovos et al., 2009, Stephan and Cordy, 2012, 2013].

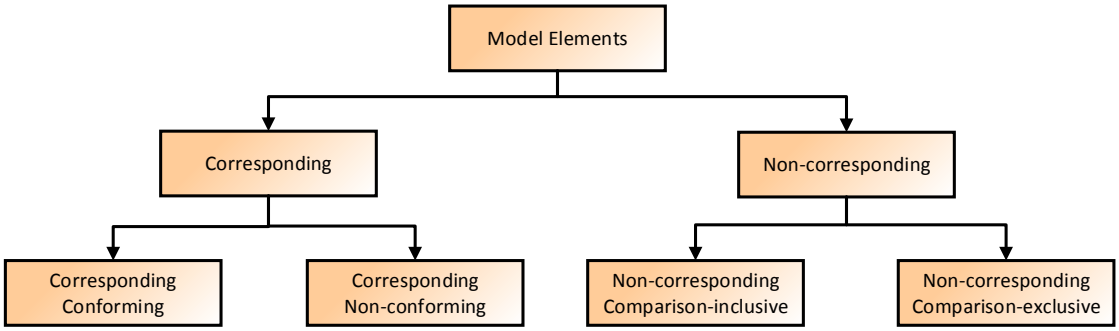


Figure 2.5: Conceptual categorization of elements in two models (Adapted from [Kolovos et al., 2006]).

isomorphism problem which is NP hard and is therefore very computation-expensive [Read and Corneil, 1977]. Once the matching is computed, the elements which do not belong to the matching or which are not identical in both model are basically regarded as bases for deriving the differences.

Although establishing correct correspondences between model plays a crucial role in properly deriving the difference between models, valid differences between models are not unique. For instance, in the case we have two models which are revisions of each other, it can be argued that all elements of the first version are principally deleted from it and then all elements in the second version are added [Wenzel, 2010]. Since it is often the case that such interpretations are not happening in daily practices, we should detect the common parts of two models and try to derive their differences upon it.

The way we find the common parts of models is also dependent on how models are presented. As mentioned in Section 2.1.6, textual presentation of models are not a suitable basis for deriving the changes. The text based differencing algorithms typically use the longest common subsequence (LCS) algorithm [Cormen et al., 2009] in order to compare lines of text files. GNU[8] `diff` and `diff3`, are among the most famous text based comparison programs which work in this way [MacKenzie et al., 2002]. Text based comparison methods typically compare text files line by line and the lines which are not identical are considered to be added or deleted.

Comparing models which are already serialized into text forms, e.g. using XMI standard, by text based algorithms is erroneous [Kelter et al., 2005, Wenzel et al., 2007]. Such approaches do not take the semantics and structures of the models into account, e.g. if an element is stored in a different position in the text representation a difference is reported which is conceptually wrong. Therefore the state-of-the-art model differencing tools use the abstract syntax graphs of the models in order to compare them [Kehrer et al., 2012b]. Principally they consider models as graphs (see Section 2.2), in which nodes represent elements within the model and the edges represent the relationships and references between them [Brosch et al., 2012].

[8] GNU is a Unix-like operating system which just consists of free software.

2.3.2 Model Differencing Approaches

As discussed in Section 2.1.6, model differencing is prerequisite for model versioning. Merging of models requires that first the differences between them are computed. There are two model versioning approaches: state based and operation based merging. Principally, the same also applies when we consider the model differencing stand alone.

In *operation based differencing*[9], the differences of models are computed through the history of applied edit operations on models which are recorded by the editors in which the models are modified [Kehrer et al., 2012b]. This approach is quite fast and accurate since there is no need to compare the models and the relationships between the elements of the original and the resulting models are already known through the applied edit operations [Altmanninger et al., 2009]. It additionally allows more complex operations to be defined and the order of modifications to be retrieved [Brosch et al., 2012]. These advantages come with the big price of tight dependency to the corresponding editor. Models which are developed by other means are not supported.

A *state based differencing* does not require any protocol of the applied changes. It principally compares two states of the models for deriving the differences in which difference computation is done through identifying common parts of the models. State based differencing can be further divided into model-type-specific and generic approaches [Kehrer et al., 2012b].

Model-type-specific differencing approaches are tailored to a specific type of models and cannot handle models of other types. For instance, [Nejati et al., 2007] addresses the problem of matching and merging statecharts and [Xing and Stroulia, 2005] deals with the detection of structural changes in the design models of object-oriented software systems. *Generic differencing* approaches on the other hand, deal with models of different types. Such approaches principally consist of algorithms which are configurable and can be adapted to different model types. Figure 2.6 shows the classification of model differencing approaches.

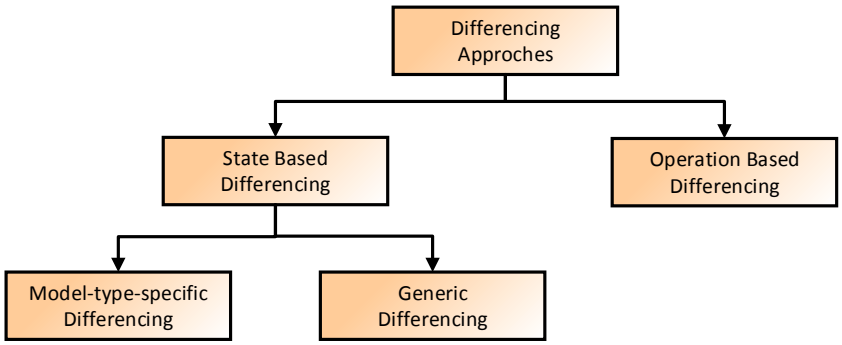


Figure 2.6: Classification of model differencing approaches.

Since operation based and model-type-specific approaches do not coincide with the

[9] Also known as the *logging based differencing* [Kehrer et al., 2012b].

contribution and research goals of this dissertation, they are not explained any further but the interested readers will find more information in [Kolovos et al., 2009, Stephan and Cordy, 2012, 2013, Brosch et al., 2012]. In the next sections we will study generic approaches in more details which are related to this dissertation.

2.3.3 Generic Model Differencing Approaches

In generic model differencing approaches, establishing correspondences and forming the matching is typically controlled by configurations which specify which characteristics of model elements are more relevant for a matching. Obviously the quality of correct matchings has a direct influence on the derived differences. Establishing correct matchings highly depends on the characteristics of the models and their types, and it is quite variant between models [Kehrer et al., 2012b]. Generic approaches try to establish correct matchings by adapting their matching strategies through taking characteristics, types and semantics of models into account.

There are different strategies for matching of elements between two models [Kolovos et al., 2009, Kehrer et al., 2012b]: static ID based matching, signature based matching and similarity based matching.

Static ID Based Matching In this approach, it is assumed that any model element has a unique persistent identifier, e.g. a UUID[10]. The unique IDs are then used to form the matching of two models which is quite fast and straightforward. Moreover, this approach does not need any user configuration [Kolovos et al., 2009]. One drawback of this approach is that the models of interest might not be revisions of each other, i.e. they might be developed independently. The other drawback is that the models might not have persistent identifiers at all, since they are created by a tool which does not support it or by two incompatible tools. Moreover, it is quite debatable that if an element with a persistent ID has changed so considerably, that we can any more regard it the same original element.

Signature Based Matching This approach uses signatures of model elements to establish correspondences and form the matching. A signature of a model element is principally a label which is created by using different properties of the element as well as of the model [Wenzel, 2010]. For instance in class diagrams, a typical simple example of forming a signature for an element is to use its name in conjunction with its parent element, recursively going to the root. But generally if names are used in forming a signature, the renamed elements will not be detected.

Since computing the signatures is deterministic, equal elements will have equal signatures. But in practice, not all of relative properties and characteristics of model elements are taken into account. Therefore, there can be more elements with the same signature [Wenzel, 2010]. Ideally, a good strategy for forming signatures should reduce the possi-

[10] Universally Unique Identifier.

bility of having identical signatures for different model elements. Moreover, it is quite typical to have a hash function to form signatures in practice [Kehrer et al., 2012b].

Similarity Based Matching The similarity based matching approaches consider the similarities of model elements for establishing correspondences. The similarities are computed using a specified function, which principally considers different features and properties of model elements. Defining similarity function and choosing relevant properties are usually heuristics and dependent on the types and the semantics of models. For instance, classes with similar names are more probable to be similar than the classes that are both abstract [Kolovos et al., 2009].

In ID based and signature based matching approaches, the elements are either matched or not matched. However in similarity based matchings, the similarities are first computed and presented as numeric values, typically in the interval of $[0, 1]$. A similarity of zero means that the models have no similarity at all based on the specified properties in the similarity function, while a value of one means that they are identical. One strategy is to match elements with the highest similarities. Another strategy is that if the similarities are greater that a threshold, then the elements are matched. Since not all properties are equally relevant in computing the similarities, the similarity based approaches are additionally provided with configuration files in order to properly adjust the matching step. In such configurations typically one can assign weights to the relevant properties. More weight shows that the respective property is more relevant in computing the similarities.

2.4 Difference Computation In This Dissertation

In Section 2.3, we talked about the general approaches that exist for model differencing. Principally, in the absence of unique IDs in models, static ID based matchings strategies cannot be used. Instead, the correspondences between model elements should be established using similarity based matchings.

In this dissertation we just deal with design models of Java software systems which are obtained by reverse engineering the source code (see Chapter 4). Since these design models lack persistent identifiers, only similarity based matching strategies can be used to form the matchings and to compute the differences. In this regard, we chose the SiDiff Model Differencing Framework [Kelter et al., 2005, Kehrer et al., 2012b] which uses similarity matching strategy for computing the differences. We should mention that other differencing tools are also available [Kolovos et al., 2009, Stephan and Cordy, 2012, 2013]. As long as a differencing tool can compute the changes between design models of software systems, it can be used instead of SiDiff in practice. As we compute the changes with SiDiff, it is enlightening that we take a deeper look at how it computes the differences. In this regard, we investigate SiDiff in detail in Section 2.4.1.

Since the changes reported by SiDiff and other differencing tools are principally low-level changes which are expressed in terms of graph edit operations (see Section 2.2), they are difficult to be processed by humans who are used to think of model modification at

a higher level of abstraction. In this regard, we used the SiLift Semantic Lifting Engine [Kehrer et al., 2011] to elevate the low-level changes into high-level ones. Section 2.4.2 will describe SiLift in more details. The notions presented in this section will be further employed in Chapter 4 where we use SiDiff and SiLift to compute the low-level and high-level changes in design models of Java systems.

2.4.1 SiDiff Differences Computation Engine

SiDiff is a generic differencing framework, initially developed to compare different types of UML diagrams [Kelter et al., 2005]. Later, SiDiff has evolved into a tool that can principally compare any two models whose metamodel is expressed in EMF Ecore [Kehrer et al., 2013b]. In SiDiff, the existence of persistent identifiers for elements of a diagram is not required. SiDiff supports both of the signature and similarity based matching approaches.

The computation of differences of two documents principally starts with loading the documents and transforming them into an internal format [Pietsch, 2009]. In the first implementation, the authors used an internal metamodel and the engine derived differences between instances of that metamodel. Although in the current version SiDiff uses EMF Ecore [Steinberg et al., 2009], it is enlightening to review the older implementation for better comprehension, since the basic concepts are the same.

The metamodel used in [Kelter et al., 2005] is depicted in Figure 2.7. The root element is a Document which can contain many Elements. An Element is typed, it may contain other Elements and may have References to other Elements. Elements have also Attributes. By this approach they could handle other model types encoded in XMI than the UML models, although in the current implementation the relationships between model elements are principally described by EMF Ecore.

Once the models of the two documents are created, SiDiff computes their differences in two steps. In the beginning, SiDiff tries to find the elements in the first model which have a corresponding element in the second one, this step is referred to as the *matching step*. The matching process is highly configurable and is based on the similarities between model elements; neither existence of persistent identifiers for model elements nor uniqueness of their names are necessarily required. In the second step, the differences between the documents is calculated based on the matching which was found earlier. The computed differences can be used for graphically representing the differences or can be used in other tools for further processing, as in Section 2.4.2 in which the changes are semantically lifted to obtain more abstract high-level changes. Now we take a closer look at how the matching is done and what kinds of differences are reported by SiDiff.

The Matching Algorithm Because of the hierarchical structure of the metamodel, the instances have a tree like structure in which nodes may contain other nodes. For instance, packages contain classes, while classes contain attributes and operations. Therefore, the matching process between two elements can be done when their possible subelements are matched. In SiDiff, the matching process is done in two steps of bottom-up

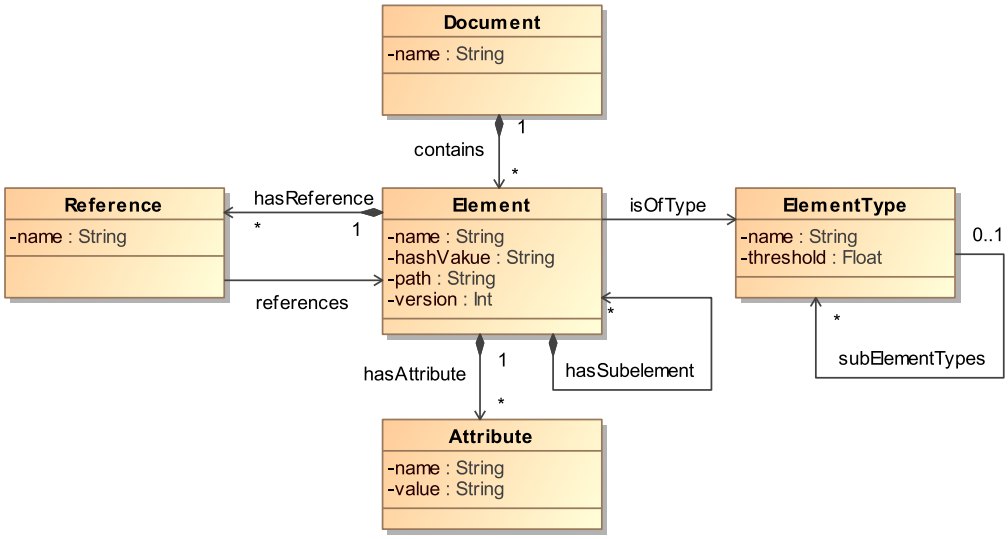


Figure 2.7: The metamodel of the difference algorithm used in SiDiff (Adapted from [Kelter et al., 2005]).

and top-down directions which are alternatively executed till all elements in the documents are processed.

(1) Bottom-Up: In this step, first all elements in the leaves which have similar types are compared. Those elements in the fist document who have just unique correspondences in the second document are matched. If there are elements with more possible candidates, then no match is established since the similarity may change in next iterations when more elements are compared. Between two elements of the same type, the matching is established when the similarity is greater than a specified threshold. After matching the leaves, the process continues to the top of the tree. When two non-leave elements are matched, the algorithm switches to top-down mode in order to propagate this knowledge to match the unresolved children or to update the similarity values of previously matched elements.

(2) Top-Down: In this step, the last match from the bottom-up phase is used to match the children of the matched elements. The fact that parent elements are matched will aid to designate the destiny of unmatched sub-elements or to improve the similarity value of already matched children from the bottom-up phase. The new matching information as well as similarity values propagates further downwards. The algorithm switches back to bottom-up mode when no new match is possible.

The algorithm terminates when there is no more element to compare in the bottom-up phase. The models without an intrinsic tree like structure, may contain cycles due to the

Node Type=Class, **Threshold**=0.5	
Criterion	Weight
Similar value for attribute *name*	0.35
Equal value for attribute *visibility*	0.05
Equal value for attribute *isAbstract*	0.05
Similar set of sub elements of type *attribute*	0.20
Similar set of sub elements of type *operations*	0.20
Similar elements following incoming *generalizations*	0.05
Similar elements following outgoing *generalizations*	0.05
Match parent element	0.05

Table 2.2: Criteria for comparing classes (Adapted from [Treude et al., 2007]).

dependencies between model elements. To handle cycles, the algorithm iterates through the cycle till no new match can be found.

Similarity Computation As mentioned earlier, the model elements with the same type should be compared. The similarity between elements is calculated based on the characteristics and features of the elements. In the case that the computed similarity it is greater than a specified threshold, the elements are matched. This requires that for each model element type, a *similarity function* is defined which considers some criteria on the structure, parts and properties of elements of that type. From the domain expert point of view, these criteria are relevant for considering instances of that type similar.

Formally, let C be the set of criteria for elements of type T and e_i, e_j be two elements of the type T. The *similarity* between the two elements is denoted by $sim\,(e_i, e_j)$ and is defined by:

$$sim\,(e_i, e_j) = \sum_{c \in C} w_c \,.\, cmp_c\,(e_i, e_j)$$

where cmp_c is the compare function for criterion c and w_c is the weight of the function, contributing in the similarity. The similarity function is a weighted arithmetic mean in which the weights are normalized, i.e. $\sum_c w_c = 1$. The higher value for a weight indicates that its corresponding criterion is of more importance when calculating the similarity. The range of the similarity function is the interval of $[0, 1]$ where 0 means no similarity and 1 means the most similar, i.e. identical.

As a simple example of the above explanation, similarity of local attributes like names or similarity of neighboring nodes, e.g. referenced nodes, can be regarded as two criteria for computing the similarity between two classes in UML class diagrams. Table 2.2 shows a sample of the criteria and their assigned weights used in SiDiff to match elements of type Class in UML class diagrams.

Matching Speed-up In the matching step of SiDiff, basically all elements in the two models should be compared. This principally results to an $O(n^2)$ run-time complexity which becomes quite time-expensive when n, the number of model elements in each document[11], increases. To tackle this problem, SiDiff uses a pre-phase of early matching based on the hashes of model elements. It matches those elements which have unique identical hashes [Kelter et al., 2005]. Hashes are calculated based on the name and the path of model elements within the document, their sub-elements and in the case that they refer to other elements, the referenced elements. The creation of hashes are done when parsing the XMI files of two documents for forming the appropriate data structures, i.e. models, in the memory. The complexity of determining the hashes is of $O(n)$ and finding the elements with equal hashes is of $O(n \log_2(n))$ [Treude et al., 2007]. This considerably reduces the run-time of the matching phase.

Additionally, [Treude et al., 2007] have improved the main matching algorithm by proposing the *Similarity Search Sparse Vector Tree*, shortly abbreviated as S^3V tree. The S^3V tree is a high dimensional balanced search tree which speeds up the matching phase. Once the elements of one model are stored in the tree, for a given element in the other model, the most appropriate candidates, i.e. the most similar elements, can be found quite efficiently. The search strategy is to avoid the linear search in all model elements of the same type and to discover the most appropriate candidates which lay within a neighborhood of the given element. The neighborhood of an element is defined based on the Euclidean distance and is principally a ball of radius r, having as center the element.

The basic idea of an S^3V tree is to partition the n-dimensional search space into disjoint cells. It is similar to the LSD (local split decision) tree presented in [Henrich et al., 1989]. A n-dimensional cell, known as an *n-cell*, is principally the sub-space defined by $I = I_1 \times I_2 \times \cdots \times I_n$ where $I_i = [a_i, b_i] \subset \mathbb{R}$, $a_i \leq b_i$; $i = 1, \ldots, n$. A S^3V tree stores the data points which lay in an n-cell, in an associated container called a *bucket*. To form the tree and to present the elements as points in the Euclidean space, each model element is mapped to a numerical vector and is then stored on the tree. A vector consists of the numerical values of the measured characteristics defined for being used in the matching step. For instance in UML class diagrams, the number of attributes and methods of a class can be used in such a vector.

In a S^3V tree, each node is either a directory node or a bucket node. A directory node contains:

- Information about the boundaries of an n-cell, i.e. the intervals of $I_i = [a_i, b_i] \subset \mathbb{R}$; $i = 1, \ldots, n$.

- The dimension d ($1 \leq d \leq n$), where a split in the interval I_d should occur. It divides the current n-cell into two sub-spaces at the dth dimension. The current cell is therefore will be divided into two disjoint sub-cells at the next level in the tree.

[11] For simplicity, we assumed each model have n elements.

- The position p_d ($a_d \leq p_d \leq b_d$), where the interval I_d is split into two sub-intervals of $[a_d, p_d]$ and $[p_d, b_d]$ [12].

In the beginning, the elements of one model, say the second model, are stored in trees. For each model element type there will be one tree which stores the elements of that type. For searching the neighboring elements of a given element in the first model, a *range query* is started. First, based on the element type of the given model, the appropriate tree is selected. The search starts from the root of the tree and according to the splitting dimensions and splitting positions, it is then followed downward the tree, provided that there is an intersection between the range query and the intervals defined by sub-trees. Once the bucket nodes are reached the elements in the bucket which lay in the neighborhood of the given element are returned as the most similar elements in the second model. It should be mentioned that the construction of the tree is done systematically and in a way that the tree is balanced.

Reported Differences As mentioned before, after establishing the correspondence, the elements which are not contained in the set of matching or are not identical within the matching, are basically the elements which have to be considered in differences. SiDiff reports the following differences when comparing two models [Kelter et al., 2005, Treude et al., 2007]:

- **Structural Changes:** The elements which are not reported in the matching, are considered to be structural differences. This consists of *additions* and *deletions* of model elements.

- **Attribute Change:** For two corresponding elements, an *attribute change* is reported when there is a change in the value of their attributes. For instance, a class is renamed or its visibility is changed.

- **Reference Change:** When two corresponding elements have two different references pointing to two different targets, a *reference change* is reported. For example, a method has a new return type.

- **Moves:** If two corresponding elements have two different parents, a *move* is reported. For example a class is moved to a new package in UML class diagrams.

In Chapter 4, differences between design models of our sample Java software systems are computed in terms of the above low-level changes. The aforementioned reported changes by SiDiff are principally the basic graph edit operation as we described in Section 2.2.

[12] For simplicity, the sub-intervals are shown by closed ends. But since the cells should be disjoint at p_d, in practice one end is considered to be open.

2.4.2 SiLift Semantic Difference Lifting Engine

As we saw earlier in Sections 2.2 and 2.4.1, the computed difference between models are expressed in terms of basic graph edit operations. Although these low-level operations are conceptually correct and can completely capture the differences between models, they are difficult to comprehend by human developers who are used to working on higher abstraction levels. In existing tools and editors[13], models are graphically represented and the developers are used to working and modifying the models from this perspective. Such approaches keep internal representations of models out of the sight. This not only provides better comprehension and flexibility, but also hides the complexity of internal presentations and lets the developers work on a conceptual higher abstraction level.

Although the state-of-the-art tools let the users work at a higher abstraction level, model modifications are internally represented in terms of low-level changes. Additionally, the diversity of tools and model types results in various internal representations which make the situation even more complex, since the reported low-level changes are then, not only dependent on model types and internal representations but also on the supported edit operations. Therefore, it is required that meaningful changes are specified in a higher abstraction level. Such conceptual high-level operations must correctly reflect the way that models are edited by users and, model editing and versioning tools must support these kind of high-level edit operations [Kehrer et al., 2012c].

The above discussion leads to the questions whether it is possible to properly categorize and group the low-level operations into more conceptual high-level ones. This has led to research attempts of semantically lifting the low-level operations into more abstract high-level operations [Kehrer et al., 2011, Langer et al., 2013]. As a simple example of such a high-level operation, we consider the deletion of a package in UML class diagrams. Once the package is deleted, all of its contents are simultaneously deleted. This results in many deletions of classes, interfaces, relationships, etc. that are reported in terms of many low-level graph edit operations defined on the abstract syntax graph representation of the model.

In the rest of this section, we first review how models are typically edited and what is a high-level change. In this regard, we provide an example. Later, we discuss how SiLift copes with the problem of semantically lifting low-level changes into high-level ones.

Editing of Models As we described earlier, even simple conceptual high-level editing operations provided by model editors and related tools, typically consists of many low-level tool-dependent operations. Such low-level operations are dependent on the internal representation of models as well as technologies used. In state-of-the-art tools, the internal representation is typically an implementation of the abstract syntax graph of the model (see Section 2.2). Therefore, the basic graph edit operations are supported as low-level operations in such tools and the conceptual changes are expressed in terms of these low-level edit operations.

[13] To name a few: IBM Rational Rose, ArgoUML, Papyrus, Sparx Enterprise Architect and MagicDraw.

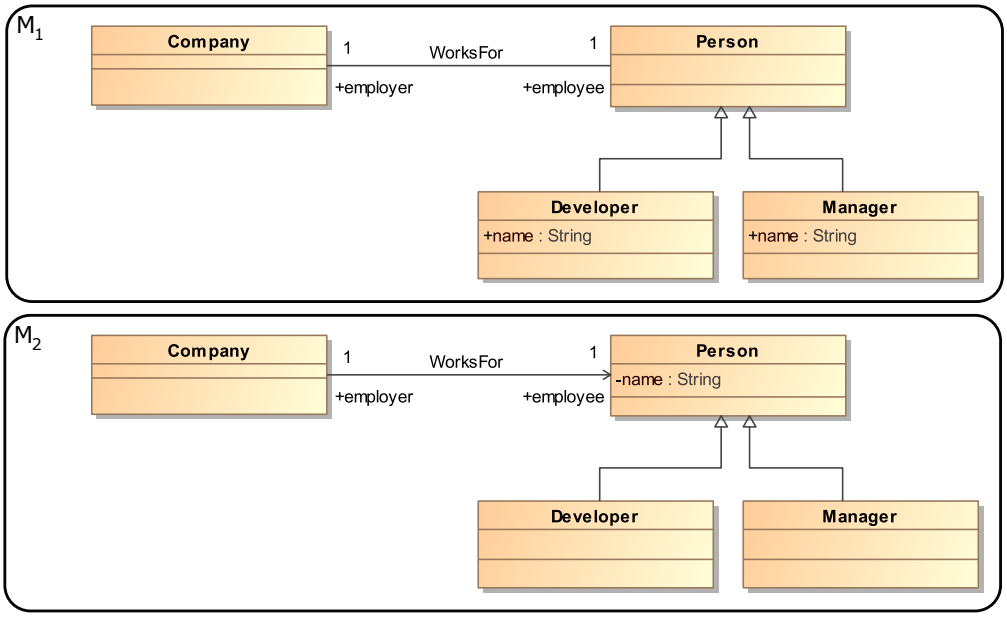


Figure 2.8: Editing of model M_1 into M_2 (Adapted from [Kehrer et al., 2011]).

In order to better comprehend what does a high-level edit operation consists of, we explore an example provided in [Kehrer et al., 2011]. In this example, it is assumed that the internal representation is based on the EMF Ecore. Figure 2.8 shows a UML2 model of M_1 which has been modified into model M_2. The following changes have occurred:

1. The association end of `WorksFor` is restricted[14] towards class `Person`.

2. A refactoring of *pullUpAttribute* has been applied. Therefore, the common attribute, i.e. `name`, of classes `Developer` and `Manager` has been moved to their parent class of `Person`.

In order to better comprehend the low-level changes caused by the previous conceptual changes, we need to know how the model is represented internally. Figure 2.9 shows an excerpt of the EMF Ecore implementation of classes and associations in UML2. Here, classes and associations are expressed via `Class` and `Association` respectively. Association ends are expressed via `Property`. In this excerpt, association ends can belong to classes via `ownedAttribute` or to associations via `ownedEnd`. In our scenario, it is assumed that navigable ends are owned by classes via `ownedAttribute` and non-navigable ends are owned by an associations via `ownedEnd`.

[14] For model M_1 of this example, it is assumed that in the absence of arrow heads for the association `worksFor`, the association is directed in both directions from `Company` toward `Person` and vice versa. Here restricting the navigability means that, the association is modified to be just one way from `Company` to `Person` in M_2.

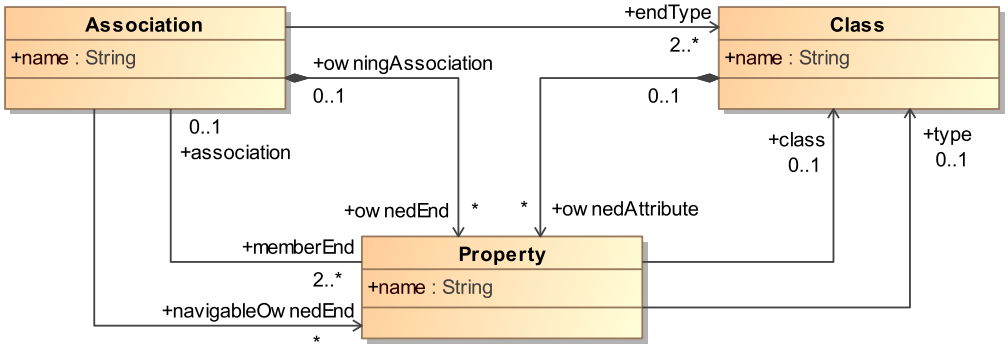


Figure 2.9: Excerpt of UML2 metamodel implemented in EMF (Adapted from [Kehrer et al., 2011]).

Using the previous metamodel, the high-level edit operation of restricting the association end of `WorksFor` toward the class `Person` is depicted by Figure 2.10[15]. This conceptual operation is internally expressed via the following low-level changes:

- The reference of `ownedAttribute` from the class `Person` to the property `employer` has been removed.
- The reference of `class` from the property `employer` to the class `Person` has been removed.
- The reference of `ownedEnd` from the association `worksFor` to the property `employer` has been added.
- The reference of `owningAssociation` from the property `employer` to the association `worksFor` has been added.

Similarly, the refactoring of "pullUpAttribute" causes even more low-level changes. Moreover, in this refactoring the number of child classes that have the common property, is not fixed and might change from one application to another.

It should be mentioned that using the low-level graph edit operations, in addition of being hard to comprehend, brings the inconsistency problem into the scene [Kehrer et al., 2013a,b]. It is then quite possible to have a bunch of low-level operations that violate the consistency of a model. For instance, in our previous example of Figure 2.10, it is possible to create a property without creation of the necessary references. Such inconsistent models cannot be processed neither by tool editors nor by model transformation engines and code generators. Therefore, it is necessary that such low-level operations are correctly identified and properly grouped together to form the high-level edit operations. Such operations are conceptually applied from the users' perspective and are consistency-preserving. In the next section, we describe the SiLift approach for conceptually lifting the low-level changes to high-level ones.

[15] In the picture, the green color shows addition and the red color shows the deletion.

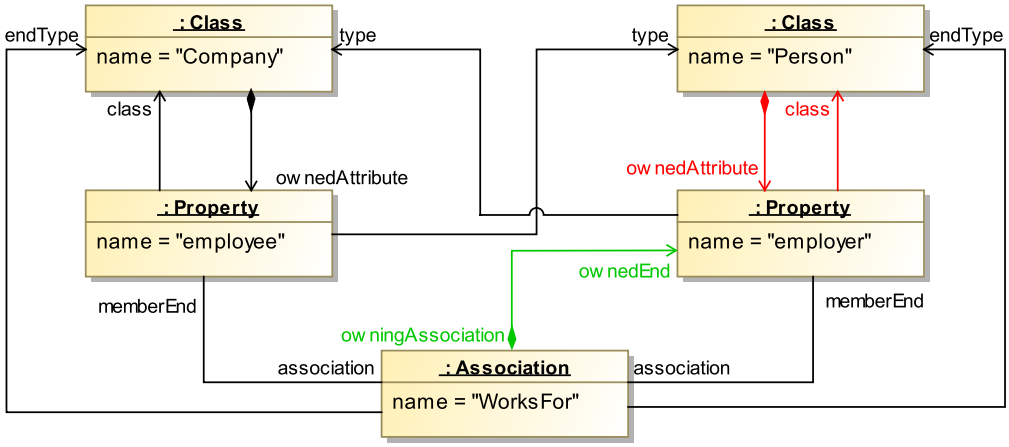


Figure 2.10: Internal representation of restricting the association end `WorksFor` (Adapted from [Kehrer et al., 2011]).

SiLift As we described earlier, conceptual high-level changes are more descriptive than low-level edit operations. Moreover, they are also consistency-preserving. When we compare two models, the changes can be expressed either via low-level graph edit operations or high-level conceptual operations. The detection of low-level changes can be done using generic model differencing engines as described in Section 2.3.3. Once the low-level changes are correctly detected, they should be investigated and appropriately grouped in a way that high-level changes can be recognized. In other terms, low-level changes should be semantically lifted to identify high-level changes.

For detection of low-level changes in this dissertation we used the SiDiff model differencing framework which was described in detail in Section 2.4.1. For post processing the low-level changes and detecting high-level changes we used the SiLift semantic lifting engine [Kehrer et al., 2011]. To detect patterns in low-level changes, SiLift internally uses Henshin. Henshin is an in-place[16] model transformation engine [Arendt et al., 2010]. The Henshin transformation language uses pattern-based rules on low-level changes. A *rule* consists of a left hand side graph (LHSG) and a right hand side graph (RHSG). These graphs are model patterns expressed in terms of underlying internal structure. Furthermore, a rule consists of attributes, parameters and application conditions. *Application conditions* are logical conditions which should be met in order to apply a rule. They control the existence (*positive application conditions*) or absence (*negative application conditions*) of other specific patterns in a model in addition to the original patterns of interest, e.g. if a node additionally has an outgoing or an incoming edge. Once an LHSG is found and the application conditions are met, the LHSG is transformed to the RHSG based on the transformation defined in the rule.

Since the recognition of high-level changes basically is a pattern matching in low-

[16] In-place model transformation engines work on the given model directly without creating any copy of it.

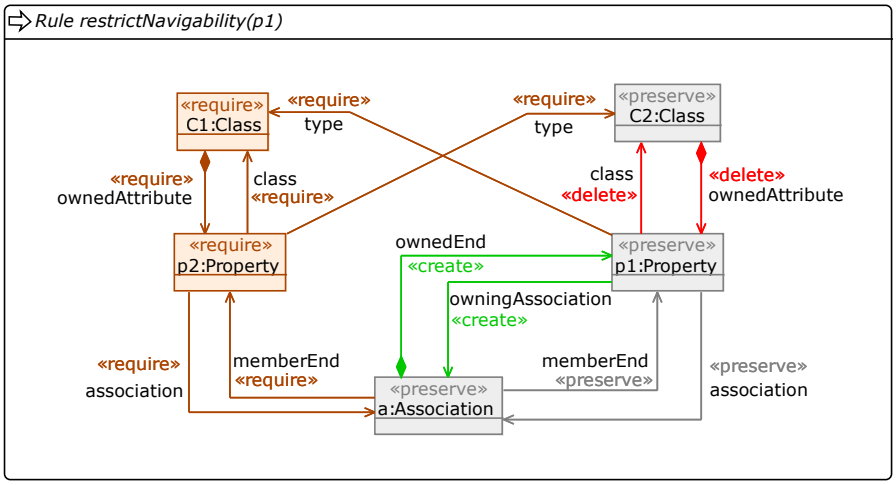


Figure 2.11: Recognition rule of restricting the association end (Adapted from [Kehrer et al., 2011]).

level changes, in order to detect high-level changes, SiLift has transformed edit rules of Henshin into recognition rules. In a *recognition rule*, the appropriate pattern of a high-level change is defined in terms of low-level changes [Kehrer et al., 2013a]. Henshin finds the defined patterns of low-level changes in the computed differences of two models in order to detect the corresponding high-level changes. For better comprehension of such rules, we provide the recognition rule of our previous example [Kehrer et al., 2011]. Figure 2.11 demonstrates the recognition rule of restricting the association end in our previous example. The "stereotype" of «`preserve`» represents the model elements which are present in both LHSG and RHSG. Such elements form the core of the pattern which should be found. The stereotype of «`require`» has a similar role to «`preserve`» in the sense that the elements should be present in both LHSG and RHSG but they are not considered as the core of the pattern (positive application condition). The stereotype of «`delete`» represents the elements which are deleted from the LHSG. Similarly, the stereotype of «`create`» represents the elements which should have been created in RHSG. In the core pattern, the values of `c2`, `a` and `p1` are the parameters of the rule which are bounded to the appropriate class, association and property respectively. Once the above pattern is found in the low-level changes, SiLift reports the high-level change of restricting the association end from model M_1 to M_2.

2.5 Summary

In this chapter we reviewed the basic or the most important concepts in the field of MDE. In this regard, we learned that MDE increases productivity by elevating the level of abstraction and separation of business logic development from the underlying technological requirements for its implementation.

In MDE, models are primary artifacts. The semantics and syntax of a model is defined in the (immediate) upper level in the abstraction hierarchy. Such a model in the upper level which defines the relationships in models of the lower level is referred to as a metamodel. The model to metamodel relationship is a relative relationship, i.e. a model conforming to a metamodel can be itself be a metamodel for models in its (immediate) lower level. Such hierarchical relationships allow a system of interest be handled from its abstraction at the top, to its implementation at the bottom. In OMG standards, the level of abstraction is limited to four levels where the topmost level is self descriptive.

Model transformations let the models to be transformed to other models or to code for execution. In this regard, a PIM (platform-independent model) can be transformed to another PIM or to a PSM (platform-specific model), defined on the platform of interest for implementation.

Since models are the main artifacts, the collaborative development of models requires that they are properly handled by model versioning systems. A model versioning system allows a consolidated model be created from different versions in the collaborative development environment. In this regard, it is essential that differences between versions of a model are properly detected, resolved and merged.

Model differencing approaches address the previous issue by comparing models. First, corresponding elements between two models are detected and based on them differences are computed. The typical approach in model versioning and models differencing is to regard models as typed graphs. This way a proper difference between two model can be computed. The differences between models are basically expressed in terms of basic graph edit operations. Although such operations are correct, they are too detailed to be comprehended by developers who use to working on a higher abstraction level. Therefore, such low-level graph edit operations are semantically lifted to more abstract edit operations which are referred to as high-level operations.

Since in this dissertation we are dealing with design models of Java software systems, we reviewed how low-level and high-level changes between such design models are computed. In this regard, we reviewed the SiDiff model differencing and the SiLift semantic lifting engines in more details by providing a detailed example.

Part II

Generating Test Models

CHAPTER 3

Controlled Generation of Models with Defined Properties

In this chapter, we focus on how we can generate models which have more realistic properties and how one can finely control the generation process of such models. The controlling mechanisms offered in this chapter allow models with various specified properties to be generated.

In this regard, we first review the related works with particular attention to the approaches and algorithms proposed to generate models. We present a set of criteria which we have collected from different research works and we investigate whether such criteria are met in the existing approaches. Most of these criteria are of great importance for generating test models for model differencing, model merging tools etc. We found out that almost all of the approaches and algorithms in the related works lack three important criteria of our list.

More specifically, the state-of-the-art model generation algorithms and methods, have very limited controlling mechanism on the generation process of models, resulting the generated models be of limited use. It is also not easy to define and create arbitrary complex structures of interest in the generated models. Moreover, they have not addressed how stochastic properties of the generated models should be handled or recreated. Such criteria are of great importance when generating test models for model differencing, model versioning and model analyzing tools.

The identified deficiencies are then addressed in this chapter, by proposing a tool which offers fine tuning mechanisms to control the generation process of models. The proposed approach additionally addresses how test models with specific complex structures and stochastic properties can be generated.

The rest of this chapter is organized as follows. Section 3.1 provides an introduction to the topic and presents the criteria which should be met by artificial model generators. Section 3.2 will review the related works in detail. We will investigate which of those criteria is met by any of the current state-of-the-art approaches in the field of model

generation. The shortcomings in the existing approaches are identified, and then are addressed in Section 3.3. In this regard, Section 3.3 is completely devoted to our approach for generating artificial models. There, we will introduce The SiDiff Model Generator tool. We will review the general features of the tool and where it can be used. We will also describe how models are modified by the tool and how new models are generated. The identified deficiencies in the existing approaches, i.e. controlling of the generation process and statistical properties of the generated models are addressed in detail in Section 3.4. Performance evaluation of the model generator is provided in Section 3.5. The chapter ends with a summary of the presented materials in Section 3.6.

3.1 Introduction and Background

As discussed in Chapter 1, MDE approaches, algorithms and tools have to be validated, checked and evaluated from different aspects such as quality, efficiency and scalability. Model transformation, model differencing, model versioning and merging tools are just few of many where test models are of much need.

Unfortunately appropriate test models are not readily available since they are quite scarce and more importantly they often lack the desired properties and features. Manually generating many test models is very tedious and even sometimes impossible. Therefore, some test model generators have been proposed in recent years, each offering a variety of features. It is well established that generating of models is a challenging task and the existing algorithms and methods have tackled different aspects of this.

In this section we first consider the most fundamental criteria which should be met by a typical model generator. The list of criteria has been formed by the studying the state-of-the-art literature in the field of model generation (see Section 3.2). The list is relatively comprehensive, i.e. it can be applied to various model generators, although possibly with just few exceptions. Such a list lets us study different aspects of the existing model generators and assists us to highlight possible deficiencies in the field. We first present the list and then we discuss what each criterion is intended for. Typically a model generator should address the following criteria:

C1 The artificially generated models must be valid and consistent with respect to their corresponding metamodel.

C2 The generated models should fulfill the requirements which are additionally expressed in terms of OCL constraints. For instance uniqueness of names or bounds on numerical values are two frequent constraints of these types.

C3 There should be a capability for creating valid complex structure within models. Such complex structure have to be formed with respect to the metamodel. They are usually formed by possibly combining model elements or by modifying model elements through permissible edit operations defined for modifying them. As a simple example, in the UML class diagrams one can consider the creation of an association which is composed of a node, two association ends and some edges.

C4 The generation of the models should be under control, i.e. the generator should provide some mechanisms in order to let the user control the generation process.

C5 In order to obtain realistic models, a model generator should have the capability of generating models with desirable statistical properties. For instance the frequency or distribution of different model elements and their internal relationship should be under control and should typically resemble real properties one observes in true software systems.

C6 Considering a set of time-variant related models, i.e. model histories, one should also be able to control the amount as well as the characteristics of changes between subsequent models.

C7 To be useful in the domain of model differencing and model versioning etc., a model generator should be able to generate differences between models (see Section 2.3.1).

Before moving to the next section, we discuss why such constraints are important. The first criterion is principally the most essential one in our list. If generated models are not conforming to their metamodel, they are not useful at all, since they are not valid constructs according to the metamodel, violating the most fundamental assumption in MDE (see Section 2.1).

Although OCL is not the only option to express other constraints of interests on the generated models, it is widely used and many MDE tools and approaches employ it (see Section 2.1.4). Therefore, supporting OCL constraints is of particular attention in the field, although very limited subsets of it have been covered in all of the existing approaches (see Section 3.2).

Creating complex structures of interest is also quite important, specially in the field of model transformation. Such complex structures are formed with different intentions in mind. For instance, in model transformation testing, the complex structures are formed based on the aspects of the transformation which should be tested [Fleurey et al., 2004, Baudry et al., 2006, Brottier et al., 2006]. However, in model differencing and merging the editing steps, i.e. the differences, which results in simple or complex structures are more useful. This way, the changes between models can be derived and different versions of models can be merged into a unified version [Brosch et al., 2012, Kolovos et al., 2009]. Thus, our criterion of **C7** is quite essential in model differencing and versioning tools. Moreover, the generation of models should be under control. This helps different properties and characteristics of interest to be created on demand [Pietsch et al., 2012a, Baudry et al., 2006].

In many of the testing scenarios specially in model versioning, differencing and merging tools, it is quite essential that the generated models represent the required stochastic properties [Shariat Yazdi et al., 2014b, 2013]. This is also true when we consider model histories where each model is derived from its predecessor by some modification. Such changes are not independent of each other and the artificially generated model histories should additionally reflect the real properties of changes one observes in true software models [Shariat Yazdi et al., 2014a].

In Section 3.2, we will see that the state-of-the-art works do not meet some of the above criteria. Specially, **C7** is not met by any of the existing approaches. Moreover, the proper controlling mechanisms (**C4**) for generating models, stochastic properties of the generated models (**C5**) and model histories (**C6**) are not addressed by them. These three criteria are principally the points that this dissertation is going to contribute.

3.2 Existing Approaches for Generating Models

In this section, we try to address the existing related works in the field of generating test models for MDE tools. To the best of our knowledge, there is just one survey which to some extend has a similar intention [Wu et al., 2012]. In this survey, the authors were interested to see where the generated models are used, which criteria are used to select model instances and which tools are there to produce models.

Using relevant keywords, the authors used a systematic way to find papers. This means that some of the reported papers in that survey are not completely in the direction of our research interests. Rather, we are more interested to see how models are generated and which algorithms and methods are proposed in this regard. More importantly, we will investigate the state-of-the-art works in order to see if the presented approaches and algorithms fulfill the criteria of **C1** to **C7** mentioned earlier in Section 3.1. Since **C7**, i.e. support for the generation of differences, is not met by any of the existing approaches, we will not report it in the coming sections when we review each of the related works, item by item. The interested reader will get other useful information by studying the work presented in [Wu et al., 2012].

By studying the related literature, we can categorize the proposed approaches and algorithms into two main categories. The first category consists of approaches and algorithms which directly try to generate models. Such approaches can be further divided into approaches which use non-formal methods and the approaches which use formal methods. The second category for generating models consists of the approaches that indirectly generate models. Figure 3.1 shows this categorization more clearly.

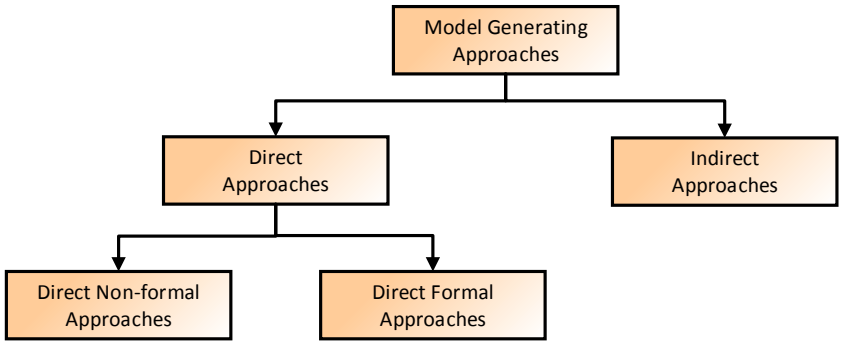


Figure 3.1: Categorization of model generating approaches.

We present the related literature based on this categorization. To this aim, Section 3.2.1 reviews the approaches that directly generate test models using non-formal methods, while Section 3.2.2 presents the approaches which use formal methods. Section 3.2.3 is devoted to the approaches which indirectly generate models. In each section, the related literature are presented in the chronological order. Reviewing of related works ends with a summary of our findings in Section 3.2.4, highlighting the deficiencies in the existing approaches which is addressed by this dissertation.

The reader who is just interested in the results and summary of our literature review, might directly refer to Section 3.2.4 and skip detailed discussions in Sections 3.2.1 to 3.2.3.

3.2.1 Direct Non-formal Approaches

In direct non-formal approaches, models are generated using algorithms and methods which are not rooted in formal foundations such as graph grammars [Ehrig et al., 2009]. In these approaches, the generation of models are mostly motivated by the application domain. For instance approaches of [Fleurey et al., 2004, Brottier et al., 2006] are motivated from the field of model transformation testing while [Pietsch et al., 2012a] is motivated from model differencing and model versioning. In this section we chronologically review related works and we investigate which of our criteria is met by each approach.

Approach of [Fleurey et al., 2004] In this paper, the authors generally address the problem of testing in MDE. They also coarsely cover the testing of model transformation and point to the test models generation problem from this perspective. In this work, no practical implementation is provided and the authors just highlight some general aspects when considering testing of transformations and test models generation. The concepts presented in this work are used in other works to generate models, e.g. [Brottier et al., 2006, Wang et al., 2008].

In this work, the authors suggest to specify parts of the metamodel which is more important from the testing point of view. Such subset of the original metamodel is referred to as the "effective metamodel". They suggest to use partitioning of the metamodel based on the interesting aspects in the transformation program which should be covered and tested.[1] For the specified partitioning, appropriate values are assigned. The values are chosen in a way that help with testing purposes, for example values of type String are typically chosen to be an empty and a non-empty arbitrary strings. Combinations of some of those partitioning with their associated values, are referred to as "model fragments". The model fragments which are essential and should be be covered for testing, are referred to as "covering items".

[1] To understand what is a partitioning in practice, please refer to the description of the approach of [Brottier et al., 2006], in which the authors partially implement the concepts presented in the work of [Fleurey et al., 2004]. Moreover, a detailed and step by step implementation of ideas presented here is discussed in [Wang et al., 2008].

The suggested strategy is to use covering items as well as other model fragments to instantiate the test models. To this aim, covering items should be instantiated first and then this partially generated model is completed in order to make it conforming to the metamodel. Not all of covering items are expected to be in one generated model, so they can be covered in a new model or can be added to the same model. This has effects on the properties of the generated models and can be controlled by user through specifying size, minimum and maximum number of model elements or covering items. Another issue affecting the property of the generated models is the order in which covering items and model elements are instantiated. To this aim, they suggest to use the Bacteriologic approach[2] to generate various test models.

Although the approach presented by [Fleurey et al., 2004] has not been implemented, it can be judged if it covers our criteria. By this approach, the created models will most likely be valid according to the metamodel, so **C1** is covered. The OCL constraints, i.e. **C2**, are not addressed and therefore we assume that they are not covered. Creating complex structures can be supported through covering items and how they are formed and linked. Due to this, not every arbitrary detailed complex structures can be created and **C3** is partially supported. The generation process can be just controlled via defining how the covering items are present in the generated test models, so **C4** is just quite limitedly supported. Controlling statistical properties and generating model histories are not covered, so **C5** and **C6** are disregarded.

Approach of [Baudry et al., 2006] The authors discuss some challenges with respect to testing of model transformations and highlight different issues for developing techniques of such testings. For testing a transformation, they believe that each class of the input metamodel should be instantiated at least once in a model of the generated test sets. Regarding attributes in the generated models, they think that some representative values should be employed in the generated models. This line of thought is actually similar to the partitioning approach introduced in [Fleurey et al., 2004]. More complex structures and properties are obtained from combining these simpler partitioning strategies and the concept of model fragments.

The authors reported to have developed a tool that automatically generates test models. The generated test models were expected to satisfy some criteria which they defined for testing of their transformation. The authors did not mention the criteria explicitly but they mentioned that those criteria are black-box criteria which should cover some structures of interest in the source metamodel of a transformation (see Section 2.1.5). In order to cover those criteria, they reported that their generation algorithm has the following variation points: (*a*) it is possible to choose values of properties form a particular range, (*b*) it is possible to choose whether the instantiated model fragments go to one test model or to a newly generated test model, and (*c*) it is possible to decide how to build new model elements in order to complete the partially generated model conforming to the metamodel.

[2] Bacteriologic approach is a kind of genetic algorithms [Michalewicz, 1996, Sivanandam and Deepa, 2008].

In this work, the generated test models are conforming to the metamodel, thus **C1** is satisfied. The authors refer to OCL constraints from transformation specification point of view. They are not considered in the generated test models, so **C2** is not addressed. Generation of complex structures is similar to the partitioning approach, so **C3** is partially fulfilled. The generation of test models is controlled by the previously mentioned variation points, so **C4** is partially satisfied. Stochastic properties of the generated test models and model histories are not addressed, thus **C5** and **C6** are not satisfied.

Approach of [Brottier et al., 2006] The focus of the work is about generating test models for evaluation of model transformations. This work can be regarded as a restricted practical implementation suggested by [Fleurey et al., 2004].

In the beginning, the effective metamodel has to be specified. The effective metamodel is a subset of the given metamodel which is considered to be more relevant for the testing scenario. In the next step, the effective metamodel is partitioned into disjoint subsets provided that the union of the subsets is the effective metamodel. Such partitioning is interesting from the testing point of view and the instances of them should be generated in the test models [Fleurey et al., 2004]. Model fragments are formed by combining some of the partitions, they show possible structures whose instances are required to be present in the generated models. The algorithm then tries to generate instances of model fragments and completes them in order to build valid instances of the given metamodel.

In their approach, they partitioned the elements in the effective metamodel whose types were primitive, e.g. attributes with the String or Integer type. As an example, strings were typically partitioned to an empty and a non-empty random strings[3]. Later, the model fragments were formed by combining such partitions, their instances were created and finally the created instances were completed to be conforming to the metamodel. The generated test models are strongly dependent on the partitioning strategy as well as the way model fragments are formed and completed.

To control the generation process, they defined some variation points in the algorithm. The maximum and minimum number of model fragments to be used and also the strategies of using the existing fragments or creating new ones, are two examples of such variation points in their approach.

In their work criterion **C1** is covered, i.e. models are correct based on their meta model, but **C2** is not met and the generated models do not satisfy OCL constraints. Regarding the complex data structures, the user has the possibility to define model partitioning and model fragments in a way that some complex structures are created, although with limitations. Therefore, criterion **C3** is not fully met by their model fragments approach and not all complex edit operations are achievable there. Controlling the generation process is done via variation points defined in the algorithm, so **C4** is

[3] Although the union of such partitioning, does not produce all possible strings, from the practical point of view, the model fragments which can be formed by them are sufficient for testing purposes [Fleurey et al., 2004].

supported. Neither statistical properties (**C5**) nor model histories (**C6**) are covered by that approach.

Approach of [Lamari, 2007] This work is a conceptual discussion about generating test models for model transformation tools. In this regard, the author discusses about verification of model transformations. In such verifications, the aim is to verify if a transformation satisfies its specified requirements.

The author suggests black-box testing based on the partitioning method. The partitioning method discussed in this work is principally the method discussed in [Fleurey et al., 2004]. Since no issue regarding the implementation is addressed in this work, it is not possible to verify our criteria of **C1**-**C6** directly. Therefore, we can assume that the same argument we had for [Fleurey et al., 2004], holds here as well.

Approach of [Wang et al., 2008] This work can be regarded as a practical implementation of the partitioning approach introduced by [Fleurey et al., 2004] for testing model transformations. In this work the authors describe their implementation in detail.

As we explained earlier, [Brottier et al., 2006] had already implemented the partitioning approach of [Fleurey et al., 2004]. The novelty of this work in comparison to the implementation done by [Brottier et al., 2006] is that here the authors obtain the effective metamodel automatically from the transformation rules. In addition, they consider the model transformation when deriving the covering items. More precisely, they automatically derive the appropriate values for partitions from the "Tefkat" transformation code. Tefkat is a declarative transformation language. The values are derived by analyzing the transformation rules, e.g. when there is an assignment of a constant value or a comparison involving constants, such constants can be used as candidate values for testing. Such values are combined with the values which are derived automatically from the effective metamodels. This set of values is then used to form the covering items which are finally covered in the generated test models.

The generated test models are conforming to the input metamodel of the transformation, so **C1** is supported. OCL constraints are not discussed, so **C2** is disregarded. Complex structures in the generated test models are created via the covering items, so due to our previous discussions **C3** is partially supported. The algorithm does not have much control on the generation of the test models, except through the way that covering items are instantiated and delivered in the test models. Therefore, **C4** is just partially supported. No stochastic properties of the generated items and no model histories are considered, so **C5** and **C6** are not fulfilled.

Approach of [Mougenot et al., 2009] This approach mainly concerns about uniformly generation of huge test models based on a given metamodel. To uniformly generate test models, they employed the Boltzmann method. The Boltzmann method is reported to be able to uniformly create tree structures with a linear run-time complexity. Here, the uniformity means that for a tree of given size (size is the number of nodes

in the tree), the method is capable of uniformly generating all tree structures of that size, i.e. each of those trees will be generated with the same probability.

To generate the models, the metamodel has to be transformed into a tree structure. The rules of the tree grammar for the Boltzmann method consist of two binary operator of union and product, and one unary operator for creating sequences of elements with arbitrary size. A union generates either of two elements while the product produces both of them. Moreover, in addition to an empty element type, two more element types are considered. The first group is elements with "leafs" type and the second one is "nodes with arbitrary types". Using the previously mentioned rules and types, binary as well as general trees can be generated by the Boltzmann method. The generator rules are parametrized. The parameters of a rule are computed by solving their corresponding system of polynomial equations which delivers the probability of applying each generator rule. For each of the rules, the probability of applying that rule is calculated in a way that trees are uniformly generated.

The process of transforming a metamodel into a tree is not complete, i.e. not every model element can be transformed into a tree, therefore they just consider the core of the metamodel for transformation. The reason for this is that metamodels are more expressive than trees. The transformation of a metamodel is done in three steps. Containment relationships were transformed first, followed by inheritance relationships in the second step. Therefore, other types of relationships are not considered. The last step considered the cardinalities. The required attribute values such are names, visibilities etc. were randomly generated.

In this approach there can be the cases that some elements in the metamodel are not transformed to the tree representation and therefore cannot be instantiated. By this approach **C1**, i.e. correct instances of the metamodel is met. OCL constraints is not considered, so **C2** is disregarded. In this approach, no complex structures are supported and **C3** also is not met. Regarding controlling the generation of test models, since the aim of this approach is uniformly creating models, there is no control on the fine-grained generation process and therefore **C4** is not regarded. From the statistical properties, just uniform generation of all trees with the same size is supported, thus **C5** is not satisfied and there is no control on the generation of specific elements of interest and their frequencies. Additionally model histories are not supported by this approach (**C6**).

Approach of [Pietsch et al., 2011, 2012a] These works are the contribution of the author of this dissertation for generation of more realistic test models and address the deficiencies in the field. They are also the basis for the materials presented in this chapter. The details of the approach is presented in Sections 3.3 to 3.5. But in order to compare all approaches, we review them briefly here.

We were interested to generate test models for MDE tools and editors, specially in the framework of model differencing and model versioning. The work is principally based on the theoretical background presented earlier in Sections 2.2 to 2.4. The basic idea is that the models have abstract syntax graph (ASG) representations. Such representations are extensively employed by many tools (see Section 2.4). In ASG representation, models

are considered as attributed typed graphs. ASG representation principally imposes the legitimate operations that can be used to alter models. As stated earlier, the basic graph edit operations are *creating, deleting, moving* and *changing* of elements, attributes and references.

As discussed in Section 2.4, these low-level graph edit operations can be combined together in order to form more complex high-level edit operations. The low-level and high-level edit operations are then used in order to modify a base model in order to derive new models. The base model can be an empty or a non-empty model. In each modification step, the consistency of the model to the metamodel as well as the specified OCL constraints is preserved. The modification of models is controlled through different fine control mechanisms. More importantly, the model modification through edit operations allows that stochastic properties of model elements to be under control. Additionally, generating model histories is also possible by this approach.

To summarize this work, the generated models conform to their metamodel, so **C1** is fulfilled. Due to the complexity of OCL in general, OCL constraints are not fully supported, so **C2** is partially fulfilled. Generation of arbitrary complex structure are handled through defining appropriate high-level edit operations that create those structures in models, so **C3** is fully supported. The generation of models is supported through different controlling mechanisms. Thus, fine control of generation process is fulfilled and **C4** is supported. The special treatment of model generation through edit operations allows the stochastic properties of a model to be under control. Additionally, this way, model histories can be also generated. Therefore, both of **C5** and **C6** are fulfilled.

Approach of [Laurent et al., 2013] This research work is concerned with generating processes (behavioral) models. Such test models are reported to be used in different domains, e.g. in business, software and medical fields. Generation of process models was motivated by replicating operations which are performed by modelers when creating them. Such operations are typically high-level operations which preserve work-flow patterns in the models.

The generation of process models is done by employing multi-objective genetic algorithms. Genetic algorithms are heuristic search methods. They scan the search space for the optimum or near to optimum point of the specified objectives. They are successfully used in domains where it is quite hard or impossible to find the optimum point due to complexity of the problem or the search space. Genetic algorithms are motivated by the natural evolution in which a population is evolved one generation after the other. In each step the best members of the population survive and the least elite ones are eliminated [Michalewicz, 1996, Reeves and Rowe, 2002]. The evolution occurs through exchanging of individuals' genes as well as mutation in the genes.

In the fist step of a genetic algorithm, a collection of initial individuals are created (usually randomly) and is refereed to as the "population". The properties of interest are encoded in vectors as "genes". In each iteration of the algorithm, these genes can be inherited, combined or mutated through the "inheritance", "crossover" and "mutation" operations respectively. The algorithm selects "fittest individuals" of the population in

each iteration and tries to generate new individuals by performing previously mentioned operations on their genes. The selection of the fittest ones, is based on a "fitness function" which evaluates the individuals' "fitness" with respect to objectives of interest. There are different selection strategies that can be applied in practice. The purpose of performing gene operations is to cover and scan the search space as well as improving the optimum point, through exchanging of information in genes. The stopping condition of the algorithm is usually when there is no better improvement in the optimum point after a specified number of iterations.

In the work of [Laurent et al., 2013], three objectives are defined for generation: (a) size of the process i.e. how many nodes it has, (b) the number of each element, and (c) constraints on the static structure of the process, e.g. the number of outgoing edges for a "ForkNode". The initial population were made of a simple process, i.e. the one with just two start and end nodes, and some user-defined processes. For altering the models, they used just mutation operations. Their set of mutation operations consisted of 18 high-level change patterns which were defined on processes. For having more control on the generation processes, they assigned probabilities to change patterns. The pattern with higher probability were more likely to be applied than a pattern with a lower probability. Such probabilities have consequences on the final generated processes. This concept is similar to our work presented earlier in [Pietsch et al., 2011, 2012a], in which the model generation is stochastically controlled through assigned probabilities to edit operations of models (see Section 3.4.1)[4].

In the presented approach, the correctness of the generated processes according to their metamodel is satisfied and **C1** is fulfilled. Supporting OCL constraints is not covered in this work, so **C2** is not addressed. Complex edit operations are supported by change patterns which were applied to the processes during their generation, so **C3** is supported. Controlling the generation process is defined by probabilities assigned to the change patterns, so **C4** and **C5** are both fulfilled. The generation of model histories is not considered and **C6** is disregarded.

Approach of [Xiao et al., 2014] The authors tried to generate large test models for scalability and performance evaluation of model transformations in industrial applications. Their approach is to randomly generate test models which are conforming to the input metamodel of the transformation. To achieve that, their algorithm is divided in four phases.

The first phase is to define a configuration model. In such configuration the user specifies the configuration needed to guide the generation process. For example, the user defines the meta-classes whose instances are roots in the model and if there is a unique root. Roots are the starting points in the generation process. Moreover, other constraints such as default range constraints, element range constraints, attributes range constraints, total size of the model etc., are defined. Such constraints define default or required values when instances of a meta-element are generated.

[4] The work of [Laurent et al., 2013] is presented later than the concepts of this chapter [Pietsch et al., 2011, 2012a], so the problems which this chapter addresses were not solved before this dissertation.

In the second phase, the model elements and their attributes with required values are generated based on the configuration model. The third phase is devoted to generation of all relationships between previously generated model elements. The last phase is the validation phase. In this phase other user-defined constraints as well as OCL constraints are evaluated against the generated test models. Their approach cannot handle such constraints in the generation phase and they have to be evaluated afterward. If such constraints are valid, the generated model is returned otherwise appropriate errors are reported.

Conformity of the generated test models to their metamodel is supported in this approach, thus **C1** is fulfilled. The validity of OCL constraits and other user-defined constraints are checked after the models are generated and they are not considered withing the generation process, so **C2** is not supported. Supporting the complex structures is not done and **C3** is disregarded. The generation process is configured by specifying range constraints and total size of the models so **C4** is partially supported. Since the generation process is random and no other stochastic properties are supported, **C5** is not fulfilled. Model histories are not considered and **C6** is not fulfilled.

3.2.2 Direct Formal Approaches

This category considers the methods of generating test models which have roots in formal and mathematical foundations. Such approaches are not quite frequent. Although the mathematical foundation of formal methods provides very precise framework for generating models, they are still not capable of handling common difficulties in generating models. For instance, it is not possible to appropriately handle OCL constraints and the generation process can hardly be controlled. In this section we just found three related works. Since they are principally the same idea developed and elaborated over time, we present all of them under one approach.

Approach of [Ehrig et al., 2006, 2009, Taentzer, 2012] In this approach graph grammars have been employed to generate instances of a given metamodel. In the latest publication, theoretical aspects of graph grammars are discussed while in the earlier publications, the generation algorithm and its implementation are covered. In graph grammars, an input graph is modified, i.e. transformed, to an output graph based on sets of defined rules. A rule is a mapping between two graph structures, known as left-hand side (LHSG) and right-hand side graphs (RHSG) of the rule. Whenever the LHSG of a rule is found within the given input graph, the input graph is transformed to the output graph in accordance to the rule's RHSG.

For having more control over the transformation, "application conditions" are defined. The application conditions are sets of conditionals which should be met before the transformation can be performed. They principally check whether some specific patterns of interest are un/available in input and output graphs.

In this approach, the metamodel is first transformed into a typed graph. Then appropriate rules were defined for transformation of the graph. The rules were defined

in three layers and applied one layer after the other. The approach should be started from an empty graph and then the three layers of rules be applied to it. The layer 1 rules create instances of each class in the corresponding metamodel. The rules of layer 1 are applied as often as desired. It means that the number of application of these rules have to be defined by the user. Typically, the users specify how many of each rule in layer 1 should be applied. Such numbers can also be randomly set.

In layers 2 and 3, the rules create links between the already created instances of the classes in the metamodel. Layer 2 rules are responsible for creation of associations with 1-multiplicity restriction. As an example, a rule searches for patterns of two nodes without a direct association and connects them by creating an association between them. Another example is to find one node, creating a new node and then connecting them. These rules should be applied as long as the multiplicity constraints are satisfied and the rules can be applied. Layer 3 rules, create associations with $0..n$-multiplicities. These rules can be applied arbitrarily often, since such multiplicities are optional. For creating attribute values, they used a post processing step, in which values are created once the instances of the metamodel are created. OCL constraints are hardly supported.

By this approach the correctness of generated instance is guaranteed, so **C1** is satisfied. The OCL constraints are not completely and directly employed in their approach, so **C2** is partially fulfilled. The creation of complex structures is not supported and **C3** is not met. Generation control of the test models are just restricted to the number of rules which are applied in layers 1 and 3, so **C4** is just partially supported. Neither controlling statistical properties of the generated instances nor generation of model histories are covered in their approach, so **C5** and **C6** are not met.

3.2.3 Indirect Approaches

Opposed to the approaches mentioned in Sections 3.2.1 and 3.2.2, which directly generate test models, indirect approaches generate models by employing other tools. Roughly speaking, first the metamodel and user requirements of generation are translated into equivalent specifications of a tool. The tool then finds an answer which satisfies the specifications. Later, the answer is translated back to its equivalent model.

A very famous tool which is extensively used in this regard is "Alloy" [Jackson, 2002, Jackson et al., 2014]. Alloy is an "SAT" solver which finds instances of an SAT problem, i.e. solutions which satisfy the specified constraints in an SAT problem [Milicevic et al., 2014]. SAT is the abbreviation of the "Propositional Satisfiability Problem" which is shortened to "Satisfiability" or simply SAT. In an SAT problem, we are interested to see if there is a solution for a Boolean formula which results in the formula to be evaluated to True [De Moura and Bjørner, 2011]. If so, the formula is said to be satisfiable, otherwise it is said to be unsatisfiable. A Boolean formula is a propositional logic formula which is formed from Boolean variables related to each other with the propositional operators of $\neg, \wedge, \vee, \Longrightarrow, \Longleftrightarrow$ and parentheses. SAT solvers typically work with the Conjunctive Normal Form (CNF) of logical formulas which is shown to be efficient in their automatic evaluation. A CNF form of a formula is its reformulating in terms of conjunction (AND) of clauses, in which a clause is a disjunction (OR) of literals, and a

literal is a Boolean variable or its negation (NOT). In other words, a CNF form is an AND combination of ORs. In a broader sense, Satisfiability Modulo Theories (SMT) additionally considers other constraints in SAT problems, such as arithmetic constraints and bit vectors. Therefore more extensive types of problems are addressed by SMT and SMT solvers [Barrett et al., 2009]. A SMT problem is typically reformulated in terms of an SAT problem and then its satisfiability is checked.

In the coming approaches which we describe in this section, typically, the metamodel as well as other related constraints are first translated into the common constraints language of Alloy[5]. Later, the Alloy Analyzer forms a CNF formula from the specified constraints and solves it in order to find a solution which satisfies the constraints. If there is no solution, the problem is unsatisfiable and the specified constraints are contradicting. Once a solution is found, it is then translated back into a model. The model which is generated in this way satisfies the original specification. In the case that more than one solution exists, the Alloy Analyzer can deliver other solutions as well. Therefore, more than one model can be generated which satisfies the original specifications.

Generally, the basic problem of such approaches are the difficulties in translation of original specification into their equivalent counterparts in Alloy. Such difficulties arise because Alloy and similar SAT solvers are quite different in nature in comparison to the metamodel constraints and the generation requirements [Sen et al., 2009]. Moreover, translating the specification into Alloy, finding their solution by the Alloy Analyzer and translating the solution back to a model is very time consuming. There is also no control on how other solutions of specifications are found by Alloy and therefore their models counterparts will miss some desired properties, e.g. stochastic properties.

Now we take a closer look at the indirect approaches, and we evaluate our criteria against them.

Approach of [McQuillan and Power, 2008] The authors were interested to develop a language-independent metrics-specific metamodel for metrics tools. The problem is reported to be originated from the variety of software metrics which are addressed or proposed in literature. Such metrics are not only reported to be quite diverse, but also in some cases incomplete, ambiguous and disputable which make developing and updating metrics tools to be frustrating. To address the problem, the authors proposed a MOF-compliant metamodel for handling different metrics. The metamodel is quite general and independent of any languages or model element types. This way they managed to define software metrics in a standard and unambiguous manner. The metrics metamodel was then used to develop different metrics tools.

In order to generate instances of the metamodel and test their metrics tools, they used Alloy. In this regard, they first translated the metrics metamodel as well as its associated OCL constraints into the Alloy constraint language. The Alloy Analyzer was then invoked in order to test the well-formedness of the metamodel and its OCL constraints, i.e. if the metamodel as well as OCL specifications are fulfilled. The analyzer delivered

[5] Two typical kinds of constraints in Alloy are: (*a*) Facts, which are always true for whole specifications and (*b*) Predicats, which are parametrized formulas that are locally applied and evaluated.

solutions of the specified constraints. The Alloy solutions were then transformed back into models.

In this approach the generated models are conforming to the metric metamodel and **C1** is fulfilled. Regrading OCL constraints, the authors used very restricted subset of OCL constraints in their approach, so **C2** is partially supported. Creating complex structures, i.e. **C3** within models in not relevant in this work. Controlling the generation of models is not possible through Alloy, so **C4** is not fulfilled. Controlling the stochastic properties of the generated models, i.e. metrics, as well as model histories are also not relevant in this approach, so **C5** and **C6** are not relevant here.

Approach of [McGill et al., 2009] The authors were interested in generation of ORM (Object-Role Modeling) models for testing the tools that generate codes from such models. ORM models are widely used in information systems and databases to conceptually model a system and perform queries on it [Halpin, 2006]. ORM allows mapping between conceptual and logical levels through natural languages and intuitive diagrams.

The authors mention two difficulties of generating ORM models: first, the validity of the generated models in the ORM metamodel and second, producing reasonable-sized test suites that cover wide range of ORM features for testing purposes. For generating test models, they developed a tool called "ATIG". ATIG takes a set of ORM features which should be present in the generated models. The set of features is an encoded subset of the ORM metamodel which are more relevant for the testing purposes. The tool then generates a set of models that exhibit different combination of such features, known as "test plans". Combining the features, i.e. producing test plans, are done by a tool called "Jenny".

To find an instance, i.e. a model, that satisfies a specified combination of features, they used Alloy. They alternatively used Jenny and Alloy to prune the combinations until the Alloy Analyzer is able to find an instance which satisfies the combination. Once an instance is found, ATIG translates it to an ORM model.

In this approach, the generated models are correct according the subset of ORM metamodel which is used for testing, so **C1** is partially supported. OCL constraints are not relevant in this work, thus **C2** is not considered. Complex structures is generated through combination of features. Although combination should be pruned to find valid instances, we can assume that **C3** is partially fulfilled. Since Alloy is used to find instances, there is no control on the way the instance is generated. Therefore, the process of generating the ORM models is not under control and **C4** is not satisfied. The stochastic features of the generated models and model histories are not regarded, therefore **C5** and **C6** are not satisfied.

Approach of [Sen et al., 2009] The authors have considered generating of test models for testing of model transformation. The generation strategy is based on constraint satisfaction. Such approach is reported to simultaneously satisfy the metamodel constraints as well as testing requirements. To this aim, they developed a tool called

"Cartier" that translates the EMF-based input metamodel of the transformation into constraint language of Alloy.

Once Cartier translated the metamodel into the Alloy language, the Alloy Analyzer solves it in order to find a satisfiable solution. Cartier then translates the Alloy solutions back to the instances of the metamodel and delivers them as test models. Since Alloy can deliver more than one solution of the constraints, the generated models are the corresponding test models of such solutions. The translation of the metamodel to Alloy is based on the partitioning and model fragments concepts presented in [Fleurey et al., 2004]. Regarding the OCL constraints, they have to be manually translated into Alloy. It is reported that not all OCL constraints can be transformed to Alloy since their languages are not designed with the same intention in mind.

In this approach, the generated test models are conforming to the input metamodel of the transformation, so **C1** is satisfied. Since not all of OCL constraints are expressible in the Alloy constraint language, **C2** is partially supported. Complex structures of interest are supported by the concept of model fragments and partitioning, so **C3** is partially fulfilled. Since Alloy solutions are used to generate test models, controlling the generation process is not possible, so **C4** is not fulfilled. Statistical properties of the generated test models and model histories are not considered, so **C5** and **C6** are not covered.

Approach of [Williams and Poulding, 2011] The authors were interested to generate instances of a metamodel. The metamodel was not directly given, rather the metamodel was generated from the grammar of a language which was defined in "Xtext". Xtext[6] is a framework which allows specification and generation of general programming as well as domain specific languages. It provides a complete infrastructure for developing languages such as parser, linker, compiler or interpreter. It is also fully integrated into the Eclipse IDE.

Once the grammar of the language is specified, Xtext produces a metamodel of the grammar. This way instances of the language grammar will be models conforming to the generated metamodel. In order to generate the instances of the language, they employed the "Grammatical Evolution" (GE) algorithm. GE can be regarded as a genetic algorithm which is tailored to produce instances of a grammar, i.e. programs. To this aim, they encoded the grammar rules (phenotype) into GE algorithm codes (genotype) using sequences of integers (codon). Each integer in the sequence was correspondent to a production rule in the grammar. The mapping started from the first production rule of the grammar. Lets assume that the rule has n non-terminal options which can be chosen and continued in the next step. The modulo operation of the first codon c_1 to n, i.e. "$c_1 \mod n$", was used to choose the next rule in the grammar. The process continues similarly for the next rule, using the second codon c_2 in the sequence. The choose of a terminal out of n terminals was done similarly. The number of codons, i.e. the length of genotype was specified by the user. The mutation and crossover operators were used to generate new models randomly from the set of population.

[6] `http://eclipse.org/Xtext`

The objective function of the GE algorithm was planned to be used to control the properties of the generated models. For instance one function might just give random models while another might impose that generated models are distinct and not similar. Unfortunately the objective functions were not defined and the authors reported them as the work which should be done.

Once an instance of the grammar was produced by the GE algorithm, it was transformed to its corresponding model element in the metamodel. Xtext allows generation of metamodels conforming to EMF-Ecore. Their transformation did not support assigning values to the attributes of the generated models.

Since the models are generated from the instances of the grammar, they are conforming to the corresponding metamodel of the grammar. Therefore, **C1** is supported. OCL constraints are not discussed, so **C2** is disregarded. Support for complex structures are not considered and **C3** is not fulfilled. Since the number of codons are limited and specified by the user, it is not guaranteed that models with big sizes are generated. Additionally, covering all of the grammar rules is not possible this way, so some model elements and structures might not be produced. Thus, we conclude that the generation of models are quite limited and there is no control on the generation process, so **C4** is not satisfied. Stochastic properties of the generated models and model histories are not considered, so **C5** and **C6** are not satisfied.

Approach of [Svendsen et al., 2012] In this work, authors were interested to generate train station models. The train stations were modeled by "Train Control Language" (TCL). TCL is a domain-specific modeling language which allows modeling of train stations. It supports configuration and code generation for controlling the signaling systems of stations.

Due to the special usage of train station models, it is required that the generated models have special properties which is more or less related to the way they are graphically presented. This needs other extra characteristics which should be met by the generated models. The authors used constraint language of Alloy to express the required specification as well as the metamodel of the train stations. Alloy Analyzer was then used to find solutions of the specified constraints. The solutions were then translated back to train station models presented to user as diagrams.

In this work, the generated models are correct based on the metamodel, so **C1** is fulfilled. Although some user specification are covered in this work, they are not expressed in OCL constraints and therefore **C2** is not supported. Complex structures of the stations is not possible by Alloy, so **C3** is not fulfilled. Controlling the generation of models is not considered, so **C4** is not regarded. The stochastic properties of the generated models and model histories does not seem to be relevant in this work, so **C5** and **C6** are no relevant here.

3.2.4 Summary of the Reviewed Literature

Before moving to the next section, we try to summarize the outcomes of the studied related literature. In addition to studying the approaches proposed in the related works, we have investigated them to see whether they fulfill the criteria of **C1**-**C7** mentioned in Section 3.1. Table 3.1 summarizes the results of our findings regarding fulfilling the aforementioned criteria.

Approaches	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**
[Fleurey et al., 2004]	✓	✗	≈	≈	✗	✗	✗
[Baudry et al., 2006]	✓	✗	≈	≈	✗	✗	✗
[Brottier et al., 2006]	✓	✗	≈	≈	✗	✗	✗
[Lamari, 2007]	✓	✗	≈	≈	✗	✗	✗
[Wang et al., 2008]	✓	✗	≈	≈	✗	✗	✗
[Mougenot et al., 2009]	✓	✗	✗	✗	✗	✗	✗
[Laurent et al., 2013]	✓	✗	✓	✓	✓	✗	✗
[Xiao et al., 2014]	✓	✗	✗	≈	✗	✗	✗
[Ehrig et al., 2006, 2009]	✓	≈	✗	≈	✗	✗	✗
[McQuillan and Power, 2008]	✓	≈	–	✗	–	–	–
[McGill et al., 2009]	≈	✗	≈	✗	✗	✗	✗
[Sen et al., 2009]	✓	≈	≈	✗	✗	✗	✗
[Williams and Poulding, 2011]	✓	✗	✗	✗	✗	✗	✗
[Svendsen et al., 2012]	✓	✗	✗	✗	–	–	–

✓ Fulfilled ✗ Not Fulfilled
≈ Partially Fulfilled – Not Relevant

Table 3.1: Fulfilling criteria of **C1** to **C6** - Summary of the related literature.

As shown, all of the approaches have fulfilled criterion **C1**, i.e. their generated models are conformal to the corresponding metamodel. Just the approach of [McGill et al., 2009] has considered a subset of the initial metamodel, but the generated models were also conforming to the subset.

Regarding extra constraints on models which are expressed in OCL (**C2**), just three approaches handle them partially. The main reason is the difficulty of satisfying OCL constraints when valid instance models should be generated. [Ehrig et al., 2006, 2009] checked the satisfaction of OCL constraints as a post-processing step when instances were already generated. Contrary, [McQuillan and Power, 2008, Sen et al., 2009] considered satisfying OCL constraints in the generation process. In this regard, they translated OCL constraints to the constraints language of Alloy and let the Alloy Analyzer to find instances which satisfy OCL as well as metamodel constraints. This approach has been partially successful since the translation of OCL constraints to the Alloy constraints is not easy due to the fact they have been designed and developed with different aims in mind.

Creating complex structures of interest (**C3**) is partially supported in the existing approaches. The support is mostly dominated by the approaches for testing model transformations. In this regard, a common practice is the partitioning approach which is initially suggested by [Fleurey et al., 2004]. In this approach, the generated complex structures are limited to the combination of instances of the metamodel partitions. This way, just limited structures are supported and generation of arbitrary complex structures are not possible.

Controlling the generation process (**C4**) in the approaches motivated by [Fleurey et al., 2004], is mostly limited to covering items, how and how many of them are combined together. Therefore, the controlling mechanisms for the generation process are often limited. Regarding the indirect approaches discussed in Section 3.2.3, the Alloy Analyzer just delivers valid instances and the user has no control on how instances are generated. Generally, controlling the generation process in quite limited in the related works.

Generating models with desired statistical properties (**C5**) is just addressed in two approaches. [Mougenot et al., 2009] generated huge models in which all possible structures within a model of a given fixed size are uniformly generated. This approach is very rigid and does not allow statistical properties of interest to be created in the generated models. [Laurent et al., 2013] generated process models using genetic algorithms. In their approach generation were controlled by assigning probabilities to the change patterns which basically acted as the mutation operators to alter the models. Moreover, the existing approaches have not considered the generation of model histories (**C6**) and the related challenges.

The last but not least, none of the existing approaches are useful in the domain of model differencing and model versioning, since they are unable to produce difference which can be applied between models (see Section 2.3.1). Therefore criterion of **C7** is not supported at all.

3.3 SiDiff Model Generator

As we saw in the previous section, the state-of-the-art tools and methods for generating test models, do not fulfill four of our criteria. The first one is that none of existing generator support generating of differences which is mostly suitable in the domain of model differencing and model versioning (**C7**). The second one is limited or improper controlling mechanisms to let users to control the generation process of models (**C4**).

The thrid one is that almost none of the proposed methods takes the stochastic properties of generated model elements into account (**C5**). Just one of the existing approaches ([Mougenot et al., 2009]) considers limited generation of model elements uniformly. Although this approach is good for some purposes, it is not appropriate to consider the uniform distribution in other applications. Particularly, in the field of model differencing and model versioning such assumption does not hold, e.g. in class diagrams we will show that the distributions of model elements are highly skewed (see Chapter 5) and generating test models for model differencing and model versioning tools just with the uniform distribution is quite of little value.

The last criterion which was not considered was that none of the current approaches take the generation of model histories into account (**C6**). Model histories are very essential in model versioning and model differencing tools. They are used in model repositories as well as model evolution analysis and presentation tools. Therefore it is quite essential that we cover these four criteria by our generator.

In this regard we introduce the SiDiff Model Generator (SMG) which addresses the shortcomings mentioned earlier. In the first step, Section 3.3.1 provides basic requirements which are needed for following the material in subsequent sections. Section 3.3.2 provides an overview of SMG. Section 3.3.3 describes the scenarios where SMG can be employed to generate test models. Section 3.3.4 discusses how probabilities of model elements are interpreted. Section 3.3.5 explains how models are modified and generated and finally Section 3.3.6 explains how models are edited and which edit operations can be applied to models.

Handling three criteria of **C4** to **C6** are discussed in detail in Section 3.4. The evaluation of SMG is presented in Section 3.5. The chapter ends with a summary in Section 3.6.

3.3.1 Requirements

Before introducing the SMG in the next section, we take some assumptions that will be valid through the rest of the chapter. Moreover, in order to better express the concepts and ideas, we also provide the necessary definitions[7].

Assumption 1. Models are typed with respect to a metamodel, i.e. each model element has a type in its metamodel.

The previous assumption is based on the fact that our intention is to produce models conforming to a given metamodel.

Definition 3.1 (Edit Operations)**.** Suppose that a metamodel is given. For each model element type in the metamodel, there is a set of operations that are defined on that type and are used to modify the instances of that type. Moreover, there are edit operations which are composed of simpler operations and modify elements with different types in one step.

As we know, the abstract syntax graph representation imposes the basic edit operation on models. The basic graph edit operations are *creating, deleting, moving* and *changing* of elements, attributes and references [Kehrer et al., 2013d]. As we discussed in Section 2.4.2, such operations are considered as low-level edit operations. We argued that these low-level operations can be combined together to form high-level edit operations [Kehrer et al., 2011]. Later in Section 3.3.6, we will see that SMG can support both the low-level and high-level edit operations for modifying models.

[7] Few concepts have been briefly reviewed in Section 2.3, but for the sake of clarity we formally introduce them here in detail.

Definition 3.2 (Edit Step)**.** An edit step is a edit operation supplied with its concrete arguments that can be executed on a model. An edit step is also referred to as the "Invocation of an Edit Operation".

As an example, one can consider UML class diagrams in which creating a class with a given name within a parent package, can be done for instance by executing an edit step of `createNamedClass(Name,ParentPackage)` on the desired package (instance of the model type), where `Name` and `ParentPackage` are concretely specified. Another example can be deleting a state in state chart diagrams by `deleteState(State)`.

Definition 3.3 (Base Model)**.** A base model is an instance of a given metamodel which is regarded as the starting point in the modification process through execution of edit steps. A base model can also be an empty model with no model elements.

Definition 3.4 (Derived Model)**.** A derived model is the resulting model after executing one or more edit steps on a base model. The derived model is regarded as the successor version of the base model.

Definition 3.5 (Patch)**.** A patch is a sequence of edit steps which are to be applied to a base model.

A *patch* is also referred to as an *asymmetric difference* (see Section 2.3.1 as well as [Kelter and Schmidt, 2008, Wenzel, 2010]). In this chapter from now on, we use the terms of "patch" and "difference" interchangeably.

Another important issue which should be clarified is that the set of constraints on the metamodel might be contradicting. The set of constraints on the metamodel principally consists of intrinsic constraints of the metamodel as well as user-defined OCL constraints. The interesting question is that whether constraints of a metamodel can be satisfied in instances or not. We have the following definition:

Definition 3.6 (Realizability of a Metamodel)**.** We say a metamodel is realizable, when there is an instance of it which satisfies all constraints of the metamodel.

The satisfiability of metamodels constraints has been studied in the literature so far, for example [Cabot et al., 2008]. It is not difficult to establish contradicting constraints, although in many practical applications it might not always be the case. To clarify the problem, consider Figure 3.2 in which a simple class diagram is depicted.

There are two classes of Paper and Researcher in the diagram. The association of Writes imposes that a Paper has at most two Researchers as authors while at the same time the association of Reviews requires that a Paper is refereed by exactly three distinct Researchers. The previous constraints on the number of Papers and Researchers, mathematically translates to the following system of equations which has just two solutions of either zero or infinity for both variables:

$$\begin{cases} |\text{Researcher}| \leq 2\,|\text{Paper}| \\ |\text{Researcher}| = 3\,|\text{Paper}| \end{cases}$$

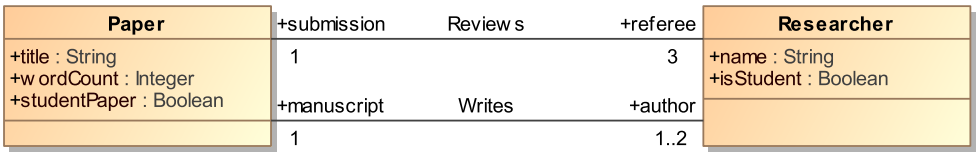


Figure 3.2: Example of contradicting constraints in a metamodel (Adapted from [Cabot et al., 2008]).

Although the satisfiability of constraints has been studied so far, it is not quite clear what is the solution region for a realizable metamodel. Due to this fact, there might be cases in which the solution region is so limited and just quite limited number of instances will satisfy the metamodel constraints. For a base model, this means that probably just a limited subset of the whole possible patches yield valid derived models which satisfy all constraints of the metamodel. Moreover, since edit steps might refer to model elements with specific types or to their positions in the base model, not every specified edit operation can be executed at a specific state of a model.

Since the previously discussed issues are out of the scope of this dissertation and since the contribution of this chapter is to propose fine control mechanisms for the generation of models, we make another assumption as follows:

Assumption 2. The metamodel is realizable and the solution space of its constraints is diverse enough that lets edit steps be executed with satisfactorily number of times and orders without going into an invalid derived model.

Before going to the next section, we provide two more definitions which are required later in the subsequent sections. They will help us to express the ideas more easily.

Definition 3.7 (ContextType)**.** A ContextType is an element of the metamodel.

Definition 3.8 (Context)**.** A Context is an instance of a given ContextType, i.e. a model element which has the type of "ContextType" in the metamodel.

3.3.2 Overview

The *SiDiff Model Generator* (SMG) is a model generator which is designed to address the shortcomings in the state-of-the-art approaches of generating models. It fulfills the criteria **C1** to **C7** mentioned in Section 3.1. Specially, criterion of **C7** which is generating differences between models is fully supported.

SMG is capable of creating new models from scratch or modifying existing models. In both cases the derived model is guaranteed to be correct according to its metamodel (**C1**). It also satisfies OCL constraints, although due to the complexity of such constraints only a subset of it is supported, therefore **C2** is also partially fulfilled. SMG supports complex as well as elementary edit operations on models (**C3**). Complex edit

operations can be arbitrarily defined. Each complex edit operation can be a collection of simple or other complex operations. This provides much flexibilities in modifying models.

SMG provides different controlling mechanism to direct and control the generation process (**C4**). In the modification process of the base model, the frequencies of the edit operations or their distributions are also under control (**C5**). This concept is also extended when a sequence of related models, i.e. model histories are going to be created (**C6**). Such features, give SMG the ability to mimic and replicate the evolution of models (see Chapter 5 and Chapter 6).

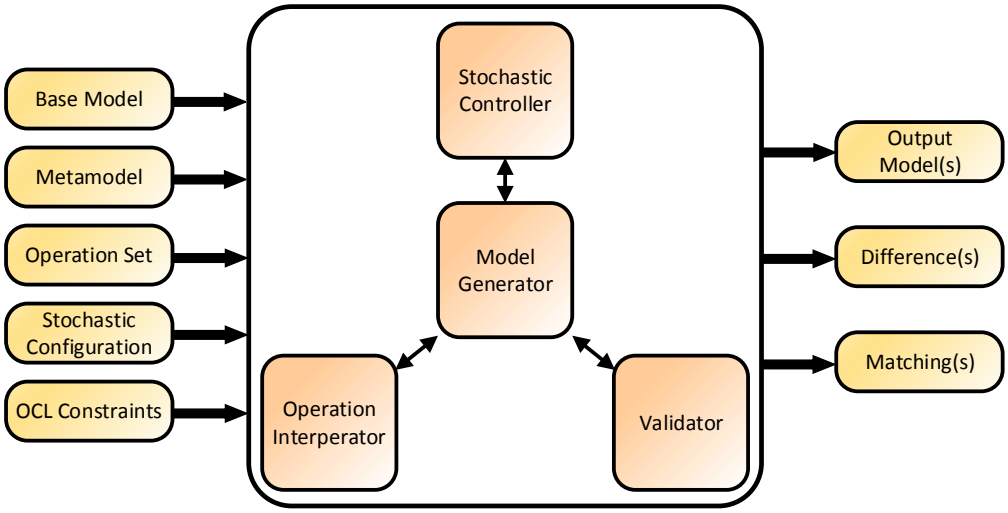


Figure 3.3: The SiDiff Model Generator - Overview.

Figure 3.3 gives an overview of the SMG and its components as well as its inputs and outputs. As input, SMG takes a base model in addition to its corresponding metamodel. Other OCL constraints are also given to the system to prevent undesired properties within the derived model(s). The set of legitimate edit operations which are defined based on the metamodel, is also given to the system. Based on the set of edit operations, a configuration file for controlling the modification process is provided to the system.

SMG has four main components. The *Stochastic Controller* is responsible for invocations of edit operations based on the stochastic configuration file. An edit operation is selected from the defined set of edit operations and its arguments are either selected from other model elements in a controlled manner, or created on demand. This component plays an important role in realistic modification of the base model. We will cover this in detail in Section 3.4. The *Operation Interpreter* is responsible for applying a selected edit operation to the input model, in other words it invokes an edit operation. In the end, the *Validator* checks the validity of the modified model. All of these components are glued together via the *Model Generator* which is their common communication interface.

As its primary artifacts, SMG generates differences between models, fulfilling crite-

rion of **C7**. As resulting products of the generated differences, SMG additionally delivers a derived model or a set of derived models, i.e. model histories. Another output is the *Matching* (see Section 2.3.1) between model elements in the base and derived models, showing their correspondences. A protocol of the applied edit operations is also produced as a *Patch* that allows recreation of the derived model from the base model at any time. If a model history is created, the corresponding patches and matchings between each two subsequently generated models are also generated.

It should be mentioned that for any provided metamodel, there is out of the box support for automatic creation of set of edit operations and an automatic stochastic configuration file. Although both provide a fast setup for the system, they do not deliver good quality derived models and should be tailored by the user.

3.3.3 Main Usage Scenarios

Before mentioning the usage scenarios of SMG, it is good to make it clear that the SMG's foremost products are patches or differences which are applicable to the given base model. The derived models are by-products, although from the user perspective they might be more important. In this regard, the generated patches are only applicable to the same base model which they are at first intended for. The only exception is when the base model is an empty model. SMG can be used in four main usage scenarios:

U1 Creating new models from scratch.

U2 Modifying existing models in a controlled manner.

U3 Creating a set of models with similar properties.

U4 Creating model histories.

Use case **U1** is achieved by generating various patches which are applicable to an empty model. Use case **U2** is achieved by repeatedly controlled creation of edit operations based on the current state of the model and applying them. Use case **U3** principally can be achieved when the set of applied edit operations are chosen in a systematic manner. For instance they can be chosen based on specific statistical distributions.

At the first sight, the use case **U4** might seem to be a special case of the use case **U3** in which a model history is created by repeatedly applying the edit operations with the same properties. But this is not the case, since there are some properties that are related to the model history which cannot be controlled by just repeatedly considering the use case **U3**, for instance life time of model elements within a history.

3.3.4 Interpretation Modes

As we mentioned in Section 3.3.2, SMG supports criterion **C5**, i.e. it allows the control over the frequencies or distributions of the created differences. In this regard, two interpretation modes for the frequencies of the generated differences are considered, each having its own application.

Literal Interpretation Mode: In this mode, both the number and the kind of edit operations which should be generated are given to the system via a configuration file. Such specification is considered to be literal, i.e. the edit operations are generated exactly as they are specified. The Stochastic Controller is responsible for doing the rest, i.e. both where the operations are applied and which parameters they need, are handled in a stochastic way based on the configuration file. In this mode, the size of the generated difference is implicitly defined and is equal to the sum of the number of specified edit operations.

As an example, in the UML class diagrams, one can consider the operation of `createNamedClass(Name)` which is specified in a configuration file to be generated five times. In this case, the derived model will have exactly five more classes compared to the base model. Where these classes are created or which names they will have is decided by the Stochastic Controller.

In this mode a user can specify the quantitative properties of the derived model. This mode is more suitable when one tries to verbatim replicate an observed model evolution which is obtained for example by studying model modifications in real software repositories. Another application can be to generate test models for model merging tools.

Stochastic Interpretation Mode: In this mode, *Probability Mass Functions* (PMF) of edit operations are specified in the configuration file for the system. Contrary to the Literal Interpretation Mode, the size of the difference should be explicitly specified. If the specified probabilities do not sum to 100%, they are automatically normalized by dividing them to the sum.

In this mode the specified size of the difference and PMF of edit operations are two important factors which affect the properties of derived models. The derived models approximately exhibit the specified probabilities. When the specified distributions are skewed with heavy tails, execution of more edit operations yields better results. Like the literal interpretation mode, parameters of the edit operations are selected or generated with the help of the Stochastic Controller. This mode is more suitable for creating sets of differences with similar properties with possibly varying sizes.

3.3.5 Model Modification Process

As discussed in Section 3.3.1, for each ContextType we have a set of edit operations which are applicable to the Contexts with that type. The model modification process consists of five successive steps in which in each step, the selection is done with the help of the Stochastic Controller (see Section 3.3.2):

S1- A *ContextType* is selected.

S2- An *Edit Operation* is selected.

S3- A *Context* is selected.

S4- Other *Parameters* of the edit operation are selected and/or created.

S5- The operation is *executed* on the base model.

In the first step in the modification process, a ContextType is selected in step **S1**. Once a ContextType is selected, an edit operations from the set of available operation for that ContextType, is selected in step **S2**. The selected operation will be applied to a Context with the type of already selected ContextType. The selection of the Context is done in step **S3**. Other required parameters for the selected edit operation is either selected or created in step **S4**. Figure 3.4 depicts the modification process.

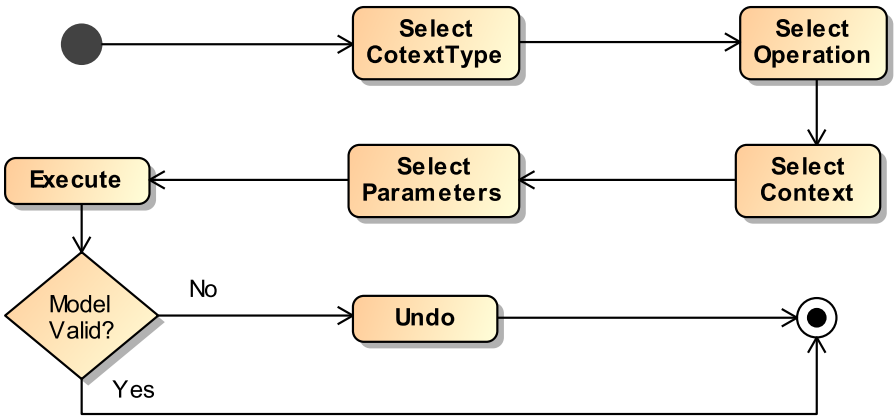


Figure 3.4: SMG - Model modification process.

It should be mentioned that in the previous five steps, step **S1** i.e. selection of ContextTypes might be disregarded in some usage scenarios. This case happens when the set of edit operations consists of high-level edit operations that are not solely defined on one specific ContextType, rather they are defined to act on two or more different ContextTypes. In such cases the selection of a ContextType is not meaningful any more. Moreover, in step **S3** two or more Contexts should be selected accordingly.

Possible Problems in the Modification Process In the model modification process some situations might arise that prevent the whole process to be accomplished successfully. For instance in step **S3** there might be no model elements available for selection in the current state of the model that have the type of ContextType. Another problem might arise for instance in step **S4** where the operation refers to some other model elements as its parameters who are currently unavailable. As shown in Figure 3.4, these problems might be solved by a backtracking mechanism which it is usually time expensive. In order to prevent such blocking situations and avoid expensive backtrackings, the concept of *Index Maps* is devised [Pietsch et al., 2011].

Index maps are initially formed according to the given metamodel, the set of edit operations and the base model. They keep track of the current state of the model as well as those operations that can be successfully executed on the current state. After each successful execution of an edit step, they are updated automatically for the next

selection round. Based on the knowledge in the index maps, the Stochastic Controller filters out those cases which will be unsuccessful in the current state of the model.

3.3.6 Edit Operations of Models

SMG supports different edit operations on model. The set of edit operation are domain specific and are defined based on the provided metamodel. Figure 3.5 depicts the core part of the metamodel used to define sets of edit operations. A set of edit operations, i.e. "OperationSet", consists of different operations. An operation can be a "Simple Operation" or a "Complex Operation". There are four kinds of simple edit operations, Add, Delete, Move and Update.

A complex operation consists of simple or possibly other complex operations. In complex operations the information about already performed operations are accessed through "Links". A link has a source and a target, which allows the access to the "Parameters" of previously executed operation.

As an example consider `createNamedClass(Name,ParentPackage)` which is a complex edit operation in the context of UML class diagrams. This operation is used to create a class with a given name in a parent package. The operation consists of two simple operations: `createClass` and `updateClassName` and one link. The link forwards the result of the `createClass` operation, i.e. the previously created class, as an input parameter to `updateClassName` operation.

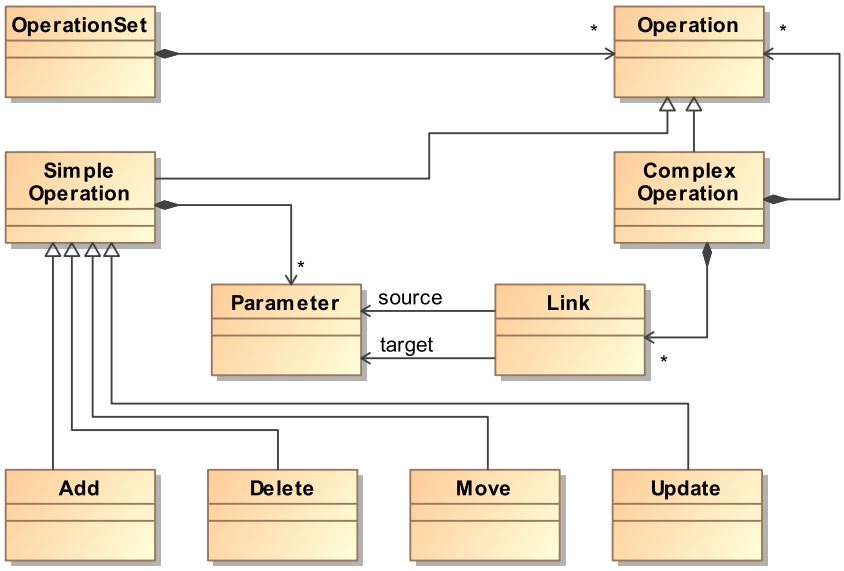


Figure 3.5: SMG metamodel of the edit operations - Simplified core.

As we discussed in Section 2.4.2, simple abstract syntax graph edit operations are not necessarily consistency-preserving, i.e. execution of such edit steps might lead to an

inconsistent state in a model. Therefore, it is desired that models are edited through consistency-preserving edit operations (CPEOs) that lead to the derived models which are consistent. A CPEO typically consists of simple edit operations.

Manually specifying all CPEOs for a model is a tedious and error-prone task. A method for automatically generating CPEOs has been proposed in [Rindt et al., 2014]. In this regard, the authors proposed an EMF-based tool called "SERGe". SERGe analyzes the elements of the metamodel and their relationships to derive CPEOs. Since SERGe principally considers the same set of basic graph edit operations, the resulting CPEOs can be used in the SMG to specify high-level complex edit operations.

3.4 Controlling the Generation Process

In the review of related works (Section 3.2), we showed that the current state-of-the-art approaches do not meet criteria of **C4** to **C6**. In other words, the current approaches are not flexible enough for fine controlling the generation process of models. Moreover, the stochastic properties of generated models have not been considered so far and the generation of model histories is not possible.

SMG supports criteria **C4** to **C6** through its Stochastic Controller module (see Section 3.3.2). It provides fine control mechanisms for the generation process. It also supports generating of realistic models, i.e. the models which have similar statistical properties as true software models. This is accomplished through controlling the stochastic properties of the generated differences. Moreover, the SMG supports the generation of model histories in which again the stochastic properties of changes between model revisions can be controlled.

Fine control mechanisms of SMG allow the modification of models resemble the real modification process as much as possible. As an example of such processes, one can consider few scattered modifications to a base model in order to simulate a debugging activity. In contrast, adding some new model elements close to each other can resemble implementation of new features.

In this section, we describe the contribution of this research work for addressing the previously mentioned drawbacks in the existing approaches. To this aim, we first describe new concepts regarding the qualitative and qualitative aspects of generated differences in Section 3.4.1. How operations and model elements are selected in the modification process is explained in Section 3.4.2. Decision Tables which are used to configure SMG are discussed in Section 3.4.3.

3.4.1 Model Properties in the Generation Process

Frequencies and distributions of edit operations (see Section 3.3.4) are principally regarded as the quantitative aspects of the generated differences. There are also qualitative properties for the generated differences. In this section, we first define required concepts and later we show that how the Stochastic Controller handles such properties.

Definition 3.9 (Effective Properties)**.** The effective properties of an individual regarding a definite purpose, are those properties which have a role regarding that purpose.

As an example, consider the selection (purpose) of a class (individual) in the UML class diagrams based on the metrics of *number of attributes* and/or *number of methods* (effective properties). Another example can be the *number of modifications* an element has already experienced for selecting that element. Also consider the case when an interface is going to be renamed, the effective property can be if its has already been renamed or not, i.e. the *number of renames*.

Regarding the effective properties, two cases are possible:

Case-1 Effective properties of elements are known.

Case-2 Effective properties are not known, e.g. because they are so complex and multifarious or they haven't been studied thoroughly.

In the first case, the domain expert has the opportunity to adjust the Stochastic Controller based on the known effective properties in order to control the quality of the generated differences. In the second case, which is more frequent, some other tools are provided that let a domain expert control the selection process and overcome typical limitations. Both cases will be described in detail in the following.

Definition 3.10 (Fitness Value)**.** The fitness value of an individual is an indicator value which shows the aptness of the individual for a definite purpose. It can be regarded as the value of a function that maps some/all of effective properties of the individual to numerical values.

For instance, consider a function that sums the *number of attributes* and *number of methods* for a class and the result is used whether to select the class or not. As another example, one can select an element based on the *number of modifications* that have already been applied to that element.

In **Case-1** where the effective properties are known and the fitness values can be calculated, it might be possible to obtain a probability distribution function. For instance, how much is the probability that a class gets a new attribute based on its number of attributes. Such information will let us invoke the appropriate number of `createClassAttribute(TargetClass,Name,Type,Visibility)` edit operation to create an attribute. When the range of fitness values is finite, it is a special case in which the probability mass functions can be calculated by calculating the corresponding frequencies. The Stochastic Controller can be adjusted accordingly to produce edit operations based on the derived distribution.

Moreover, the fitness value(s) of each model element is calculated and the element is annotated by the value(s). The annotations then can be explicitly used to select the model elements or prevent redundant operations such as renaming an element twice or deleting a newly created element. For evaluating model comparison algorithms, such kind of operations has no value and cannot be traced.

The fitness values of model elements can be updated immediately after any successful execution of an edit step or they can be calculated in the beginning and are not updated during the whole modification process. Such a decision has a direct effect on the quality of the derived model.

In **Case-2**, where the effective properties and their probability distribution functions are not known or are too difficult to obtain, one can do the selections randomly based on the uniform distribution, although it is quite an oversimplification of the situation. For this purpose, *Selection Policies* are devised which principally work based on the concept of fitness values and give the user the ability to finely control the selection process. For detailed information please refer to Section 3.4.2.

As discussed earlier, SMG can be used to generate model histories (Section 3.3.3). In this case, there are some properties which are related to the history as the whole and cannot be obtained by just simply applying consequent differences to a given base model. For instance, consider the life time expectation of model elements within a history. The number of modification for model elements through the history and their corresponding distributions over the whole history are other examples. SMG is capable of handling these complex scenarios with the help of the fitness values.

To close this section, we will point to another issue which will affect the quality of derived models. As mentioned in Section 3.3.5, the model modification process is done in five steps, the selection of Edit Operations and Contexts have been discussed so far. The selection of ContextTypes also has a direct effect on the quality of delivered models. An example will narrate the story much easier.

Consider the UML class diagrams. Suppose a base model with two packages P1 and P2 and five classes C1 to C5 are given. Also suppose that just two edit operation are provided, `createNamedClass` and `createClassAttribute`.[8] The first function is defined on packages, i.e. when ContextType=Package and the second one when ContextType=Class. For simplification suppose that Contexts are selected randomly based on the uniform random distribution. Also suppose that we are going to invoke the edit operations just four times, with the frequencies of two for each edit operation.

Now suppose that we assign 99.99% as the probability of selecting ContextType=Packge and just 0.01% for selecting ContextType=Class. In this situation, with a great likelihood the two invocations of `createNamedClass` will be executed before the other one, causing two new classes, say C6 and C7, are created first. Then `createClassAttribute` will be invoked two times in which all classes will have equal chance of being selected (each 1/7) in order to have a new attribute.

Conversely suppose that we assign 0.01% when ContextType=Package and 99.99% when ContextType=Class. In this situation, `createClassAttribute` will be most likely executed two times before the other edit operation, causing the attributes appear within already existing classes (C1 to C5). Later two new classes appear most likely with no attributes. To sum up, the way ContextTypes are selected also has a direct effect on the quality of delivered models.

[8] For simplification, we have omitted the associated parameters of the edit operations.

3.4.2 Selection Policies

Up to now, we have discussed the model modification process and the quantitative and qualitative properties in the generation process. In the model modification process, ContextTypes, Edit Operations, Contexts and Parameters have to be selected. To select, the concept of Selection Policies are introduced in this section. *Selection Policies* are adjustable tools which enable us to do the selections in a controller manner. These tools principally work based on the notion of fitness values (see Section 3.4.1). Selection policies have already been used in the field of Genetic Algorithms [Michalewicz, 1996, Reeves and Rowe, 2002, Sivanandam and Deepa, 2008, Pohlheim, 2006].

From now on we will assume that the fitness values which are used, are bounded and nonnegative. In our usage scenario of model modification, this will be a consistent assumption. Currently five different kinds of selection policies are implemented in the Stochastic Controller: *Random*, *Roulette Wheel*, *Simple Ranking*, *Linear Ranking* and *Nonlinear Ranking*.

Random Selection Policy (RSP) The Random Selection Policy is based on the uniform distribution function as its names implies. Every individual will have an equal chance for being selected.

Roulette Wheel Selection Policy (RWSP) In Roulette Wheel Selection Policy, the selection probability of each individual is proportional to its fitness value. An individual with a bigger fitness value will have more chance for being selected. After each selection round, if the fitness values of the individuals are updated increasingly, there is risk that the most fitted individuals, i.e. with the highest fitness values, take dominance in the selection process. This effect can be intensified after each round of selection.

Simple Ranking Selection Policy (SRSP) As mentioned in the RWSP, there is a risk that the most fitted individuals overwhelm the selection process. To avoid such effect, one can sort the individuals based on their fitness values in a non-decreasing order and use the rank (order index) of the individuals as the new fitness values for the selection process. In this selection policy, another problem might arise when there are too many individuals participating in the selection, since the lowest fitted ones will have very small chance of being selected. To solve this, Linear and Nonlinear Ranking Selection Policies are advised which principally work on an extend concept of the SRSP.

Linear Ranking Selection Policy (LRSP) Let n individuals participate in the selection process and they are already sorted non-decreasingly based on their fitness values. The new rank (fitness value) of the i-th individual is calculated as follows:

$$R_{\mathrm{LRSP}}(i,n) = \frac{1}{n}\left(\alpha + 2\,(1-\alpha)\,\frac{i-1}{n-1}\right) \tag{3.1}$$

where $1 \leq i \leq n$ and $\alpha \in [0,2]$, $\alpha \in \mathbb{R}$. "α" is called the selection pressure and three cases can be considered for it:

a) $\alpha = 1$, in this case all individuals will have equal probabilities of $1/n$ for being selected.

b) $\alpha < 1$, in this case those individuals in the beginning of the initially sorted list will get lower probabilities of selection comparing to the ones in the end of the list.

c) $\alpha > 1$, this case is the reverse of the previous case, i.e. the individuals in the beginning of the sorted list will get more change of being selected.

Figure 3.6 will show the LRSP when $\alpha < 1$ and $\alpha > 1$ in contrast to the RWSP. Both are applied to the same individuals.

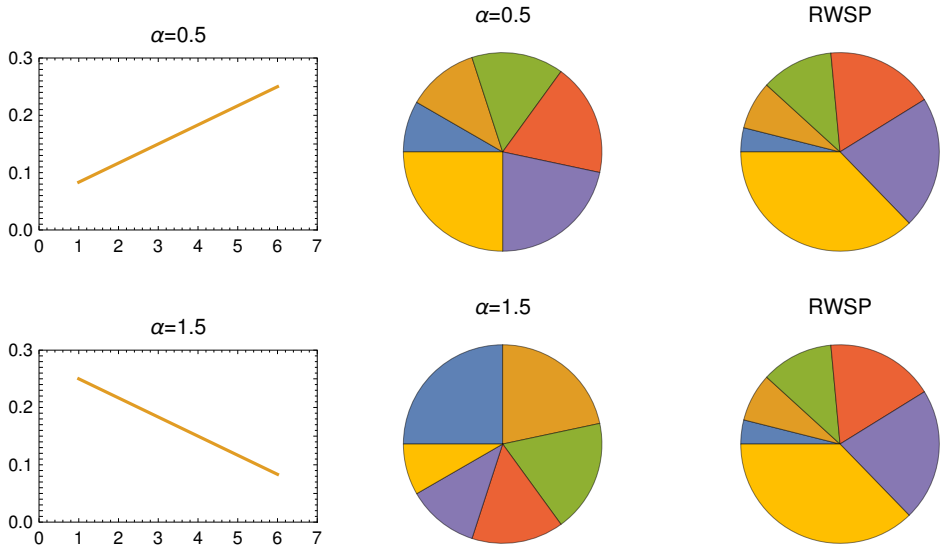


Figure 3.6: Selection probabilities of individuals - RWSP vs LRSP with $\alpha = 0.5$ and $\alpha = 1.5$.

Nonlinear Ranking Selection Policy (NLRSP) Suppose as the LRSP, n individuals are again sorted non-decreasingly. The new rank (fitness value) for the i-th individual is defined as:

$$R_{\text{NLRSP}}(i, n) = \frac{x^{i-1}}{\sum_{j=1}^{n} x^{j-1}} \tag{3.2}$$

in which x is the positive root of the following equation:

$$(\beta - n)\, x^{n-1} + \beta\, x^{n-2} + \cdots + \beta\, x + \beta = 0$$

where $\beta \in [1, n-2]$ is the selection pressure and we assume that $n \geq 3$. The root of the previous equation can be computed for instance by the Bisection or the Newton-Raphson methods [Burden and Faires, 2000, Kaw and Kalu, 2008].

If we consider the linear transformation of $\alpha = (\beta - 1) / (n - 3)$, then $\alpha \in [0, 1]$ will be our new selection pressure which is better matched to the presentation of the LRSP.

When α tends to 1, the selection probabilities of the individuals, i.e. their new ranks, increases in a nonlinear way, but when α tends to 0 then probabilities of selection for the individuals tend to become equal and when $\alpha = 0$ then all of them will have equal chance of being selected. Figure 3.7 shows the NLRSP in contrast to the RWSP when both applied to the same individuals.

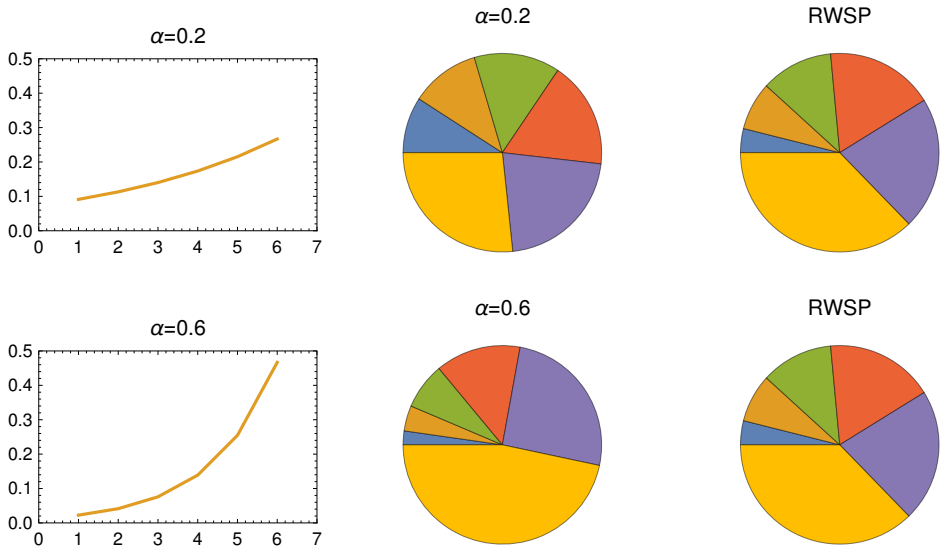


Figure 3.7: Selection probabilities of the individuals - RWSP vs NLRSP with $\alpha = 0.2$ and $\alpha = 0.6$.

It should be mentioned that when there are big number of individuals participating in the NLRSP, even moderate selection pressure will have great impacts to those individuals in the end of the sorted list, i.e. just very few of them in the end will get the most of selection likelihood whereas others get almost nothing. In this regard maybe some smaller values for the selection pressure are more favorable. Lastly, when using Equations of (3.1) and (3.2), there is no need to normalize the values of rankings, since $\sum_{i=1}^{n} R_{\text{LRSP}}(i, n) = \sum_{i=1}^{n} R_{\text{NLRSP}}(i, n) = 1$.

3.4.3 Decision Tables

So far, we have discussed the model modification process, quantitative and qualitative properties in the generation process as well as the selection policies. We have not discussed how a model is modified in a controlled manner. In this regard, we introduce the concepts of Decision Tables and Configuration Files in this section.

Suppose that the base model M is given in which we try to modify it in a controlled manner to get the derived model of M'. A *Decision Table* (DT) can be roughly considered

as a map that tells the Stochastic Controller how to modify M in order to eventuate in M'. This concept can be generalized to the creation of model histories. Let n revisions ($n \geq 2$) be needed in the model history and let $1 \leq i < n$. DT_i is a map that tells the system how to modify the model in revision i in order to obtain its successor in revision $i+1$. Decision tables are highly configurable and are given to the system via the Configuration Files. Each decision table roughly contains the following items:

Item-1 A list of ContextTypes, in which each ContextType has a corresponding probability for being selected, and a selection policy for selecting one of them.

Item-2 For each ContextType, a list of defined edit operations for that type as well as a selection policy for selecting an operation out of the list. Each edit operation is accompanied with (a) its selection probabilities and (b) number of frequencies. The case (a) is only active in the stochastic interpretation mode, and the case (b) is only enabled in the literal interpretation mode (see Section 3.3.4).

Item-3 For each Edit Operation, a selection policy that selects a Context for that operation. The Context should be of type ContextType.

Item-4 Selection policies for selecting parameter values of edit operations.

Decision tables are provided to the system via a configuration file. In order to have a better comprehension of the previously presented concepts, we provide a "conceptual" configuration file in Listing 3.1. The configuration file shows how the materials that are presented up to now are mixed together in order to configure the system. The file is in the XML format but for the sake of clarification, some parts are omitted or simplified[9].

Considering the specification of decision tables, the above configuration file is interpreted as follows: **Item-1** corresponds to $<$ **SPCT** $>$ which is a Selection Policy for ContextTypes followed by $<$ **CT** $>$ tags specifying the ContextTypes themselves.

For each $<$ **CT** $>$ tag, i.e. for each ContextType, there are different edit operations which are defined for that type. Such operations are denoted by $<$ **O** $>$ tags. For selecting an Operation from the set of defined edit operations, a Selection Policy is defined through $<$ **SPO** $>$ tags (**Item-2**). For each Operation, there is a Selection Policy which specifies how Contexts of the previously selected ContextTypes are selected (**Item-3**). The Selection Policy for Contexts are denoted by $<$ **SPC** $>$ tags. Selection Policies of Parameters for each edit operation (**Item-4**) is omitted from the configuration file for simplicity, but principally the same concepts presented earlier are applied.

For each edit operation which is denoted by $<$ **O** $>$, we have the name of that operation as well as $\hat{p}$ that assigns the probability of selection for that operation, this

[9] In this configuration file, the following abbreviations are used:
DT=Decision Table, CT=ContextType, C=Context, O=Operation, SP=Selection Policy, SPCT=Selection Policy for ContextTypes, SPC=Selection Policy for Contexts, SPO=Selection Policy for Operations, RWSP=Roulette Wheel SP, SRSP=Simple Ranking SP, LRSP=Linear Ranking SP, NLRSP=Nonlinear Ranking SP, $\hat{p}$=Selection Probabilities (PMF), $\tilde{f}$=Frequency of an Edit Operation, α=Selection Pressure for LRSP and NLRSP.

Listing 3.1 Sample conceptual configuration file of SMG.

```
<DT >
    <SPCT Name="RWSP" UpdateMode="Fix" />
    <CT Name="UMLPackage" p̂="20">
        <SPO Name="RWSP" UpdateMode="Fix">
        <O Name="createInterface" p̂="1" f̃="2" />
            <SPC Name="NLRSP" α ="0.8" UpdateMode="Dynamic" />
        </O>
        <O Name="createClass" p̂="20" f̃="3" >
            <SPC Name="NLRSP" α ="0.6" UpdateMode="Dynamic" />
        </O>
        ⋮
    </CT>
    <CT Name="UMLClass" p̂="1">
        <SPO Name="RWSP" UpdateMode="Fix" />
        <O Name="createAttribute" p̂="1" f̃="5">
            <SPC Name="LRSP" α ="0.8" UpdateMode="Dynamic" />
        </O>
        <O Name="createMethod" p̂="1" f̃="5">
            <SPC Name="LRSP" α ="0.8" UpdateMode="Dynamic" />
        </O>
        ⋮
    </CT>
    ⋮
</DT>
```

value is only active in the stochastic interpretation mode. There is also a $\tilde{f}$ value which specifies the frequency of that operation which is only enabled in the literal interpretation mode (see Section 3.3.4).

In the above configuration file, we have the possibility to update fitness values of individuals after each successful execution of an edit step (UpdateMode=Dynamic) or not updating them for the whole modification process (UpdateMode=Fix). As discussed earlier in Section 3.4.1, this will affect the qualitative properties of the produced models.

3.5 Evaluation

In this section we provide an evaluation for SMG. The results are obtained by taking an average over five executions. The executions were run on an Apple MacBook Pro with

2.66GHz Intel Core i7 processor with 4GB of RAM.

Performance Evaluation For evaluating the SMG performance, at first the configuration file consisted of only creation operations for generating UML class diagrams with the sizes of 100, 1000, 5000 and 10000 elements. To estimate the modification times of models, we applied all available edit operations with the same probabilities. On each generated model we applied three different number of edit operations, proportional to the total number of model elements in the initially generated models. The total number of applied edit operations were set at 25%, 50% and 75% of the total number of elements available in the model. For instance if the model consists of 1000 elements then 250, 500 and 750 edit operations were applied to it.

Table 3.2 shows the run-time performance of SMG with the above settings. SMG performance is good in practice. The times are given in seconds and they do not include times for loading and serialization.

#Elem. in Base Model (NEBM)		**100**	**1000**	**5000**	**10000**
Creation Time (second)		0.03	0.52	10.77	55.47
Modification Time (second)	**25%**	0.02	0.36	9.43	50.31
#Edit Operations =	**50%**	0.03	0.70	19.95	117.21
xy% of NEBM	**75%**	0.02	1.04	35.76	249.59

Table 3.2: Runtime evaluation of SMG - Creation and modification times.

Qualitative Evaluation In order to evaluate SMG for the quality of generated models, we used the frequencies of model elements available in a real project. In this regard, we used the frequencies of available model elements in a real software system. The specified probabilities of edit operations in Table 3.3 are given based on our observation of the repository of the ASM project[10] (see Section 4.4). Again we created models with sizes of 100, 1000, 5000 and 10000 elements with those specified probabilities.

As shown, the frequencies of the delivered model elements are not quite close to the specified ones for small models. This behavior is intrinsic characteristic of statistics and probabilities and the reason of it is the fact that the specified frequencies are quite skewed. For skewed distributions execution of more edit operations provide better approximations. For non-skewed ones, smaller number of executions yields also good results. In this particular case, for models with the size of 1000 and more, the approximations are satisfactorily close to the specified ones.

[10] The ASM project available at `http://asm.ow2.org/`, is a general purpose framework for analysis and manipulation of Java bytecodes.

Edit Operation	Spec. Freq.	100	1000	5000	10000
createPackage	0.64%	3.33%	0.42%	0.84%	0.50%
createClass	4.20%	5.56%	3.85%	4.27%	4.19%
createInterface	0.60%	0.00%	0.73%	0.49%	0.59%
createAttribute	11.50%	12.22%	12.50%	11.64%	11.25%
createMethod	28.83%	11.11%	25.46%	27.96%	29.54%
createParameter	51.83%	62.22%	55.00%	51.97%	51.68%
createAssociation	2.40%	5.56%	2.08%	2.83%	2.25%

Table 3.3: Qualitative evaluation of SMG - Observed frequencies of the created elements vs the specified ones.

3.6 Summary

In this chapter we focused on generating test models for model differencing, model versioning and history analysis tools. In this regard, we studied the state-of-the-art approaches in the field of test models generation. The approaches can be categorized to direct and indirect approaches. In indirect approaches, the original metamodel should be translated into constraints of a SAT solver (usually Alloy) and the solution is then translated back to an equivalent model. The problem is that such approaches are slow and there is no control over the generation process.

The majority of the existing direct approaches were motivated from the domain of model transformation testing. There, partitioning of a metamodel to subsets which are more appropriate for a testing scenario and combining instances of those partitions in test models, is a common approach. On the contrary, in the domain of model differencing and model versioning which typically deal with problems resulting from the collaborative development paradigm, finding proper differences between models and merging different versions of models are of great importance. Different versions of models result from modifying models by applying edit operations which are supported by common model editors. Typical editors usually have an internal implementation of the abstract syntax graph representation of models and their edit operations are principally the basic graph edit operations. Therefore, a suitable generator should principally mimic the editing process of models using graph edit operations.

Additionally, the generated test models should exhibit realistic properties. Therefore it is essential that the process of applying edit operations to be under control and the generated models have desired properties. In this regard, this chapter contributed to the generation of more realistic test models for model differencing, model versioning and model analysis tools. The proposed generator generates models by controlled application of edit operations. More importantly, it supports the stochastic properties in the generated models. It is also capable of generating more complex test models like model histories where the properties of model elements and their modifications should be under control. Finally, in addition to the basic edit operations, the proposed generator sup-

ports more complex edit operations, which helps it to create more complex structures in the generated models.

Part III

Analysis of Design Models Evolution

CHAPTER 4

Capturing the Evolution of Design Models

In this chapter, we present our approach to capture the evolution of software systems based on their structural changes. The approach is generic in the sense that it can be adapted to other model types than design models. Here, we illustrate the approach by an application to our design-level representation of Java systems, but the process can be readily adapted to the design models of other object-oriented languages. The chapter is organized as follows: first, we motivate the requirement for capturing structural changes in Section 4.1. Next, the structural model differencing framework which is used in our approach is introduced in Section 4.2. We showcase its application to design models of Java systems in Section 4.3. Finally in Section 4.4, we discuss how our sample set of real Java systems was selected. The sample set is used to study the evolution of software systems at the abstraction level of design models. The computed differences between design models and the resulting difference metrics that we measure in this chapter are the basis for our statistical analyses of evolution in Chapters 5 and 6.

4.1 Motivation

The evolution of a model typically leads to a sequence of revisions in which revision r_{n+1} replaces its predecessor revision r_n. Thus, a basic prerequisite to analyze the evolution of a model is to capture the changes between r_n and r_{n+1} in a suitable way. State-of-the-art approaches[1] to understand the evolution of models of software systems are based on software metrics [Lanza and Marinescu, 2006, Fenton and Bieman, 2014] and similar static attributes; the extent of the changes between two revisions of a software system is expressed as differences of metrics values, and further statistical analyses are based on

[1] See Chapter 5 for more information.

these differences. Unfortunately, such approaches do not reflect the dynamic nature of changes well.

The static metric *Number of Methods* (NOM) of classes is an example of this: if 5 existing methods are deleted, 6 new methods are added and 3 methods are moved to another class between two subsequent revisions we will observe an increase of one in this metric, although the actual amount of change is much larger.

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 [Wenzel, 2010]. 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 structural changes that can be observed between subsequent revisions of a system, where each change represents the invocation of an edit operation which is applicable on the given model type.

As we discussed in Section 2.3.1, textual differences consisting of insertions and deletions of lines of source code will not be a basis for computing meaningful difference metrics, because they lack the appropriate level of abstraction. Instead, models should be represented in their natural graph-like way.

4.2 Structural Differencing of Models

4.2.1 Representation and Editing of Models

Structural changes in models can only be specified precisely if a (runtime-) representation of models is available. To that end, we follow the common MDE approach (see Section 2.2) and consider models conceptually as abstract syntax graphs (ASG). Types of nodes and edges of the ASG are specified by a corresponding metamodel (as an example see the one shown in Figure 2.4).

Based on our discussions in Section 2.4, two different levels of abstraction can be considered for computing model changes, each level is based on a specific set of edit operation definitions. The first set of edit operations consists of **low-level** graph modifications, i.e. generic graph operations such as creating/deleting single nodes/edges of an ASG and changing their attribute values. The second set of edit operations contains **high-level** (or user-level) edit operations, including model refactorings, which are applicable to a given model type from a user's point of view.

4.2.2 Differencing of Models

Based on our previous discussions in Sections 2.3 and 2.4, Figure 4.1 shows the structure of a model differencing tool chain which is able to produce differences on both levels of abstraction.

In the initial matching phase, correspondences between elements which are considered to be "the same" in both versions of a model are identified. A low-level difference is then derived from this matching.

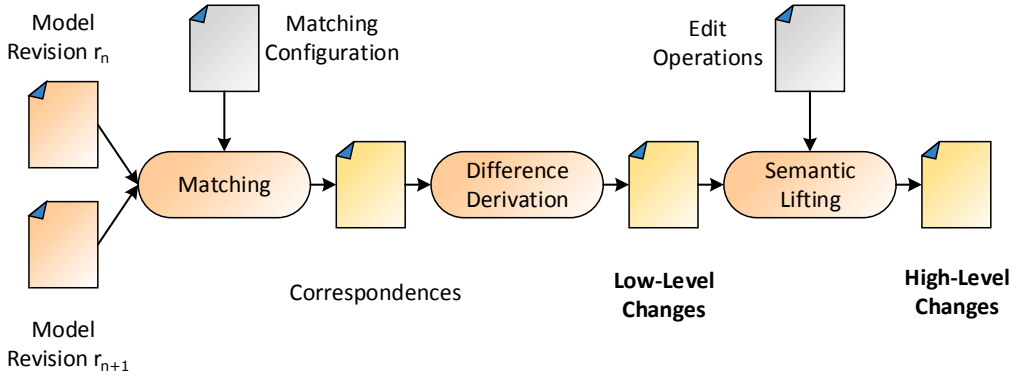


Figure 4.1: Coarse-grained structure of a model differencing pipeline (Adapted from [Kehrer et al., 2013a]).

Since such low-level differences captures the changes between two models on the level of the ASG, they are often hard to comprehend for model developers who are used to thinking on the abstraction level of the conceptual edit operations available in model editors. To that end, low-level differences can be semantically lifted to high-level (or developer-friendly) differences in an additional processing step (see Section 2.4.2). Each high-level change represents the invocation of a conceptual edit operation (referred to as high-level edit operation in this dissertation) which is applicable for a given model type. An important characteristic of high-level edit operations is that they are not generic, but have to be individually engineered for a given model type. Such high-level edit operations oftentimes lead to several low-level changes when applied to a model. In case of complex edit operations such as refactorings, the number of low-level changes induced by the application of such an operation can be very large.

4.2.3 Difference Calculation Using the SiDiff/SiLift Framework

The difference between two versions of a model can generally be computed by any state-of-the-art model comparison tool (see Section 2.3 and see [Kolovos et al., 2009, Stephan and Cordy, 2013, 2012] for surveys on different approaches and tools), as long as the approach can be adapted to the given model type. The SiDiff/SiLift framework [Kehrer et al., 2012b,a] is well-suited for this purpose (see Section 2.4). Every processing step of the difference calculation pipeline shown in Figure 4.1 can be tailored to the specific characteristics of a given modeling language. The matching configuration is written in a domain-specific language which provides various strategies for the identification of corresponding elements. Edit operations are specified using the Henshin language [Arendt et al., 2010], a model transformation language which is based on graph transformation concepts (see Sections 2.1.5 and 2.4.2).

The implementation of the conceptual differencing pipeline of Figure 4.1 within the SiDiff/SiLift framework is based on the Eclipse Modeling Framework (EMF) [Steinberg

et al., 2009]. EMF uses Ecore, which is basically a structural data modeling language, for the definition of meta-models. Ecore supports all concepts being used in common MDE metamodels, e.g. the definition of containment structures, inheritance relations between type definitions etc. It is based on object-oriented principles, thus the implementation of ASGs is straightforward; nodes of an ASG are represented by (runtime-) objects, while edges are represented as references between those objects. An interpreter for the Henshin transformation language that targets EMF models is readily available.

In this work, using the SiDiff engine, we use five different kinds of low-level edit operations to express the low-level changes between two revisions r_n and r_{n+1} (see Figure 4.1):

- **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 attribute of a model element is changed 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.

As mentioned, the high-level changes have to be specifically tailored for each model type. In section 4.3.4 we show how high-level changes are specified on design model representation of Java systems (provided in Section 4.3.2).

4.3 Application to Evolving Java-based Systems

In this work, we reverse-engineered Java software systems and derived "design-level" class diagrams[2] out of them. In order to compute the changes between subsequent revisions, these class diagrams were compared using the model differencing techniques provided by the SiDiff/SiLift framework.

Section 4.3.1 introduces an example of two successive versions of a sample Java program serving as running example throughout this section. An overview of our model-based representation of Java projects is given in Section 4.3.2. Subsequently, we give a brief overview of the configuration of our model differencing pipeline which we used for calculating low-level and high-level changes between revisions of design-level Java models: different types of low-level changes are considered in Section 4.3.3, while Section 4.3.4 summarizes available edit operations for the detection of high-level changes in design models of our Java systems.

[2] Simplified class diagrams are also used in the analysis and definition phase of a project. In this work, we will always refer to the more detailed class diagrams as used in the design phase, which are translated into source code in many MDE methods.

Java code - revision r_n

```
package Geometry;

class Circle{

  // Fields
  int id;
  double radius;

  // Methods
  double circumference(){
    // ...
    // ...
  }

  double area(){
    // ...
    // ...
  }

}
```

Java code - revision r_{n+1}

```
package Geometry;

abstract class Figure{
  int id;
  class Fraction{
    int numerator;
    int denominator;
  }
}

class Circle extends Figure{
  // Methods
  Fraction fRadius;
  Fraction getCircumference(){
    // ...
  }
  Fraction getArea(){
    // ...
  }
}
```

Figure 4.2: Excerpt from a sample Java program - Original version r_n and its revision r_{n+1}.

4.3.1 Example

Figure 4.2 shows two revisions of a sample Java program. Version r_n has been modified to become version r_{n+1} based on the following intentions:

- The field `id` has been extracted to a new superclass `Figure` in order to be reusable by other figure classes.
- Instead of using floating point real number representations, a new data type `Fraction` is introduced. The data type is defined as an inner class of class `Figure`.
- The names of field `radius` as well as methods `circumference` and `area` have been changed due to project-specific style guides and general Java coding conventions.

4.3.2 Representation of Java Projects as Models

An excerpt of our metamodel for design-level Java models is shown in Figure 4.3. The complete metamodel consists of 15 different element types and is available in [Website, 2015].

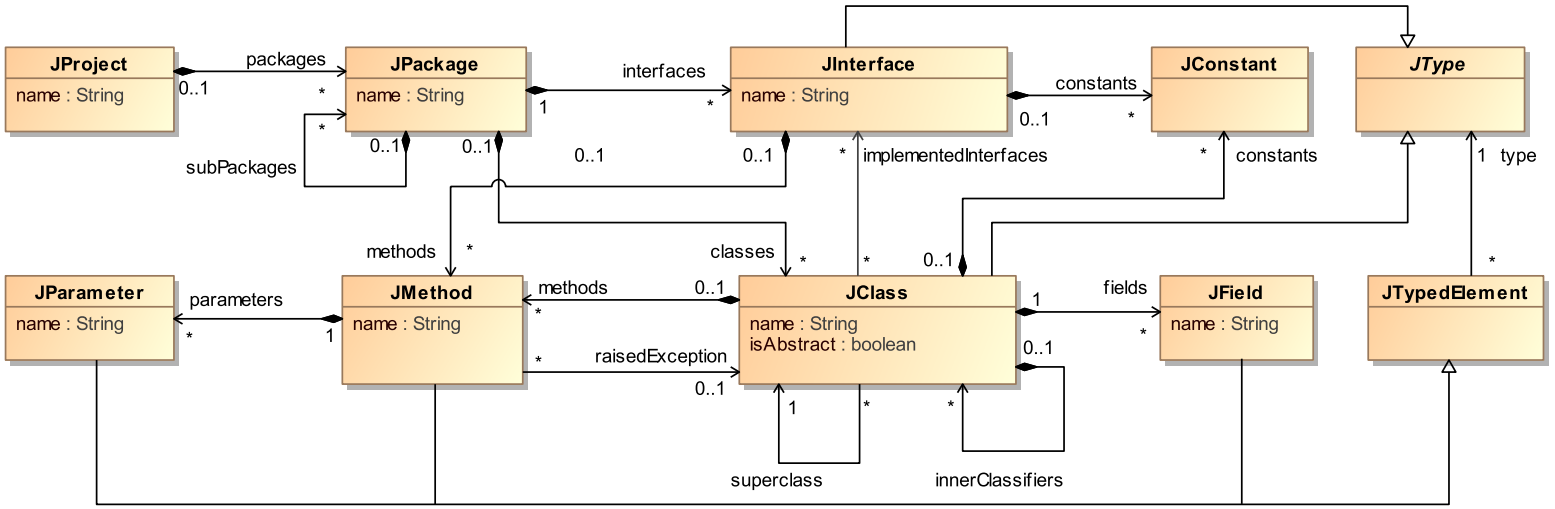


Figure 4.3: Excerpt of our metamodel for class diagrams of Java source code.

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 contain only methods (JMethod) and constants (JConstant), whereas classes can additionally contain attributes (JField). Naturally, methods can have parameters (JParameter).

The seven element types omitted in Figure 4.3 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).

ASG representations of our sample revisions r_n and r_{n+1} of Figure 4.2 are shown in Figure 4.4. Some details, e.g. certain attribute values of ASG objects, are omitted for the sake of readability.

4.3.3 Low-level Changes

As explained in Section 4.2, low-level changes can be derived from a given matching in a generic way. The only component in our differencing pipeline which needs to be adapted to design-level models of Java programs is the matcher being used in the differencing pipeline (see Figure 4.1).

In our case, the reverse-engineered models do not have persistent identifiers; therefore matching approaches based on unique identifiers cannot be used (see Section 2.3.2). Because of this, we used the similarity-based matching algorithm provided by the SiDiff model differencing framework [Kehrer et al., 2012b]. We carefully configured the matching engine such that only "correct" correspondences between our design-level class diagrams were computed.

Low-level Changes in our Example Table 4.1 lists the low-level changes for our running example of Figure 4.2. We assume here that elements having the same name in both versions are matched. In addition, the matching contains correspondences for the renamed field `radius` (named `fRadius` in revision r_{n+1}) as well as for the renamed methods `circumference` and `area` (named `getCircumference` and `getArea` in revision r_{n+1}).

Low-level Difference Metrics Since the SiDiff/SiLift framework (Section 4.2.3) reports 5 different kinds of low-level edit operations, for each of our 15 element types of our metamodel and for each of these 5 kinds of low-level edit operations, we finally define specific difference metrics, namely the number of occurrences of these edit operations on model elements of this type in a low-level difference. Thus, we obtain a total number of 75 (=15×5) low-level difference metrics.

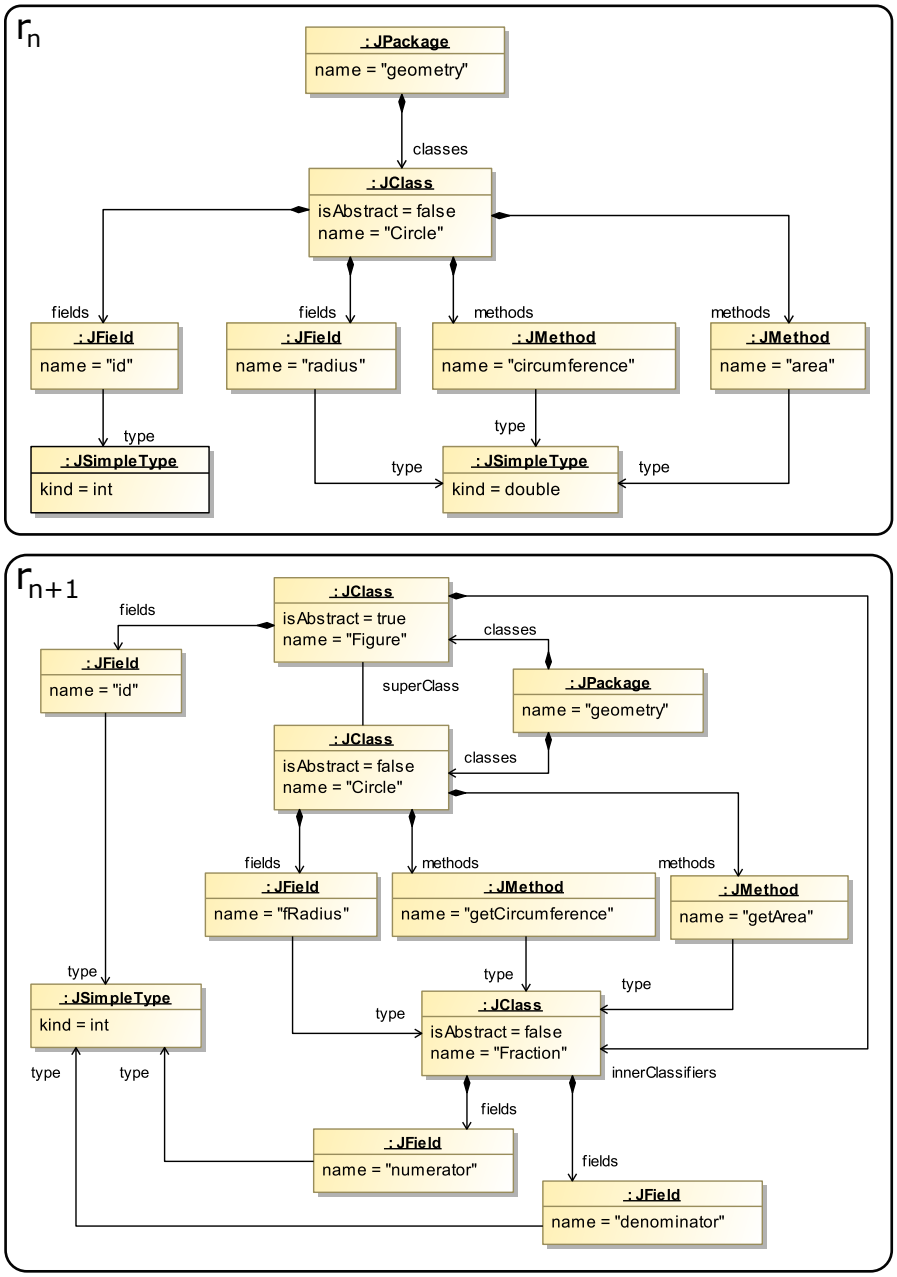


Figure 4.4: ASG representations of our sample revisions r_n and r_{n+1}.

4.3.4 High-Level Changes

As explained in Section 4.2, the set of high-level edit operations to be detected has to be defined individually for each modeling language. Exact specifications for all operations defined for our Java class diagrams can be found at the accompanying website [Website,

Low-level Op. Kind	Elem. Type	Element
Addition	JClass	Figure
Addition	JClass	Fraction
Addition	JField	numerator
Addition	JField	denominator
Reference Change	JClass	Circle
Move	JField	id
Attribute Change	JField	radius
Attribute Change	JMethod	circumference
Attribute Change	JMethod	area
Reference Change	JField	radius
Reference Change	JMethod	circumference
Reference Change	JMethod	area

Table 4.1: Low-level changes in our running example.

2015]. In sum, we specified a total number of 188 high-level edit operations in design models of Java systems. Table 4.2 provides a summary of the specified high-level edit operations. As shown in the table, the 188 high-level edit operations can be roughly classified into six kinds of edit edit operations which will be explained in the remainder of this section.

Each edit operation has been implemented in a Henshin transformation rule [Arendt et al., 2010]. We illustrate a set of selected edit operations in terms of the Henshin visual syntax, which is very intuitive. As introduced earlier in Section 2.4.2, a rule is defined on the ASG and specifies model patterns which have to be found and preserved, to be deleted or to be created, using the "stereotypes" *preserve*, *delete* and *create*, respectively. In addition, one can specify model patterns which are forbidden and which prevent a rule from being applied (stereotype *forbid*). The examples show that a Henshin rule can define formal parameters. Thus, the context in which a rule is applied can be determined by actual parameters which are passed to a rule.

Kinds of High-Level Edit Operations	Specified Num.
Create	22
Delete	22
Move	12
Set/Unset	85
Modify Non-Containment References	35
Refactorings	12

Table 4.2: High-level edit operations - summary.

Create and Delete Operations We identified 22 **create** operations, the operations to create a new class shown in Figure 4.5 are examples of this. The number of *create* operations is slightly higher than the number of the low-level *addition* counterparts because we clearly distinguish the context in which an element is created. As illustrated in Figure 4.5, we have two operations creating a class in different contexts; the left one creates a nested class in an already existing class, the operation on the right creates a class in a package.

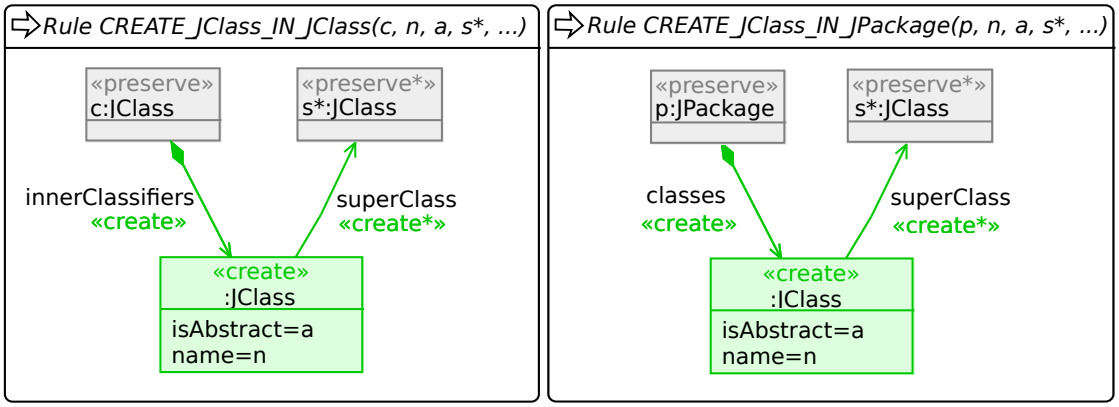


Figure 4.5: Creation of a class in different contexts.

The example shows that high-level edit operations can have several parameters, only a subset of them is shown in Figure 4.5. For example, the context in which a class is to be created is passed to the *create* operations shown in Figure 4.5; the first operation creates a new class in a given class c, while the second one creates a new class in a package p. In addition, the name n of the new class and a boolean value a assigned to attribute isAbstract are passed to both *create* operations. If desired, an existing class can be assigned as superclass to the new class which is to be created, indicated by the optional multi-object s* in both rules.

For each of the 22 create operations, we specified an inverse **delete** operation deleting an object of a particular type in a specific context.

Move Operations **Move** operations are similar to our low-level move operations, i.e. they shift a selected object to a different parent. Similar to our create and delete operations, the old and the new parent context is explicitly specified. The operation shown in Figure 4.6, for instance, moves a field from one class to another.

Note that the number of high-level move operations is slightly lower than the total number of low-level move operations because some movements, e.g. moving parameters between methods, are possible, but not meaningful from a user's point of view. Such move operations are thus excluded from the overall set of high-level operations.

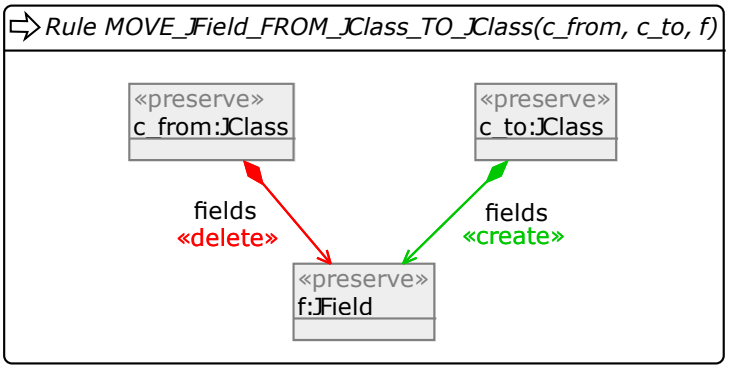


Figure 4.6: Move operation shifting a field from one class to another.

Set/Unset Operations The 15 low-level attribute changes expand to 85 concrete **set** operations which provide dedicated access to each local attribute for all types of elements. While our low-level change operations only report that an attribute of a certain element has been changed, a high-level set operation explicitly states which attribute is to be changed.

If there is at most one outgoing reference of a particular type for a particular ASG object, then references of this type can be handled similar to attributes. An example is shown in Figure 4.7. The operation modifies a method such that an exception that might be thrown by this method is being declared. The creation of such a reference is conceptually treated as *set* operation. The negative application condition (NAC) of «`forbid`» in Figure 4.7 prevents the operation from being successfully executed if method m already has an outgoing reference of type raisedException (we assume here that a method can declare only one type raised exception). An inverse operation to a **set** operation deletes a reference of this type and is considered as **unset** operation. Obviously, the NAC can be omitted for **unset** operations.

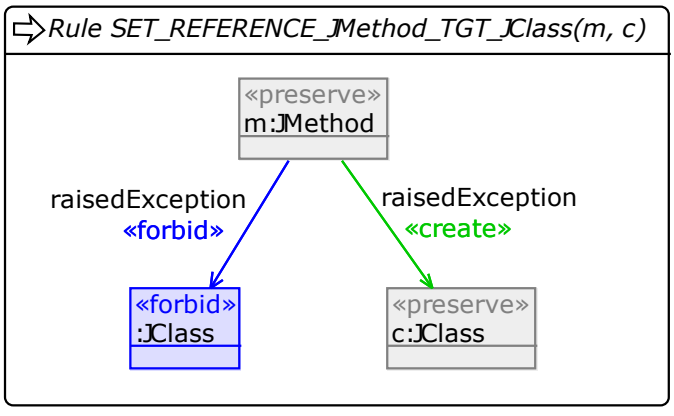


Figure 4.7: *Set* operation adding a "neighbour" to an existing object.

Operations modifying Non-Containment References Finally, the 15 low-level reference change operations unfold to 35 operations which are denoted as **modifying non-containment references**. Examples include the addition (removal) of an interface to (from) the set of interfaces being implemented by a particular class, as illustrated in Figure 4.8. We refer to the respective edit operations as **add** and **remove** operations since we conceptually add (remove) an object to (from) a collection of objects.

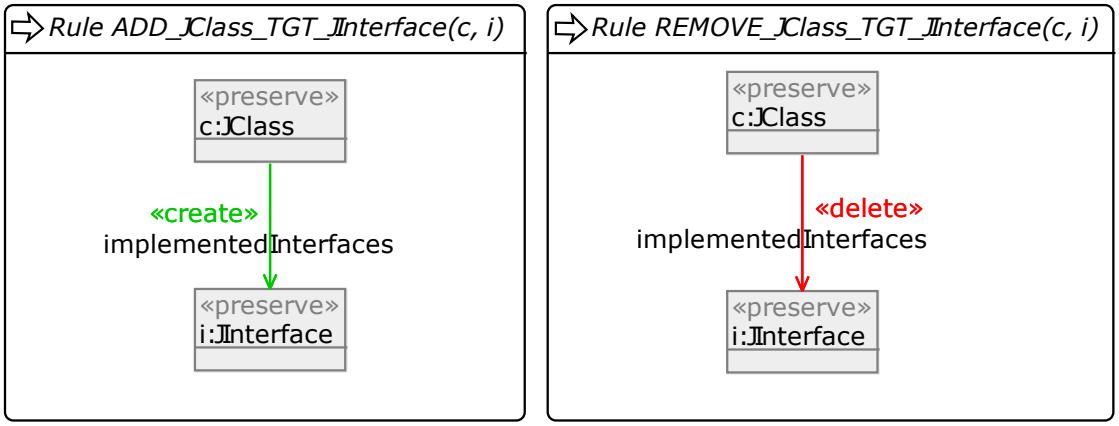


Figure 4.8: *Add/remove* operations modifying non-containment references.

Further examples of operations modifying non-containment references are **change** operations such as the example shown in Figure 4.9: This operation changes the current type of a given field `f` to a new type `t_new`.

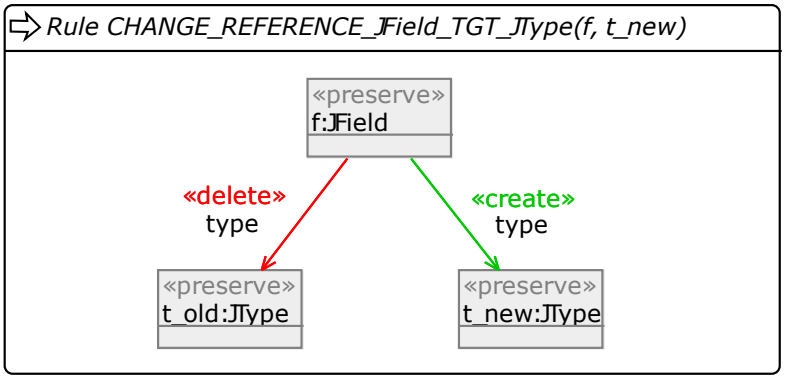


Figure 4.9: *Change* operation modifying a non-containment reference.

Refactorings Additionally, we chose the well-known catalog of object-oriented **refactoring operations** [Fowler et al., 1999] and selected 12 refactorings which are adaptable to design-level class diagrams. For some of these refactorings, the actual size of the sets

of observed changes in a difference can even vary between occurrences of the same edit operation. The refactoring "pullUpField" and "pullUpMethod" are two examples; both edit operations are applied to all common fields/methods in the set of all direct subclasses of a given class.

High-level Changes for our Example Figure 4.10 lists the high-level changes which occur in our running example of Figure 4.2, assuming the same correspondences as in Section 4.3.3. Note that some arguments in high-level edit operation invocations are omitted due to space limitations.

```
createSuperclass(geometry, Circle, "Figure", true, ...)
CREATE_JClass_IN_JClass(Figure, "Fraction", false, ...)
CREATE_JField_IN_JClass(Fraction, "numerator", int, ...)
CREATE_JField_IN_JClass(Fraction, "denominator", int, ...)
pullUpField(id)
SET_ATTRIBUTE_JField_name(radius, "fRadius")
SET_ATTRIBUTE_JMethod_name(circumference,"getCircumference")
SET_ATTRIBUTE_JMethod_name(area, "getArea")
CHANGE_REF._JField_TGT_JType(radius, Fraction)
CHANGE_REF._JMethod_TGT_JType(circumference, Fraction)
CHANGE_REF._JMethod_TGT_JType(area, Fraction)
```

Figure 4.10: High-level changes for our running example.

High-level Difference Metrics Quantitative measurements of high-level changes can be easily obtained by counting the occurrences of the high-level edit operations on the computed differences between two model revisions.

4.4 Selection of Sample Projects and the Data Sets

In the previous sections we showed how the evolution of software systems can be captured at the abstraction level of design models. We discussed two sets of changes that can be detected between revisions of software models. One is the set of low-level edit operations which are basic graph edit operations defined on design models, and the second set contains high-level edit operations which are usually implemented by many low-level edit operations. In this section we discuss how our sample software systems were selected for the measurement of low-level and high-level changes and what are our data sets for our analyses in Chapters 5 and 6.

Obviously, a quantitative study of the evolution must be based on typical, representative projects. We selected our sample projects according to the following main criteria:

C1: The projects must be written in Java programming language, be actively used by end users and be open-source.

C2: The projects must have been actively developed over a long period of time in order to let us study their evolutions appropriately.

C3: All projects must be non-trivial systems which contain at least 100 model elements, preferably more. This requirement excludes trivial small projects.

C4: The selected projects must be typical Java software systems from different application domains.

We found out that the projects reported in the Helix Software Evolution Data Set (HDS) [Vasa et al., 2010] fulfill these four requirements. We randomly selected nine projects from the HDS. Table 4.3 shows the basic information about our sample projects. All revisions of each project were checked out from the respective repositories and reverse engineered into design models. For each project, using the methodology described in Section 4.3, we computed the low-level and the high-level changes between all of its subsequent revisions.

Since the version control systems (VCSs) used by our sample projects are source code centric and are not exclusively used for models, there are cases where the changes in the source code does not influence the reverse engineered design models, which are at a higher abstraction level. There are also cases where other parts of the systems other than source code are modified and committed to the VCSs. Such changes are not reflected in design models, either. All resulting model versions which turned out to be identical to their predecessor are disregarded in our analysis. We considered only differences where there is at least one change between the compared design models. Using this approach, 3809 design models were considered.

Finally, the measured 75 low-level and 188 high-level difference metrics between revisions of models are our data sets for our statistical and time series analyses in the coming chapters.

4.5 Summary

In this chapter we learned that the evolution of software systems at the abstraction level of design models cannot be properly captured by measuring the changes between static software metrics. Therefore, we employed difference metrics which are more appropriate in this regard.

Measuring the evolution of design models in terms of difference metrics requires that the differences between models are properly computed. In this regard we used the SiDiff/SiLift model differencing pipeline and we measured the differences of models at two abstraction levels. The first level is reported in terms of the applied low-level edit operations defined on design models. Such operations are basic graph edit operations which are difficult to work with, by developers who are used to working on model editors

Name	Description	#Rev. in VCS	#Model Rev.	Min #Elem.	Mean #Elem.	Max #Elem.
ASM	Byte code manipulation framework	1448	260	781	3635	5264
CheckStyle	Code formatting tool	2586	1011	388	3181	5791
DataVision	Reporting tool	153	29	5431	5520	5640
FreeMarker	Template engine library	1193	340	5825	6895	7729
HSQLDB	SQL relational database engine	3810	961	17520	22873	26591
Jameleon	Automated testing framework	1772	285	2276	3139	3954
JFreeChart	Library for creating charts	2270	366	18302	21583	24614
Maven	Project management tool	9343	781	1150	3786	5910
Struts	Web application framework for Java EE	4242	737	757	4323	6383

Table 4.3: Selected Projects.

that support more user-friendly operations. Therefore, the set of low-level changes were semantically lifted to high-level operations. High-level operations typically comprise of many low-level operations.

The way low-level and high-level changes are computed was discussed in detail by providing a working example in Java. In Short, we measured 75 low-level and 188 high-level difference metrics on design models of real Java systems. To this aim, we selected 9 typical Java systems and we used them as the sample set of our analyses in this dissertation. We checked out source code of the systems from their respective repositories and we reversed engineered the code of subsequent revisions into design model representations. The evolution of design models of Java systems was measured by low-level and high-level metrics on our sample set. The measured metrics were used in the coming chapters as the basis for analyzing the evolution of Java systems at the abstraction level of design models.

CHAPTER 5

Statistical Analysis and Simulation of Design Models Changes

As we discussed in Chapter 1, one of our goals in this dissertation was to statistically study the evolution of design models. The evolution of design models was captured through analysis of changes between revision models of our sample software systems. This has led to sets of evolution measurements which were captured in two levels of low-level and high-level changes (see Chapter 4). Such an analysis of the changes not only allows us to better understand the evolution of software systems at the abstraction level of design models, but also let us generate more realistic test models for model differencing, mode versioning and model processing tools (see Chapter 3).

In this regard, in this chapter we focus on statistically studying the evolution of software systems by analyzing the evolution of both low-level and high-level changes. In Section 5.1 we introduce mathematical requirements and statistical models, i.e. distributions, which are used to study the evolution. The results of the analysis for low-level and high-level changes are separately and in detail presented in Section 5.2. The analysis of changes shows that the proposed statistical distribution in Section 5.1 are quite suitable to mathematically model both of low-level and high-level changes. Therefore, such statistical models can be used to replicate the real observed changes and create more realistic test models. In order to use these statistical distributions in our model generator, we studied how random variates of these distributions can be generated in Section 5.3. This chapter ends with a comparison with the related works in Section 5.4 and a chapter summary in Section 5.5.

5.1 Statistical Models for Describing Changes

The previous chapter described how the sample projects were selected and how our data sets of low-level and high-level changes have been computed. Our goal is 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.

As an example, consider Figure 5.1 which depicts the histogram of the frequencies of additions of methods for the HSQLDB project. The y-axis of the histogram is on the logarithmic scale for better comprehension. Most differences between HSQLDB revisions contain only a small number of additions of methods. However, there are also few differences which contain a large number of additions of methods. To be more specific, the history of the HSQLDB project has two revisions with 864 and 1094 additions of methods.

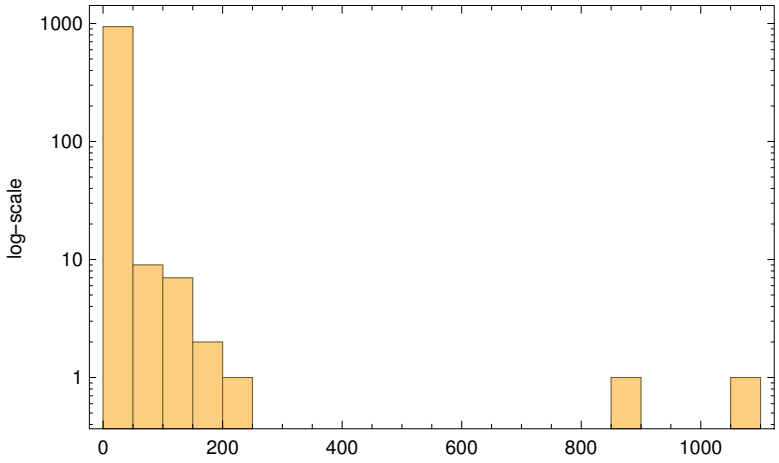


Figure 5.1: Histogram of the frequencies of addition of methods for HSQLDB.

Generally, for distributions with heavy tails, the probability ratio between an outcome with a small probability and an outcome with a big probability is very large. This is not happening for distributions without heavy tails, e.g. the normal distribution [Bilar, 2008b,a].

Many continuous and discrete univariate distributions are known [Johnson et al., 1994, 2005, Wimmer and Altmann, 1999, Krishnamoorthy, 2006, Walck, 2007, Forbes et al., 2011]. We tested the adequacy of 60 distributions[1] which could be promising[2]. For the selection of these distribution, we considered the shapes of the histograms of

[1] See [Website, 2015], for the full list of the tested distributions.

[2] Some of these distributions are famous and were also quite successful in other fields of research, e.g. binomial, Poisson, gamma and chi-squared distributions.

the data as well as the shapes of the probability distribution functions of the candidate distributions. The main feature of the data is that heavy tail histograms as well as the histograms with short tails are both observable. From our candidate distributions, only six discrete distributions which also support heavy tails performed acceptable, although with different levels of success (see Section 5.2). These six are the discrete Pareto distribution of the power law family, the beta binomial distribution, the generalized Poisson distribution and finally the Yule, the Waring and the beta-negative binomial distributions from the family of the hypergeometric distributions [Johnson et al., 1992].

It should be noted that in the related literature, there are some similar concepts which are used in other research works and usually the terms and definitions of the employed methods and distributions have been used ambiguously making them difficult to differentiate. In order to make our proposed results reproducible for other researchers and avoid the ambiguity, we tried to summarize the mathematical infrastructure needed to replicate the results presented in this chapter.

In Section 5.1.1 we briefly provide the mathematical requirements which are frequently used throughout the chapter. Section 5.1.2 provides more information about the discrete Pareto distribution. Section 5.1.3 introduces the beta binomial distribution while Section 5.1.4 gives the formal definition of the Yule, the Waring and the beta-negative binomial distributions as well as their relationships to some other distributions that we use later in Section 5.3 for generating their random variates. Finally, in Section 5.1.5 the generalized Poisson distribution is presented.

5.1.1 Mathematical Requirements

In this section we briefly introduce some mathematical functions and definitions which we use frequently in the rest of the chapter. We do not go further into details but the interested reader might refer to [Erdelyi et al., 1955, Gradshteyn and Ryzhik, 2007, Olver et al., 2010, Wimmer and Altmann, 1999, Johnson et al., 2005] for more information.

Required Mathematical Functions:

The *gamma function* is defined as:

$$\Gamma(a) = \int_0^\infty t^{a-1} e^{-t} dt, \quad \mathrm{Re}(a) > 0 \,.$$

in which $\mathrm{Re}(a)$ indicates the real part of a. When a is a positive integer we have: $\Gamma(a) = (a-1)!$.

The *beta function* is defined by:

$$\begin{aligned} \mathrm{B}(a,b) &= \int_0^1 t^{a-1}(1-t)^{b-1} dt \\ &= \frac{\Gamma(a)\,\Gamma(b)}{\Gamma(a+b)}, \quad \mathrm{Re}(a) > 0,\ \mathrm{Re}(b) > 0 \,. \end{aligned}$$

The *ascending factorial*[3] is denoted by $x^{(n)}$ and is defined as[4]:

$$\begin{aligned} x^{(n)} &= x\,(x+1)\,(x+2)\cdots(x+n-1) \\ &= \frac{\Gamma\,(x+n)}{\Gamma\,(x)},\ x \in \mathbb{R},\ n \in \mathbb{N} \text{ and } x^{(0)} = 1\ . \end{aligned}$$

where $\mathbb{R}$ and $\mathbb{N}$ indicate the sets of real and natural numbers respectively.

The *Riemann zeta function*[5] $\zeta(s)$ is defined by:

$$\zeta\,(s) = \sum_{j=1}^{\infty} \frac{1}{j^s}, \quad \mathrm{Re}\,(s) > 1\ . \tag{5.1}$$

and its derivative, i.e. $\zeta^{'}(s)$ is given by:

$$\zeta^{'}\,(s) = -\sum_{j=1}^{\infty} \frac{\ln\,(j)}{j^s} = -\sum_{j=2}^{\infty} \frac{\ln\,(j)}{j^s}, \quad \mathrm{Re}\,(s) > 1\ . \tag{5.2}$$

Fundamental notions of probability distributions:

Suppose that there is a finite population and consider just two outcomes: *success* with the probability of p and *failure* with the probability of $1-p$. A *Bernoulli trial* is a random experiment in which trials are independently done with two possible outcomes of success and failure. The distribution of a Bernoulli trial is denoted by $\mathrm{Bernoulli}\,(p)$.

In a Bernoulli trial, the probability of k successes in n draws without replacement is classically called the *hypergeometric distribution*. When the same is done with replacement it is called the *binomial distribution* and is denoted by $\mathrm{Bino}\,(n,p)$. In the first case the draws are dependent, while in the latter case they are independent. In a Bernoulli trial, the distribution of the number of failures before n successes happen is called the *negative binomial distribution*. This distribution is denoted by $\mathrm{NegBino}\,(n,p)$. For more information about the binomial and the negative distributions please respectively see Equation (5.14) in Section 5.3.4 and Equation (5.12) in Section 5.3.3.2.

Throughout this chapter we use "PDF" and "CDF" as the abbreviation for the *Probability Density Function* and the *Cumulative Distribution Function* respectively.

5.1.2 Discrete Pareto Distribution and Power Law

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

[3] Similarly the *descending factorial* is also defined.

[4] There is also another symbol for the ascending factorial that is used in the related literature. It is called the *Pochhammer symbol* and is usually denoted by $(x)_n$. It is also used as descending factorial in some cases, so caution is advised.

[5] The Riemann zeta function is a special case of the *Hurwitz zeta function* and the latter is itself a special case of the *Lerch transcendent function*.

observed in linguistics, biology, geography, economics, finance, physics and also computer science, e.g. the size of computer files, grid complex networks, the Internet and web pages hit rates (see [Newman, 2005, Mitzenmacher, 2004, Ilijašic and Saitta, 2010, Adamic and Huberman, 2002, Gabaix, 2009]). Formally we have:

Definition 5.1 (Power Law)**.** A function $y = f(x)$ is said to obey power law to x when $f(x)$ is proportional to some power of x, i.e. $y = c\,x^{\alpha}$ in which α is called the power law exponent or scaling parameter and c is a constant.

The discrete Pareto distribution[6] that is used throughout this chapter is of the power law family and is based on the Riemann zeta function introduced earlier [Erdelyi et al., 1955, Gradshteyn and Ryzhik, 2007, Olver et al., 2010]. It takes a real value $\rho > 0$ as the shape parameter.

Definition 5.2 (Discrete Pareto Distribution)**.** The PDF of the discrete Pareto distribution with shape parameter of ρ is given by [Johnson et al., 2005]:

$$\mathrm{P}[X = x] = \frac{x^{-s}}{\zeta(s)} = \frac{x^{-(\rho+1)}}{\zeta(\rho+1)}, \qquad x = 1, 2, \cdots . \tag{5.3}$$

where $\rho = s - 1 > 0$ and $\rho \in \mathbb{R}$.

For demonstration, Figures 5.2 and 5.3 provide the PDF and the CDF plots of the discrete Pareto distribution for different values of ρ.

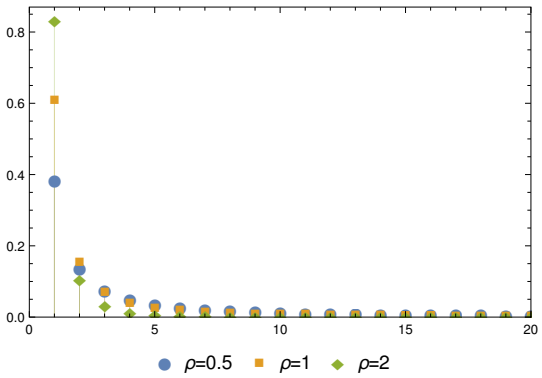


Figure 5.2: PDF-plot of the discrete Pareto distribution for different ρ.

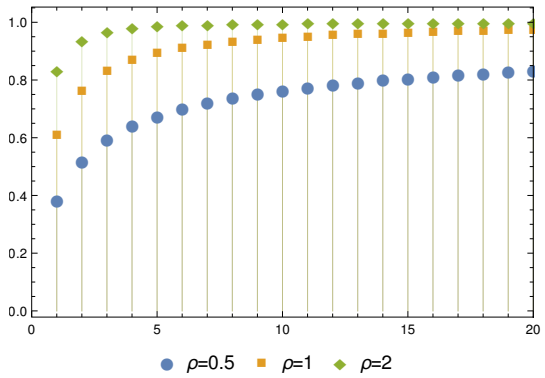


Figure 5.3: CDF-plot of the discrete Pareto distribution for different ρ.

To estimate the parameter of ρ in (5.3), suppose that N dependent random variates x_i of the discrete Pareto distribution are given. Using (5.1) and (5.2), $\hat{\rho}$, the estimator of

[6] Sometimes is also referred to as the Zipf distribution, the Riemann zeta distribution or the zeta distribution.

ρ, which is obtained by the maximum likelihood estimation method satisfies the following relation:

$$\frac{1}{N}\sum_{i=1}^{N}\ln\left(x_i\right)=\frac{-\zeta^{'}\left(\hat{\rho}+1\right)}{\zeta\left(\hat{\rho}+1\right)} \tag{5.4}$$

[Johnson et al., 2005] provides more information for the solution of (5.4) as well as an alternative poorer estimation for the parameter ρ in (5.3). Moreover, the discrete Pareto distribution can also be obtained from the general Lerch distribution[7] [Wimmer and Altmann, 1999]. [Zörnig and Altmann, 1995] have provided moments of the general Lerch distribution as well as estimation of its parameters.

5.1.3 Beta Binomial Distribution

As we saw in Section 5.1.1, the binomial distribution is the distribution of successes in n draws of a Bernoulli trial with success probability of p. When the probability of the Bernoulli trials, i.e. p, is not fixed and is changing according to the beta distribution[8] of $\text{Beta}\left(\alpha,\beta\right)$, we have a beta binomial distribution.

Definition 5.3 (Beta Binomial Distribution)**.** The beta binomial distribution with shape parameters of α, β and n is denoted by $\text{BetaBino}\left(\alpha,\beta,n\right)$ and its PDF is given by [Wimmer and Altmann, 1999]:

$$\begin{aligned}\text{P}\left[X=x\right]&=\binom{n}{x}\frac{\text{B}\left(\alpha+x,n+\beta-x\right)}{\text{B}\left(\alpha,\beta\right)}\\&=\frac{\Gamma\left(n+1\right)\Gamma\left(\alpha+\beta\right)\Gamma\left(\alpha+x\right)\Gamma\left(\beta+n-x\right)}{\Gamma\left(\alpha\right)\Gamma\left(\beta\right)\Gamma\left(x+1\right)\Gamma\left(n-x+1\right)\Gamma\left(\alpha+\beta+n\right)},\\x\in\left\{0,1,\ldots,n\right\},\ \alpha>0,\ \beta>0&\text{ and }\alpha,\beta\in\mathbb{R}.\end{aligned} \tag{5.5}$$

To better comprehend the distribution shape, Figures 5.4 and 5.5 shows the PDF of the beta binomial distribution for different values of the shape parameters α and β.

5.1.4 Yule, Waring and Beta-Negative Binomial Distributions

Before going into details, we briefly introduce the Yule, Waring and beta-negative binomial distributions, which are all discrete distributions. The Yule distribution, which has applications in the taxonomy of species in biology, has just one parameter b, which is a positive real. The Waring distribution, which yields the Yule distribution as a special case, has two real parameters, $b>0$ and $n>0$. Both of the Yule and the Waring distributions have been generalized ([Irwin, 1975]) to a hypergeometric distribution called the generalized Waring or the beta-negative binomial distribution[9] [Johnson et al., 2005,

[7] The general Lerch distribution is defined based on the Lerch transcendent function. The Lerch transcendent function gives the Riemann zeta function as its special case.

[8] Beta distribution is introduced by Equation (5.11) in Section 5.3.3.1.

[9] When its support is shifted, it is referred to as the beta-Pascal distribution.

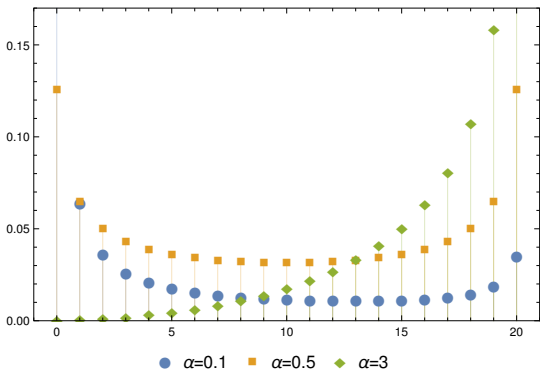


Figure 5.4: PDF-plot of the beta binomial distribution $\text{BetaBino}(\alpha, 0.5, 20)$ for different α.

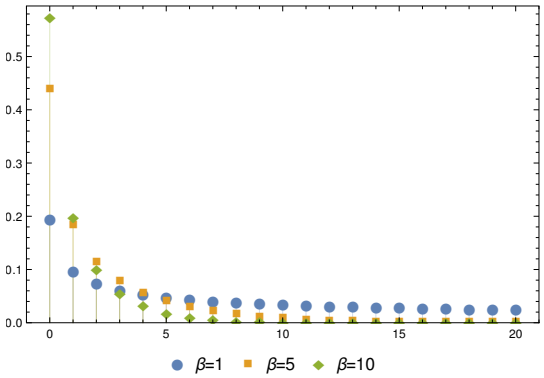


Figure 5.5: PDF-plot of the beta binomial distribution $\text{BetaBino}(0.5, \beta, 20)$ for different β.

Wimmer and Altmann, 1999]. The distribution has three parameters, α, β and n which are positive reals.

Definition 5.4 (Yule Distribution)**.** The Yule distribution with shape parameter of b (denoted by $\text{Yule}(b)$) is specified by the following PDF [Wimmer and Altmann, 1999]:

$$\begin{aligned} \text{P}[X = x] &= b\ \text{B}(b+1, x+1) \\ &= \frac{b\ x!}{(b+1)^{(x+1)}}\ ,^{10} \\ x &= 0, 1, 2, \cdots,\ b > 0 \text{ and } b \in \mathbb{R}. \end{aligned} \tag{5.6}$$

Definition 5.5 (Waring Distribution)**.** The PDF of the Waring distribution with shape parameters of b and n (denoted by $\text{Waring}(b, n)$) is given by [Wimmer and Altmann, 1999, Johnson et al., 2005]:

$$\begin{aligned} \text{P}[X = x] &= \frac{\text{B}(n+x, b+1)}{\text{B}(n, b)} \\ &= \frac{b\ n^{(x)}}{(b+n)^{(x+1)}}\ , \\ x &= 0, 1, 2, \cdots, b > 0,\ n \geq 0 \text{ and } b, n \in \mathbb{R}. \end{aligned} \tag{5.7}$$

According to (5.6) and (5.7), it is easy to see that the Yule distribution is a special case of the Waring distribution when we set $n = 1$. Figures 5.6 and 5.7 depict the PDF plot of the Waring distribution with different shape parameters of b and n.

[10] The denominator is expressed in ascending factorial notation (see Section 5.1.1).

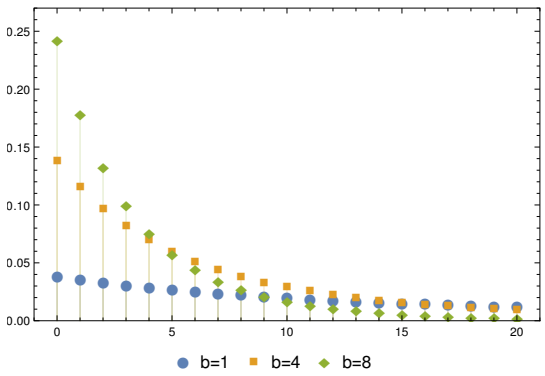


Figure 5.6: PDF-plot of the Waring distribution Waring $(b, 25)$ for different b.

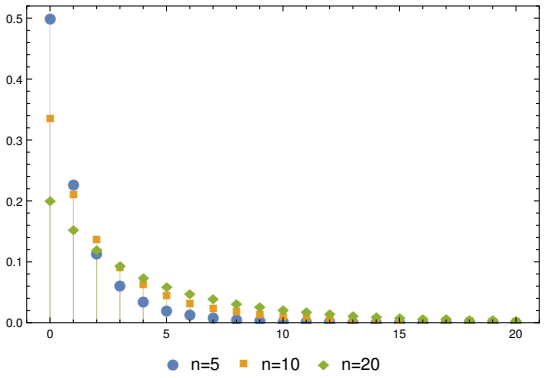


Figure 5.7: PDF-plot of the Waring distribution Waring $(5, n)$ for different n.

As we mentioned earlier, the beta-negative binomial distribution is an extension to the Yule and the Waring distributions. This distribution can also be obtained from the negative binomial distribution NegBino (n, p) (see Section 5.1.1 and Equation (5.12) in Section 5.3.3.2) when the probability of success in the Bernoulli trials, p, changes between different experiments and is distributed according to the continuous univariate beta distribution, i.e. $p \sim \text{Beta}(\alpha, \beta)$ [Wimmer and Altmann, 1999, Johnson et al., 2005]. The beta distribution is defined on the interval of $[0, 1]$ with two shape parameters of α and β. For more information about the beta distribution please see Section 5.3.3.1 (Equation (5.11)).

Definition 5.6 (Beta-Negative Binomial Distribution)**.** The generalized Waring or the beta-negative binomial distribution with shape parameters of α, β and n, is denoted by BNB (α, β, n) and its PDF is specified by [Wimmer and Altmann, 1999, Johnson et al., 2005]):

$$\begin{aligned} \mathrm{P}[X = x] &= \frac{\alpha^{(n)}\beta^{(x)}n^{(x)}}{x!\,(\alpha+\beta)^{(n)}\,(\alpha+\beta+n)^{(x)}} \\ &= \frac{\Gamma(\alpha+n)\,\Gamma(\beta+x)\,\Gamma(\alpha+\beta)\,\Gamma(n+x)}{\Gamma(\alpha)\,\Gamma(\beta)\,\Gamma(n)\,\Gamma(x+1)\,\Gamma(\alpha+\beta+n+x)}\,, \\ x = 0,1,2,\cdots,\ \alpha > 0,\ \beta > 0,\ n > 0 &\text{ and } \alpha, \beta, n \in \mathbb{R}\,. \end{aligned} \tag{5.8}$$

In Equation (5.8), by setting $n = 1$ and changing β by n and α by b, i.e. BNB $(b, n, 1)$, we obtain the Waring distribution. Similarly by setting $n = 1$, $\beta = 1$ and changing α by b, i.e. BNB $(b, 1, 1)$, we get the Yule distribution ([Wimmer and Altmann, 1999]). [Irwin, 1975] has deeply studied the generalized Waring distribution including its limiting cases, special cases, moments etc. Figures 5.8 and 5.9 show the PDF plots of the beta-negative binomial distribution for different values of α and β.

For more information about the moments of the beta-negative binomial, the Waring and the Yule distributions, please refer to [Wimmer and Altmann, 1999, Johnson et al., 2005].

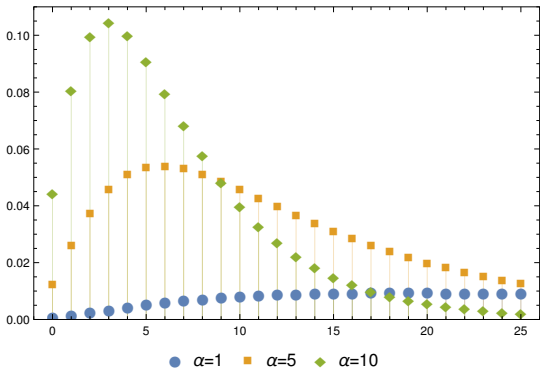


Figure 5.8: PDF-plot of the beta negative binomial distribution $\mathrm{BNB}(\alpha, 3, 20)$ for different α.

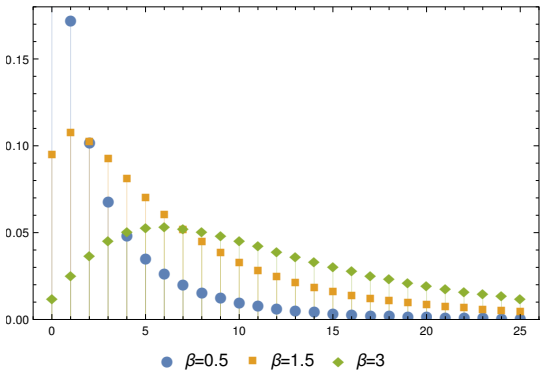


Figure 5.9: PDF-plot of the beta negative binomial distribution $\mathrm{BNB}(5, \beta, 20)$ for different β.

5.1.5 Generalized Poisson Distributions

The Poisson distribution plays an important role in statistics and in modeling of random systems [Cogdell, 2004]. We assume, actually with some simplification, that there are some events happening independently at a constant rate of μ in time. The probability of observing exactly x events in a given period of time obeys the Poisson distribution [Walck, 2007]. The probability density function of the Poisson distribution with shape parameter μ (denoted by $\mathrm{Pois}(\mu)$) is defined as follows [Forbes et al., 2011, Wimmer and Altmann, 1999]:

$$\begin{gathered} \mathrm{P}[X = x] = \frac{\mu^x e^{-\mu}}{x!}, \\ x = 0, 1, 2, \cdots, \mu > 0 \text{ and } \mu \in \mathbb{R}. \end{gathered} \tag{5.9}$$

[Consul and Jain, 1973] has generalized the Poisson distribution which has an extra shape parameter. The distribution is reported to have a very close fit to the Poisson, binomial and negative binomial distributions. It is also reported that it covers heavy tails distributions.

Definition 5.7 (Generalized Poisson Distribution)**.** The generalized Poisson distribution with shape parameters of μ and λ is denoted by $\mathrm{GenPois}(\mu, \lambda)$ and its PDF is given by [Consul and Jain, 1973, Devroye, 1989]:

$$\begin{gathered} \mathrm{P}[X = x] = \frac{\mu(\lambda x + \mu)^{x-1} e^{-(\lambda x + \mu)}}{x!}, \\ x = 0, 1, 2, \cdots, \mu > 0,\ 0 \leq \lambda \leq 1 \text{ and } \mu, \lambda \in \mathbb{R}. \end{gathered} \tag{5.10}$$

In the case that $\lambda = 0$, i.e. GenPois $(\mu, 0)$, we get the Poisson distribution Pois (μ). To have an idea how the generalized Poisson distribution looks like, Figures 5.10 and 5.11 show the PDF of the distribution for different values of the shape parameters μ and λ.

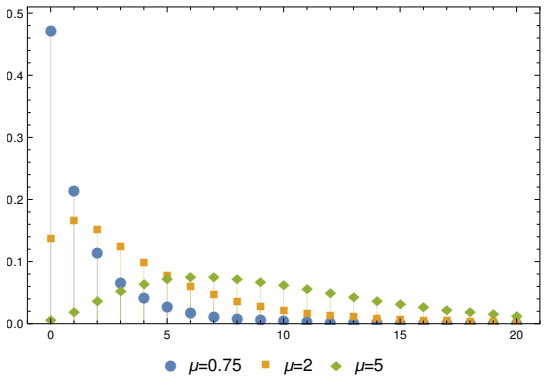


Figure 5.10: PDF-plot of the generalized Poisson distribution GenPois $(\mu, 0.5)$ for different μ.

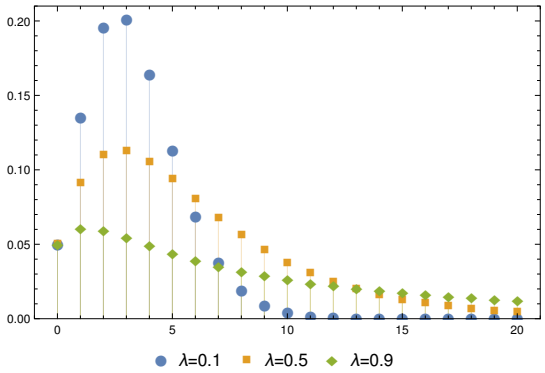


Figure 5.11: PDF-plot of the generalized Poisson distribution GenPois $(3.0, \lambda)$ for different λ.

5.2 Analysis of Changes and Results

Up to now, we have introduced the low-level and high-level changes between design models of Java software systems in Chapter 4. We have also provided the statistical distributions for analyzing the changes in Section 5.1. As discussed in Section 4.4, our data sets were the measurements of 75 low-level and 188 high-level difference metrics on the design models histories of 9 Java software systems. More formally, for each project p let $L_{p,m}$ denote the set of computed frequencies of the low-level metric m between revisions of p. Similarly by $H_{p,m}$ we denote the set of measurements of the high-level metric of m on the project p.

To decide whether or not a distribution $\mathcal{D}$ fits to a data set ($L_{p,m}$ or $H_{p,m}$), the null and alternative hypotheses, i.e. $\mathcal{H}_0$ and $\mathcal{H}_a$, are defined as follows:

$\mathcal{H}_0$: The data set obeys the distribution $\mathcal{D}$.

$\mathcal{H}_a$: 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) [Wackerly et al., 2007, Bohm and Zech, 2010]. 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. The MLE method

estimates the parameters by trying to maximize the logarithm of the likelihood function. The moments and/or the maximum-likelihood estimations of the discussed distribution are provided in detail in the accompanying website of the dissertation [Website, 2015]. In this chapter the MLE method is employed and the calculations are performed using the Wolfram Mathematica® 9.0.1 computational engine [Wolfram Research Inc., 2014].

Due to the discrete nature of the difference metrics and the six distributions, the Pearson's chi-square test was used [NIST, 2013] (see Section 5.2.4 for detailed discussion about the power of the Pearson's test against other alternative tests). The significance level was set to 0.05 for the test in our analysis. At first the parameters of the candidate distribution $\mathcal{D}$ were estimated, then the p-value of the Pearson's chi-square statistic was calculated in order to decide whether to reject $\mathcal{H}_0$ in favor of $\mathcal{H}_a$ or not.

For the 60 distributions that were initially tested, totally 40500 $(60 \times 9 \times 5 \times 15)$[11] fittings were considered for low-level changes and 101520 $(60 \times 9 \times 188)$[12] for high-level ones. From those, just the results for the six proposed distributions are covered in detail here, separately for low-level and high-level edit operations. In the rest of this section only successfully accomplished fittings are reported, i.e. when we were able to decide whether to reject $\mathcal{H}_0$ 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 there are too many (188) high-level operations we cannot publish detailed individual results of their analysis. Hence, we have grouped them into 6 categories namely Create, Delete, Move, Set/Unset, Modifying Non-Containment Reference and complex high-level operations, i.e. Refactorings (see Section 4.3). We provide only the summary of our findings for each category. Details about the fittings of the six distributions on all 188 high-level operations are available on the accompanying website of the dissertation [Website, 2015].

5.2.1 Analysis of Low-Level Changes

In this section we provide the results of the fittings of our six candidate distributions on low-level changes in detail.

5.2.1.1 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.

Totally 294 successful fittings were performed for low-level operations for the discrete Pareto distribution. $\mathcal{H}_0$ was not rejected 157 times, so the non-rejection ratio is about 53%.

The discrete Pareto distribution is most successful in describing changes of packages and interfaces, but with lower success rate for additions of new packages. It has generally

[11] 60 distributions, 9 projects, 5 low-level operations, 15 model element types.
[12] 60 distributions, 9 projects, 188 high-level operations.

a moderate rate of success in describing changes of classes and performs worse when fields are considered. The discrete Pareto distribution 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 5.12 shows one probability plot[13] 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 although $\mathcal{H}_0$ is rejected. Here, the p-value of the test was slightly lower than our significance level.

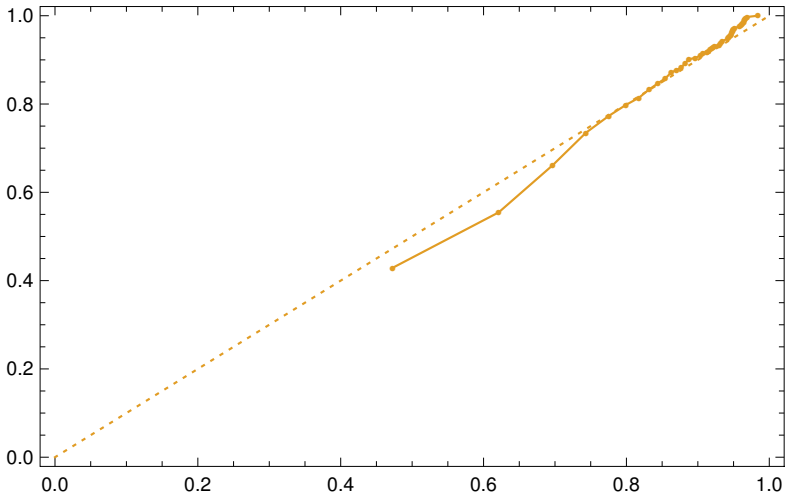


Figure 5.12: Probability plot of JFreeChart: addition of methods, the discrete Pareto distribution.

5.2.1.2 Beta Binomial Distribution

For the beta binomial distribution, $\mathcal{H}_0$ was not rejected 199 out of 294 times, which gives a non-rejection rate of almost 68%. This distribution shows a considerable improvement comparing to the discrete Pareto distribution whose success rate is at 54%.

The beta binomial distribution is fully successful in describing different changes on packages and interfaces. Regarding classes, it is just completely successful on modeling moves and updates. Addition, deletion and reference changes of classes is just modeled around one third of the times. Considering fields, it was just successful on modeling the moves but other change types were modeled at most half of the times. Changes on methods and parameters were modeled with various success rates of around 10% to 75%. For other model elements, all types of changes except additions were fully captured by

[13] The probability plot depicts the cumulative distribution functions (ranging from 0 to 1) of two data sets or distributions against each other in order to visually assess their closeness.

this distribution. Generally additions, except for packages and interfaces, were not quite successfully modeled by this distribution.

5.2.1.3 Yule Distribution

From the 294 fittings of the Yule distribution, $\mathcal{H}_0$ was rejected in favor of $\mathcal{H}_a$ 180 times, giving a non-rejection rate of almost 39% which makes this distribution the least successful one.

The Yule distribution 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 Yule distribution performs worse than the discrete Pareto distribution for both kinds of edit operations. Figure 5.13 shows the CDF[14] 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 (brown lines) and those which are obtained by the fitted distribution (red lines), which indicates that the fitting is bad and $\mathcal{H}_0$ is strongly rejected.

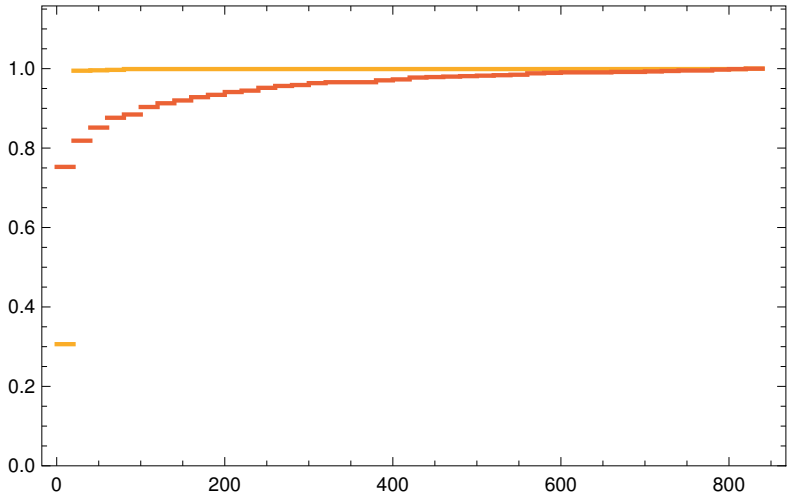


Figure 5.13: The CDF plot of the HSQLDB project: reference change of methods, the Yule distribution (Brown: observed probabilities, Red: fitted model).

5.2.1.4 Waring Distribution

For the Waring distribution, $\mathcal{H}_0$ was not rejected 252 out of 294 times, which gives a very good non-rejection rate of 86%.

[14] CDF: Cumulative Distribution Function.

The Waring distribution was fully successful in describing changes of packages and interfaces. For classes 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 5.14 shows the fitted Waring distribution to the histogram of additions of methods in the Maven project. As shown, the Waring distribution successfully models the changes. 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 4.3.2).

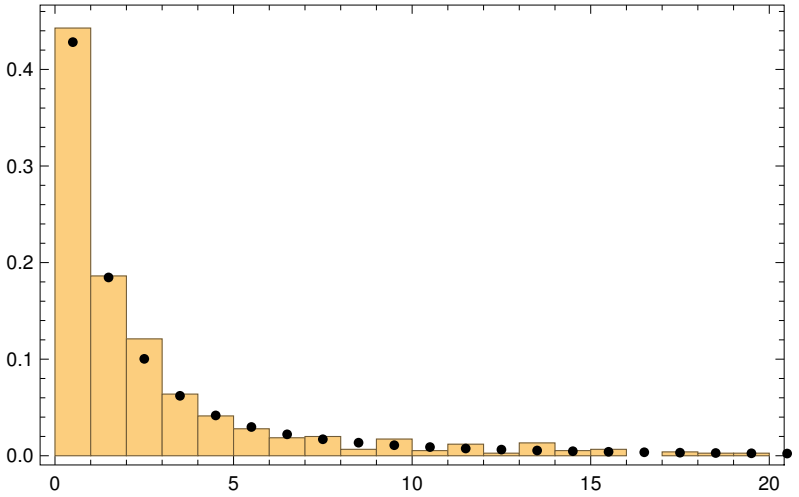
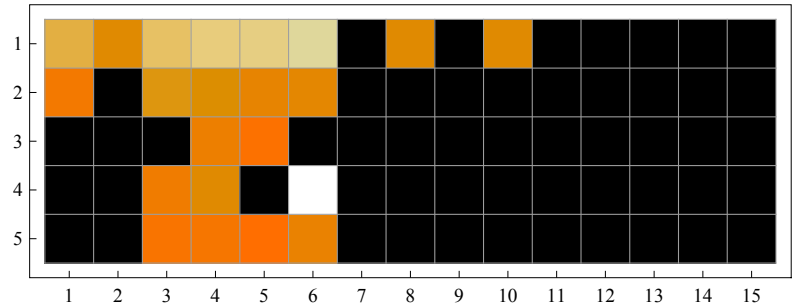


Figure 5.14: Plot of the fitted Waring distribution to the histogram of addition of methods in Maven.

Figure 5.15 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). Colors in the plots are used as follows: Black cells express that either all values of difference metrics were 0 or that the distribution could not be fitted because either the data did not fulfill the requirements of the distribution or parameters could not be estimated. The first case happens most of the time, while the second case occurs very rarely[15]. White cells indicate that the calculated p-value was less than the specified significance level, i.e. $\mathcal{H}_0$ was rejected. Finally, when the calculated p-value was above the significance level, i.e. $\mathcal{H}_0$ was not rejected, the cell is colored. The more intense the color of the cell, the higher the p-value.

[15] To be more precise, almost 0.02% of all possible fittings.



Note: The rows correspond to change types: 1. Additions, 2. Deletions, 3. Moves, 4. Reference Changes, 5. Attribute Updates.
The columns correspond to the element types: 1. Packages, 2. Interfaces, 3. Classes, 4. Fields, 5. Methods, 6. Parameters, 7. Projects, 8. Array Types, 9. Constants, 10. Simple Types, 11. Generic Types, 12. Template Wrappers, 13. Template Bindings, 14. Enumerations and 15. Enumeration Literals.

Figure 5.15: P-Value plot of the whole DataVision project when the Waring distribution is used.

5.2.1.5 Beta-Negative Binomial Distribution

For the beta-negative binomial distribution, $\mathcal{H}_0$ was rejected 34 out of 294 times in favor of $\mathcal{H}_a$, yielding an 88% non-rejection rate, which is a slight improvement to the Waring distribution.

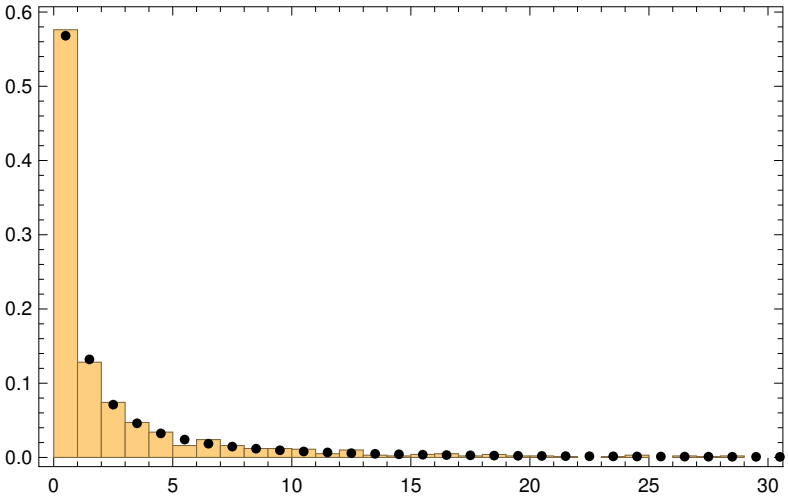
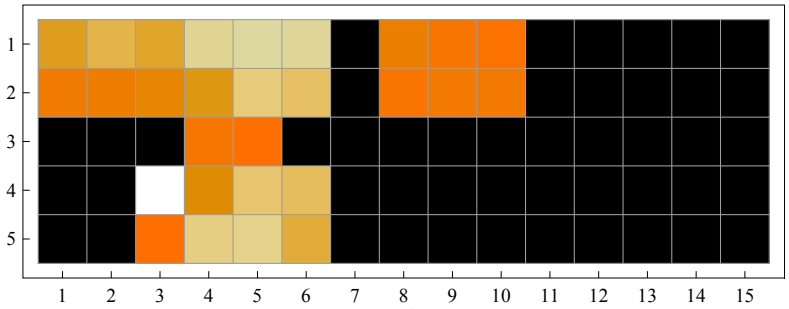


Figure 5.16: Plot of the fitted beta-negative binomial distribution to the histogram of addition of parameters in CheckStyle.

The beta-negative binomial and the Waring distributions performed almost identical for the difference metrics over the 15 element types. Like the Waring distribution,

the beta-negative binomial distribution is not successful in modeling reference changes for parameters with a success rate of only about 25%. Figure 5.16 shows the fitted beta-negative binomial distribution to the histogram of additions of parameters in the CheckStyle project. It can be seen that the predicted and observed probabilities completely overlap. Figure 5.17 shows the p-value plot of the Struts project which is almost fully statistically modeled by the beta-negative binomial distribution. In this particular example, only reference changes of interfaces could not be modeled (row=4, column=3).



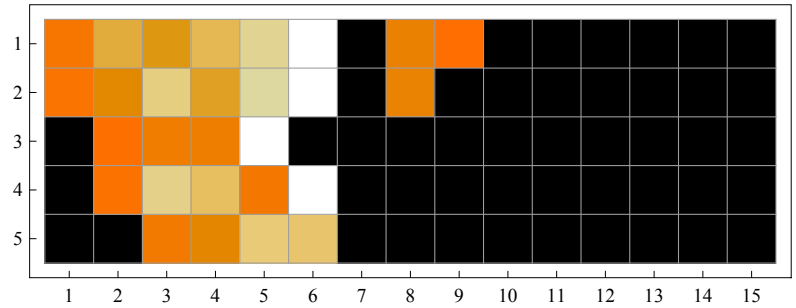
Note: The rows correspond to change types: 1. Additions, 2. Deletions, 3. Moves, 4. Reference Changes, 5. Attribute Updates.
The columns correspond to the element types: 1. Packages, 2. Interfaces, 3. Classes, 4. Fields, 5. Methods, 6. Parameters, 7. Projects, 8. Array Types, 9. Constants, 10. Simple Types, 11. Generic Types, 12. Template Wrappers, 13. Template Bindings, 14. Enumerations and 15. Enumeration Literals.

Figure 5.17: P-Value plot of the whole Struts project when the BNB distribution is used.

5.2.1.6 Generalized Poisson Distribution

For the generalized Poisson distribution, $\mathcal{H}_0$ was not rejected 244 out of 294 times which gives a success rate of around 83%. This is an slight decrease in comparison of the Waring distribution. The performance of this distribution is around 5% less than the beta-negative binomial distribution. Comparing to the Yule and the discrete Pareto distributions, this distribution is performing better with around 44% and 30% respectively.

The generalized Poisson distribution is completely successful in describing all types of changes on packages and interfaces. It is also completely successful in describing moves and updates on classes and fields. Regarding classes, fields, methods and parameters, it performs around 60% or more in describing additions, deletions and reference updates. For parameters, the success ratio is various from 20% to 90%. In particular, this distribution has a similar deficiency like the other distributions in describing reference update of parameters with around 20% of success rate. Figure 5.18 shows the p-value plot of the JFreeChart project. As shown, the generalized Poisson distribution generally has not been successful in describing different types of changes on parameters. Considering other model elements, this distribution is performing very good and almost captures all types of changes.



Note: The rows correspond to change types: 1. Additions, 2. Deletions, 3. Moves, 4. Reference Changes, 5. Attribute Updates.
The columns correspond to the element types: 1. Packages, 2. Interfaces, 3. Classes, 4. Fields, 5. Methods, 6. Parameters, 7. Projects, 8. Array Types, 9. Constants, 10. Simple Types, 11. Generic Types, 12. Template Wrappers, 13. Template Bindings, 14. Enumerations and 15. Enumeration Literals.

Figure 5.18: P-Value plot of the whole JFreeChart project when the generalized Poisson distribution is used.

5.2.2 Analysis of High-Level Changes

In this section we present the results of our statistical analysis for the high-level changes, first by summarizing the success rate of each distribution and later by providing more detail about the analysis results. The summarization is based on all successful fittings (where we can decide on the rejection of $\mathcal{H}_0$), done on all high-level metrics of all projects. Totally 554 successful fittings were performed on the high level changes. The rest were cases where the set of computed changes were all zero, i.e. we did not detect any occurrences of those changes in our analysis.

For the discrete Pareto distribution, totally $\mathcal{H}_0$ was not rejected 69% of all cases; a considerable improvement for this distribution compared to its application on low-level operations. We conclude that the discrete Pareto distribution serves better to describe high-level changes than low-level ones for which the success rate was at 53%.

Regarding the beta binomial distribution, the non-rejection rate of $\mathcal{H}_0$ was at 85% which shows around 17% improvement, quite similar to the discrete Pareto distribution, to its application to low-level changes whose success rate was at 68%. In this case, as for the discrete Pareto distribution, we conclude that the application of the beta binomial distribution is more promising when high-level changes are considered.

$\mathcal{H}_0$ was not rejected at the rate of 53% for the Yule distribution when considering all edit operations on all sample projects. This shows an improvement of 14% compared to low-level operations. Among all distributions, the Yule distribution also performs the worst for both low-level and high-level operations. Its success rate is much lower compared to its generalized counterparts, i.e. the Waring and the beta-negative binomial distributions.

The non-rejection ratio of the null hypotheses is almost at 93% for the Waring distribution, which is a good result. The Waring distribution performs very well in describing

Project Name	Pareto	Beta-Bino.	Yule	Waring	BNB	Gen. Pois.
ASM	80%	88%	57%	97%	97%	97%
CheckStyle	66%	83%	46%	85%	89%	83%
DataVision	96%	93%	89%	96%	96%	96%
FreeMarker	76%	80%	51%	94%	94%	95%
HSQLDB	57%	79%	30%	84%	91%	82%
Jameleon	74%	86%	60%	98%	96%	96%
JFreeChart	62%	92%	49%	96%	96%	96%
Maven	58%	81%	42%	93%	90%	85%
Struts	76%	81%	48%	97%	100%	92%
Average	71%	85%	53%	93%	94%	92%

Table 5.1: Success rates of each distribution for modeling high-level changes of each sample project.

the change behavior observed in the models, almost for all defined high-level operations. This distribution is performing almost 25% better compared to the discrete Pareto distribution and 40% comparing to the Yule distribution. In comparison to the beta binomial distribution, the Waring distribution is also superior. The success rate of this distribution is slightly higher than the generalized Poisson distribution and slightly lower than the beta-negative binomial distribution.

For the beta-negative binomial distribution, the non-rejection ratio of the null hypotheses is more than 94% which is the highest rate among all distributions. The success rate of this distribution is slightly higher than the Waring and the generalized Poisson distributions. This distribution performs the best in describing the evolution of class diagrams based on high-level changes.

The generalized Poisson distribution is also performing very good in describing different changes with a success rate of 92%. This rate is slightly lower than its competitors of the Waring and the beta-negative binomial distributions. This distribution also performs quite well in describing different high-level edit operations. Comparing to the beta binomial distribution, it performs quite better.

Table 5.1 shows the success rates of each of these six distributions for each sample project separately. The HSQLDB project could be weakly modeled by the discrete Pareto distribution while the DataVision project was almost completely modeled by this distribution. Generally the generalized Poisson, the Waring and the beta-negative binomial distributions are the most successful distributions in our table with almost 93% success rates.

Table 5.2 shows the success rates of each distribution when applied to each category of high-level operations (see Section 4.3.4). As the table shows, the generalized Poisson, the Waring and the beta-negative binomial distributions are the most successful ones when considering each category separately. These distributions perform only weak with

Edit Operations	Pareto	Beta-Bino.	Yule	Waring	BNB	Gen. Pois.
Create	62%	71%	47%	91%	91%	87%
Delete	72%	83%	57%	92%	95%	89%
Move	69%	89%	47%	94%	97%	94%
Set/Unset	73%	89%	48%	97%	97%	95%
Mod. Non-Cont. Ref.	59%	81%	39%	85%	87%	84%
Refactorings	93%	98%	69%	100%	100%	100%

Table 5.2: Success rates for each category of high-level changes.

operations modifying non-containment references. This is due to the fact that type changes of method parameters, which is a modification of a non-containment reference, are modeled by these two distributions with only 33% success rate. In fact, the high-level operation `changeParameterType` consists of just one low-level change. Hence this observation is consistent with our observations for low-level changes. Therefore, for modeling this edit operation, these three distributions should be used with caution.

All the generalized Poisson, the Waring and the beta-negative binomial distributions were 100% successful in describing model refactorings.

5.2.3 Conclusion of Analyses

With low-level operations, as discussed in Sections 5.2.1.1 and 5.2.1.3, the discrete Pareto distribution is to some extent successful in describing the observed changes. The Yule distribution is not generally recommended due to its low success rates and its failure to models most kinds of changes on various element types.

More precisely, the discrete Pareto distribution is basically 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 relatively well. When high-level operations are taken into account, the discrete Pareto distribution performs much better (near 70%) in describing the changes.

Since the discrete Pareto distribution is of the power law family, we additionally conclude that the "power law" is observable[16] 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[17].

The beta binomial distribution is moderately successful in describing low-level and high-level operations. It is more powerful than the discrete Pareto and the Yule distributions and less powerful than the generalized Poisson, the Waring and the beta-negative binomial distributions.

Considering low-level operations, the generalized Poisson, the Waring and the beta-negative binomial distributions show much higher success rates than the other three

[16] Actually based on the shifted data (see Section 5.2.1.1).

[17] See [Newman, 2005], for a short summary of where else the power law is also observed.

distributions. All of them perform quite well at explaining the observed difference metrics for almost all element types. Despite these successes, they are not completely successful in predicting reference changes of parameters and therefore should be used with caution in this specific case.

For the high-level operations the success rates of the generalized Poisson, the Waring and the beta-negative binomial distributions even increases to almost 93%; making them capable of statistically modeling almost any high-level edit operations in addition to low-level ones. These distributions are very good at modeling all categories of the high-level operations. Most of the times reaching near to 100% success rates.

Non-rejection values shows that the beta-negative binomial distribution is the most successful one among all of our distributions. Although the beta-negative binomial distribution is an extension to the Waring distribution and has one additional shape parameter, this does not add much benefit to its descriptive power. Furthermore, estimating the parameters of the Waring distribution needs less effort and is less time consuming.

Comprehensive information about our tests is provided in [Website, 2015]. This includes detailed tables of the estimated parameters, the summary of the fitting results as well as the cases of success and failure, and a complete set of p-value plots for each distribution.

5.2.4 Threats to the Validity of Analyses

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

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. [Wenzel, 2010] has analyzed this error for class diagrams and the SiDiff differencing framework [Kehrer et al., 2012b]; the total number of errors was typically below 2%. This very low error rate cannot have any 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 [Kehrer et al., 2011], 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 4.1 and Section 4.2) performed quite well and the results are not distorted.

Power of the Pearson's Chi-Square Test Another threat to validity can be the power[18] of the Pearson's test for fitting the distributions and for evaluating the null and the alternative hypotheses. One might argue that the power of this test is disputable when samples with small sizes are considered.

To answer this, as we discussed in Section 4.4, we used four criteria to select our sample projects for our statistical analysis (criteria **C1-C4**). We excluded the projects which were not developed over a long enough period of time or had few revisions, since this would prevent us to appropriately study the "evolution". Therefore, the size of our samples is not small (see Table 4.3) and the use of the Pearson's chi-square test would not devalue the results.

Furthermore, we should mention that other tests, e.g. discrete versions of Kolmogorov-Smirnov (KS) and Cramér-von Mises (CVM) are also available for discrete data. The power of these tests in comparison with the Pearson's test for small samples can also be of interest. It is known that the power of a test highly depends on the sample size, the null and alternative hypotheses as well as if the hypotheses are simple or complex [Lemeshko et al., 2007].

For small samples, Steele et al. have studied the power of six different tests (including the Pearson's, the KS and the CVM tests) considering uniform null distribution against few alternative distributions (Decreasing, Step, Triangular, Platykurtic, Leptokurtic and Bimodal) [Steele and Chaseling, 2006, Steele et al., 2007]. Generally, they conclude that it is not possible to prefer one test to the others. Their detailed results show that the Pearson's test is even performing better than the others in some cases. Thus, this additionally confirms the suitability of the Pearson's test for our analysis even if our sample sizes were small.

External Validity Another important question is whether our results are generalizable. Principally, the approach of extracting the change information between two diagrams (see processing pipeline at Figure 4.1) can be repeated on other model types as long as the employed matching and difference derivation engines can be accordingly adapted. Furthermore, our statistical analysis of changes are independent of any diagrams type and just focuses on the frequencies of detected edit operations in the computed differences. However, it is unclear whether the frequencies of edit operations in other types of diagrams exhibit the same statistical distributions. Related analyses should be a topic of further research.

Additionally, 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 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++ or C#. The question is whether the data model for class diagrams (see

[18] Power of a test is the probability to reject the null hypothesis, when it is wrong.

Figure 4.3) is still appropriate. Since the concepts of object-oriented are similar in most languages, the concept can be readily adapted.

5.3 Generating Random Variates of the Proposed Distributions

As discussed in Sections 5.2.1 and 5.2.2, the discrete Pareto, the beta binomial, the Yule, the Waring, the beta-negative binomial and the generalized Poisson distributions were successful in order to statistically model the changes in the design models of software systems. We can employ their random variates in the configuration of our model generator (SMG) in order to synthesize realistic test models (see Section 3.3). We should note that some of the methods which will be presented in this section, have been successfully implemented in mathematical computation engines like Matlab, R, Maple and Mathematica as well as numerical libraries such as NAG (Numerical Algorithms Group), RAGE (Random Variate Generator), SSJ (Stochastic Simulation in Java), GNU Random Number Generation and www.netlib.org. To support our methodological framework to simulate the evolution, and for the sake of self containment and clarity we provide related algorithms in detail.

In this section we introduce and present methods and techniques that can be used in generating random variates of each of the six proposed distributions. In Section 5.3.1, we first provide a general overview on how random variates of arbitrary distributions are generated. In Section 5.3.2, we introduce an algorithm for generating random variates of the discrete Pareto distribution. Section 5.3.3 is entirely devoted to the generation of random variates of the beta-negative binomial, the Waring and the Yule distributions. Section 5.3.4 discusses how random variate of the beta binomial distribution is generated. Random variate generation of the generalized Poisson distribution is discussed in Section 5.3.5. Finally, a summary about generating of random variates is provided in Section 5.3.6.

5.3.1 Introduction to Random Variate Generation

The term "Random Number Generation" (RNG) refers to methods used in producing sequences of independent and identically distributed (iid) numbers from the uniform distribution of $\mathrm{U}[0,1]$. Randomly generated numbers of $\mathrm{U}[0,1]$ are then employed in generating random numbers of distributions other than the uniform distribution, using different techniques e.g. transformations. This procedure is referred to as "Random Variate Generation" (RVG) of a given (non-uniform) distribution [Banks, 1998, Banks et al., 2010].

Roughly speaking, the RNG methods are deterministic programs with finite set of states including an initial state (called a seed) and a mapping (transient function) which maps those states to themselves. The states correspond to a finite set of output symbols that the program produces. The role of the transient function is to create the next state based on the previous state [Banks, 1998]. Since the states are finite, the output of the

program is also finite and the generator repeats itself with a period. Due to this reason these methods are usually referred to as "pseudo random number generators" since they do not really create iid random numbers, but they try to meet the statistical requirements for truly iid uniform random numbers as much as possible. These requirements are, uniformity and independence of the generated numbers as well as greater repetition periods [Banks et al., 2010, Banks, 1998].

Generating truly uniform random numbers is still an active research field and different RNG methods have been proposed. *Linear congruential method* and *combined linear congruential method* are frequently used and implemented ([Knuth, 1998]) and can be also employed here. We should also note that better methods are also available. In this section we suppose that a suitable random number generator which creates iid uniform random numbers is available and we do not go into any further details, rather we try to focus on generating random variates of our proposed distributions. L'Ecuyer has studied in detail the requirements of good generators as well as different RNG methods in [Banks, 1998, L'Ecuyer, 1994].

For generating random variates of a known distribution (other than the uniform distribution) different methods have been proposed such as inverse transform, acceptance-rejection, composition and transformation. In this section, we introduce the inverse transform and the acceptance-rejection methods which are used later for generating our desirable random variates of our six distributions. Before that we provide the basic requirements.

Suppose $F_X(x) = \mathrm{P}[X \leq x]$ is the *cumulative distribution function* (CDF) of the random variable X for which we intend to generate its random variates. Generally, F is a non-decreasing function and its range lies within the interval of $[0, 1]$. In the case that X is a continuous random variable we have:

$$F(x) = \int_{-\infty}^{x} f(z)\,\mathrm{d}z$$

where $f(x)$ is the *density function* of X. F is also continuous and strictly increasing in this case.

If X is a discrete random variable which takes the values of $x_1 < x_2 < \cdots$ and if the probability of x_i is $p_i = \mathrm{P}[X = x_i]$, then we have:

$$F(x) = \sum_{x_i \leq x} \mathrm{P}[X = x_i] = \sum_{x_i \leq x} p_i$$

The inverse of F is defined as [Banks, 1998, Devroye, 1986, Embrechts and Hofert, 2010, Law, 2007a]:

$$F^{-1}(u) = \inf\{x \in \mathbb{R} \mid u \leq F(x)\},\ 0 < u < 1$$

and is usually referred to as the *generalized inverse distribution function* of X. Note that this definition allows discontinuity and is used for both cases where X is continuous or discrete. Additionally we have the following theorem:

Theorem. Let X be a random variable with cumulative distribution function of F, i.e. $X \sim F$, then we have:

1. If F is continuous, $F(X) \sim \mathrm{U}[0,1]$.
2. If $U \sim \mathrm{U}[0,1]$, $F^{-1}(U) \sim F$.

Proof. For the proof please refer to [Embrechts and Hofert, 2010, Devroye, 1986, Ripley, 1987]. □

The above theorem is the infrastructure of the *inverse transform method* for generating random variates. In this method we generate a random number u from the uniform distribution $\mathrm{U}[0,1]$ and we try to solve the equation of $x = F^{-1}(u)$. The theorem guarantees that x is a random variate of the distribution F.

This method is very straightforward when F is continuous and F^{-1} exists in closed form. Most of the time, it is difficult or even impossible to calculate F^{-1} in closed form. In this situation some numerical solvers such as the Bisection, the Secant and the Newton-Raphson methods can be employed [Devroye, 1986, Burden and Faires, 2000]. In the case that F is discrete, by considering the definition of F^{-1}, we generate u of $\mathrm{U}[0,1]$ and we note that we should return $x_k \in \{x_1, x_2, \ldots\}$ as the desired random variate which satisfies the following relation:

$$F(x_{k-1}) = \sum_{x_i < x_k} p_i < u \leq \sum_{x_i \leq x_k} p_i = F(x_k)$$

In other words we look for smallest k such that $u \leq F(x_k)$.

The second method that we are going to discuss is called the *acceptance-rejection method*. Suppose the $f(x)$ is the density function of X and suppose that we have a dominate function, say $t(x)$, such that $t(x) \geq f(x)$ for all x. Intuitively in this method, we try to generate (X, Y) points in the x-y plane in such a way that they are uniformly scattered under the graph of $t(x)$ and we accept X as the desired random variate of $f(x)$ when $Y \leq f(X)$ and otherwise we reject it. Therefore the rejection is proportional to the area between $f(x)$ and $t(x)$ and we try to have $t(x)$ in a way that this area is as small as possible. Another important factor in taking $t(x)$ is that the uniform construction of points under it should be fast and easy [Banks, 1998, Law, 2007a]. Since $t(x) \geq f(x)$ we have:

$$c = \int_{-\infty}^{\infty} t(x) \geq \int_{-\infty}^{\infty} f(x) = 1$$

and we assume that c is finite. Therefore if we take $g(x) = f(x)/c$, then $g(x)$ is a distribution. We should choose $t(x)$ in a way that generating random variates of $g(x)$ is easy and fast. The general case of the acceptance-rejection method is given by Algorithm 1 [Law, 2007a]. Calling the algorithm n times produces n iid random variates of the distribution with the density function $f(x)$.

Although the algorithm might not seem trivial on the first sight, it can be shown that the returned values of X obey the density function $f(x)$. Here we interpret the

Algorithm 1: General acceptance-rejection method for generating a random variate of the density function $f(x)$.

1 **repeat**
2 | Generate G, a random variate of the density function $g(x)$;
3 | Generate U, a random number of the uniform distribution $\mathrm{U}[0,1]$ independently of G ;
4 **until** $U \leq f(G)/t(G)$;
5 **return** $X = G$;

algorithm without going into the detailed lengthy mathematics. For the formal proof of the algorithm please refer to [Law, 2007a].

The algorithm produces a point (G, U) in the x-y plane in steps 2 and 3. The G component of the point is generated based on the distribution $g(x)$. We get more points where $g(x)$ is denser. In step 4, it checks if the generated point is suitable. In this step the uniformly generated U is rescaled by multiplying it to the height of $t(x)$ at point $x = G$ and if it is not greater than $f(G)$, the point is accepted and its first component, i.e. G, is returned as the random variate of the density function $f(x)$.

It can be shown that the acceptance probability for the generated points in the algorithm is $1/c$ [Law, 2007a]. Therefore it is desirable that $t(x)$ is chosen in a way that c ($c \geq 1$) is as close as possible to 1 in order to have less points rejected in step 4. As we discussed earlier the other goal is that the generation of random variates of $g(x)$ is easy and quick.

5.3.2 Random Variates of the Discrete Pareto Distribution

Here we provide the acceptance-rejection algorithm for generating random variates of the discrete Pareto distribution which is given in [Devroye, 1986]. It is shown that if $U \sim \mathrm{U}[0,1]$ then the density function of the random variable $Y = \left\lfloor U^{\frac{-1}{s-1}} \right\rfloor$ is a good dominating function[19] [20] for the density function of the discrete Pareto distribution (see Equation (5.3) in Section 5.1.2). Based on this fact, Algorithm 2 generates random variates of the discrete Pareto distribution.

5.3.3 Random Variates of the Yule, Waring and Beta-Negative Binomial Distributions

There are well established references for statistical distributions that cover almost all of known distributions introduced in different research fields and discuss related literature, e.g. reference books by Johnson et al and also by Wimmer and Altmann [Forbes et al., 2011, Johnson et al., 1992, 2005, 1997, Krishnamoorthy, 2006, Walck, 2007, Wimmer

[19] $\rho = s - 1$ is the parameter of the discrete Pareto distribution (Equation (5.3)).
[20] $\lfloor x \rfloor$ is the floor function of x which is by definition, the largest integer not greater than x.

Algorithm 2: Generating a random variate of the discrete Pareto distribution.

Input: $\rho > 0$; /* shape parameter of the discrete Pareto distribution */

Output: X ; /* a random variate of the discrete Pareto distribution with the shape parameter ρ */

/* Initialization */

$s \leftarrow \rho + 1$;

$b \leftarrow 2^{s-1}$;

/* Generator */

repeat

  Generate two independent and identically distributed uniform random numbers U and V of $\mathrm{U}[0,1]$;

  $X \leftarrow \left\lfloor U^{\frac{-1}{s-1}} \right\rfloor$;

  $T \leftarrow \left(1 + \frac{1}{X}\right)^{s-1}$;

until $\frac{X\,V\,(T-1)}{b-1} \leq \frac{T}{b}$;

return X ;

and Altmann, 1999]. The same can be said regarding simulation and random variate generation, e.g. [Banks, 1998, Banks et al., 2010, Devroye, 1986, Law, 2007a, Fishman, 2001, Saucier, 2000, Ross, 2006, Bennett, 1995, Bratley et al., 1983, Ripley, 1987, R-forge, 2008-2009]. There is no tailored algorithm for generating random variates of the beta-negative binomial distribution in the previously mentioned references. The same is true when considering all other related literature, e.g. the works of Irwin, Xekalaki, Wang and others, which we do not cite them here. Therefore, we have to use other statistical properties of the beta-negative binomial distribution in order to generate its random variates.

As we explained earlier in Section 5.1.4, the beta-negative binomial distribution can be obtained from the negative binomial distribution when the probability of success in Bernoulli trials, i.e. p, changes according to the beta distribution. We use this fact in order to produce random variates of the beta-negative binomial distribution[21]. Therefore we need an algorithm for generating random variates of the beta distribution and also another method for generating random variates of the negative binomial distribution. Algorithm 3 generates random variates of the $\mathrm{BNB}(\alpha, \beta, n)$ distribution based on this property and the assumption that algorithms for generating random variates of the beta and the negative binomial distributions are already provided. Furthermore, we already mentioned that the Yule and the Waring distributions can be considered as special cases of the beta-negative binomial distribution, so their random variates can be

[21] The RVG method presented in R package of "SuppDists" uses an indirect method based on the hypergeometric distribution which is more complicated than the one we present here.

directly obtained from Algorithm 3 by setting appropriate values in the corresponding beta-negative binomial distribution (see Section 5.1.4).

Algorithm 3: Generating a random variate of the beta-negative binomial distribution $\mathrm{BNB}(\alpha, \beta, n)$.

Input: $\alpha > 0$, $\beta > 0$ and $n > 0$; /* the shape parameters of the beta-negative binomial distribution */

Output: X ; /* a random variate of the beta-negative binomial distribution $\mathrm{BNB}(\alpha, \beta, n)$ */

/* Generator */

1 Generate B, a random variate from the beta distribution with parameters α and β, i.e. $\mathrm{Beta}(\alpha, \beta)$, using Algorithm 4 ;

2 Generate X, a random variate from the negative binomial distribution with parameters n and $p = B$, i.e. $\mathrm{NegBino}(n, B)$, using Algorithm 5 ;

3 **return** X ;

Unfortunately the generation of the random variates for the beta and the negative binomial distributions are not straightforward. In Section 5.3.3.1, we first focus on generating random variates of the beta distribution. Since the random variate generation of the negative binomial distribution is complex and is based on the random variate generation of the Poisson and the gamma distributions, we devote Section 5.3.3.2 completely for this purpose.

5.3.3.1 Random Variates of the Beta Distribution

The PDF of the beta distribution with the shape parameters of α and β (denoted by $\mathrm{Beta}(\alpha, \beta)$) is given by [Johnson et al., 1994, Forbes et al., 2011]:

$$\mathrm{P}[X = x] = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{\mathrm{B}(\alpha, \beta)}, \quad 0 \leq x \leq 1, \alpha > 0, \beta > 0 \text{ and } x, \alpha, \beta \in \mathbb{R}. \tag{5.11}$$

in which $\mathrm{B}(\alpha, \beta)$ is the beta function (see Section 5.1.1).

Several methods have been proposed for generating random variates of the beta distribution [Walck, 2007, Devroye, 1986, Law, 2007a]. The algorithm we present here is an acceptance-rejection method proposed by [Cheng, 1978]. It is extensively used and is easy and straightforward to be implemented. Cheng offers a basic algorithm which generates random variates for all values of α and β. Some speedup has been gained by altering the basic algorithm for two separate cases of $\min(\alpha, \beta) > 1$ and $\min(\alpha, \beta) \leq 1$. Algorithm 4 is the Cheng‘s basic algorithm which generates random variates of the beta distribution for all values of shape parameters.

Algorithm 4: Generating a random variate of the Beta (α, β) distribution.

Input: $\alpha > 0$ and $\beta > 0$; /* shape parameters of the beta distribution */

Output: X ; /* a random variate of the beta distribution Beta (α, β) */

/* Initialization */

$s \leftarrow \alpha + \beta$;

if $\min(\alpha, \beta) \leq 1$ **then**

  $\lambda \leftarrow \max\left(\frac{1}{\alpha}, \frac{1}{\beta}\right)$;

else

  $\lambda \leftarrow \sqrt{\frac{s-2}{2\alpha\beta - s}}$;

end

$\gamma \leftarrow \alpha + \frac{1}{\lambda}$;

/* Generator */

repeat

  Generate U_1 and U_2, two independent and identically distributed uniform random numbers of U $[0, 1]$;

  $v \leftarrow \lambda \ln\left(\frac{U_1}{1-U_1}\right)$;

  $w \leftarrow \alpha\, e^v$;

until $s \ln\left(\frac{s}{\beta+w}\right) + \gamma\, v - \ln(4) \geq \ln\left(U_1^2\, U_2\right)$;

$X \leftarrow \frac{w}{\beta+w}$;

return X ;

5.3.3.2 Random Variates of the Negative Binomial Distribution

The PDF of the negative binomial distribution[22] with shape parameters of n and p (denoted by NegBino (n, p)) is given by [Wimmer and Altmann, 1999, Forbes et al., 2011]:

$$\begin{gathered} \mathrm{P}[X = x] = \frac{\Gamma(n+x)\, p^n\, (1-p)^x}{\Gamma(n)\, \Gamma(x+1)}, \\ x = 0, 1, 2, \cdots,\ 0 < p < 1,\ n > 0 \text{ and } n \in \mathbb{R}. \end{gathered} \tag{5.12}$$

When n is an integer, the density function can be expressed using the binomial

[22] For the interpretation of the negative binomial distribution, please see Section 5.1.1.

coefficients[23] as follows:

$$\begin{aligned} \mathrm{P}\left[X = x\right] &= \binom{n+x-1}{n-1} p^n \left(1-p\right)^x, \\ x &= 0, 1, 2, \cdots, 0 < p < 1 \text{ and } n \in \mathbb{N}. \end{aligned} \tag{5.13}$$

For generating random variates of the negative binomial distribution we use the fact that the negative binomial distribution is obtained from the Poisson distribution when the parameter of the Poisson distribution obeys the gamma distribution [Wimmer and Altmann, 1999, Johnson et al., 1992, Devroye, 1986]. More formally, suppose that n and p are parameters of the negative binomial distribution and X obeys the distribution, i.e. $X \sim \mathrm{NegBino}\left(n, p\right)$, then $X \sim \mathrm{Pois}\left(\mu\right)$ where $\mu \sim \mathrm{Gamma}\left(n, (1-p)/p\right)$.

Therefore for generating random variates of the negative binomial distribution we should generate a random variate of the Poisson distribution when its shape parameters is generated from the gamma distribution. Algorithm 5 generates random variates of the negative binomial distribution using the above technique. In this section we provide algorithms for generating random variates of the Poisson and the gamma distributions after providing the formal definition of these two distributions.

Algorithm 5: Generating a random variate of the negative binomial distribution $\mathrm{NegBino}\left(n, p\right)$.

Input: $n > 0$ and $0 < p < 1$; `/* shape parameters of the negative binomial distribution */`

Output: X ; `/* a random variate of the negative binomial distribution` $\mathrm{NegBino}\left(n, p\right)$ `*/`

`/* Generator */`

1 Generate G, a random variate of the gamma distribution using Algorithm 9 with shape parameters of $\alpha = n$ and $\beta = \left(1-p\right)/p$, i.e. $\mathrm{Gamma}\left(n, \left(1-p\right)/p\right)$;

2 Generate X, a random variate of the Poisson distribution using Algorithm 6 with shape parameter $\mu = G$, i.e. $\mathrm{Pois}\left(G\right)$;

3 **return** X ;

Algorithm 6 generates random variates of the Poisson distribution [Devroye, 1986, Banks, 1998]. The algorithm uses the following theorem which is proven in [Devroye, 1986]:

Theorem. Let $U_1, U_2, \ldots$ be independent and identically distributed random numbers of the uniform distribution $\mathrm{U}\left[0, 1\right]$ and let X be the smallest integer such that $\prod_{i=1}^{X+1} U_i < e^{-\mu}$, then $X \sim \mathrm{Pois}\left(\mu\right)$.

[23] $\binom{n}{k} = \frac{n!}{k!(n-k)!}$.

Algorithm 6: Generating a random variate of the Poisson distribution.

Input: $\mu > 0$; /* shape parameter of the Poisson distribution */

Output: X ; /* a random variate of the Poisson distribution with the parameter μ */

/* Initialization */

1 $a \leftarrow e^{-\mu}$;
2 $P \leftarrow 1$;
3 $X \leftarrow -1$;

/* Generator */

4 **while** $P \geq a$ **do**
5 | Generate U, a random number of the uniform distribution $\mathrm{U}[0,1]$;
6 | $P \leftarrow P\,U$;
7 | $X \leftarrow X + 1$;
8 **end**
9 **return** X ;

The PDF of the gamma distribution, which has applications in modeling of waiting times, with parameters of α and β (denoted by $\mathrm{Gamma}(\alpha, \beta)$) is given by [Johnson et al., 1994]:

$$\mathrm{P}[X = x] = \frac{(x/\beta)^{\alpha-1}\, e^{-x/\beta}}{\beta\, \Gamma(\alpha)},$$
$$x, \alpha, \beta > 0 \text{ and } x, \alpha, \beta \in \mathbb{R}.$$

in which α is referred to as the shape parameter and β is called the scale parameter. Generating random variates of the gamma distribution is not quite straightforward since there is no single algorithm that can generate random variates for all values of α and β [Devroye, 1986, Banks, 1998].

Ahrens and Dieter have provided an acceptance-rejection method for the case $0 < \alpha < 1$ and $\beta = 1$ [Ahrens and Dieter, 1974]. They used $g(x) = x^{\alpha-1}/\Gamma(\alpha)$ as the dominating function when $0 < x \leq 1$. When $x \geq 1$, they used $g(x) = e^{-x}/\Gamma(\alpha)$. Cheng has slightly modified the algorithm to cover all values of the scale parameter β [Banks, 1998]. Algorithm 7 delivers random variates of the gamma distribution for the case $0 < \alpha < 1$.

If $\alpha \geq 1$ we propose the Algorithm 8 by Cheng [Cheng, 1977, Banks, 1998]. He used $g(x) = \lambda\, \mu\, x^{\lambda-1} / \left(\mu + x^{\lambda}\right)^2$ when $x > 0$ as the dominating function in which λ and μ are parameters which can be adjusted. Finally, Algorithm 9 combines these two into one framework for generating random variates of the gamma distribution.

We should mention that when n is an integer and the negative binomial distribution is given by Equation (5.13), other random variate generation algorithms are also available

Algorithm 7: Generating a random variate of the gamma distribution Gamma (α, β) when $0 < \alpha < 1$.

Input: $0 < \alpha < 1$ and $\beta > 0$; /* shape parameters of the gamma distribution */

Output: X ; /* a random variate of the gamma distribution with the parameters of α and β using the algorithm by Ahrens and Dieter */

/* Initialization */

$\lambda \leftarrow (e + \alpha)/e$; /* e is the base of the natural logarithm */

/* Generator */

while *True* **do**
  Generate U, a random number of the uniform distribution U $[0, 1]$;
  $W \leftarrow \lambda U$;
  if $W < 1$ **then**
    $Y \leftarrow W^{(1/\alpha)}$;
    Generate V, a random number of the uniform distribution U $[0, 1]$;
    if $V \leq e^{-Y}$ **then**
      return $X = \beta Y$;
    end
  else
    $Y \leftarrow -\ln\left(\frac{\lambda - W}{\alpha}\right)$;
    Generate V, a random number of the uniform distribution U $[0, 1]$;
    if $V \leq Y^{\alpha - 1}$ **then**
      return $X = \beta Y$;
    end
  end
end

e.g. the ones provided in [Ahrens and Dieter, 1974]. Before closing this section we provide one of these methods which is simple and can be used when we need to generate random variates of the negative binomial distribution with an integer parameter of n. It cannot be used when n is a real number, in this case, as we mentioned, Algorithm 5 should be employed.

In short, as we saw earlier if n is an integer, the PDF of the negative binomial distribution can be expressed by binomial coefficients (see Equation (5.13)). It has been shown that in this case the negative binomial distribution NegBino (n, p) is the sum of n independent and identically distributed random variates of the geometric distribution each with parameter p [Johnson et al., 2005]. In other word, if $X_1, X_2, \ldots, X_n \overset{\text{iid}}{\sim} \text{Geom}(p)$, then $X_1 + X_2 + \ldots + X_n \sim \text{NegBino}(n, p)$ [Law, 2007a].

The geometric distribution is the distribution of the number of failures in a sequence

Algorithm 8: Generating a random variate of the gamma distribution $\mathrm{Gamma}(\alpha, \beta)$ when $\alpha \geq 1$.

Input: $\alpha \geq 1$ and $\beta > 0$; /* shape parameters of the gamma distribution */

Output: X ; /* a random variate of the gamma distribution with the parameters of α and β using the algorithm by Cheng */

/* Initialization */

$\lambda \leftarrow (2\,\alpha - 1)^{-1/2}$;

$\mu \leftarrow \alpha - \ln(4)$;

$\gamma \leftarrow \alpha + 1/\lambda$;

$\delta \leftarrow 1 + \ln(4.5)$;

/* Generator */

while *True* **do**

  Generate $U1$, a random number of the uniform distribution $\mathrm{U}[0, 1]$;

  Generate $U2$, a random number of the uniform distribution $\mathrm{U}[0, 1]$, independently of $U1$;

  $V \leftarrow \lambda \ln\left(\frac{U1}{1-U1}\right)$;

  $Y \leftarrow \alpha\, e^{V}$;

  $Z \leftarrow U1^2\, U2$;

  $W \leftarrow \mu + \gamma\, V - Y$;

  if $W + \delta - 4.5\, Z \geq 0$ **then**

    return $X = \beta\, Y$;

  else

    if $W \geq \ln(Z)$ **then**

      return $X = \beta\, Y$;

    end

  end

end

Algorithm 9: Generating a random variate of the gamma distribution Gamma (α, β).

Input: $\alpha > 0$ and $\beta > 0$; /* shape parameters of the gamma distribution */

Output: X ; /* a random variate of the gamma distribution with the parameters of α and β */

/* Generator */

if $\alpha < 1$ **then**
  Generate G, a random variate of the gamma distribution using Algorithm 7 ;
else
  Generate G, a random variate of the gamma distribution using Algorithm 8 ;
end
return $X = G$;

of Bernoulli trials, with the success probability p, before the first success happens. The PDF of the geometric distribution with parameter p (denoted by Geom (p)) is given by [Wimmer and Altmann, 1999]:

$$\mathrm{P}[X = x] = p(1-p)^x,$$
$$x = 0, 1, 2, \cdots, 0 < p < 1 \text{ and } p \in \mathbb{R}.$$

Algorithm 10 is an inverse-transform method for generating random variates of the geometric distribution [Law, 2007a].

Algorithm 10: Generating a random variate of the geometric distribution Geom (p).

Input: $0 < p < 1$; /* the probability of success in the Bernoulli trial */

Output: X ; /* a random variate of the geometric distribution with the parameter p */

/* Generator */

Generate U, a random variate of the uniform distribution U $[0, 1]$;
$X \leftarrow \left\lfloor \frac{\ln(U)}{\ln(1-p)} \right\rfloor$;
return X ;

5.3.4 Random Variates of the Beta Binomial Distribution

A random variate generation algorithm for BetaBino (α, β, n), i.e. the beta binomial distribution, can be designed based on the fact that the beta binomial distribution is

principally the binomial distribution Bino (n, p) in which the probability of its Bernoulli trials obeys the beta distribution with shape parameters of α and β, i.e. $p \sim \text{Beta}(\alpha, \beta)$ [Wimmer and Altmann, 1999]. The PDF of the binomial distribution with shape parameters of n and p, i.e. Bino (n, p), is given by [Banks et al., 2010]:

$$\begin{gathered} \mathrm{P}[X = x] = \binom{n}{x} p^x (1-p)^{n-x} \\ x \in \{0, 1, \ldots, n\}, \, 0 \leq p \leq 1 \text{ and } x \in \mathbb{R}. \end{gathered} \tag{5.14}$$

Algorithm 11 uses the above technique to generate random variates of the beta binomial distribution of BetaBino (α, β, n). The algorithm assumes that a random variate generator for the beta and the binomial distributions are available. The random variate generation of the beta distribution is addressed by Algorithm 4 in Section 5.3.3.1. Regarding random variate generation of the binomial distribution, on can use the fact that if $X_1, X_2, \ldots, X_n$ are independent and identically distributed random variates of the Bernoulli trials with success probability of p, i.e. $X_1, X_2, \ldots, X_n \overset{\text{iid}}{\sim} \text{Bernoulli}(p)$, then $X_1 + X_2 + \ldots + X_n \sim \text{Bino}(p, n)$ [Law, 2007a, Banks, 1998]. Algorithm 12 uses this property for generating random variates of the binomial distribution.

Although Algorithm 12 is simple, straightforward and frequently used in practice (e.g. [Walck, 2007, R-forge, 2008-2009]), its execution time linearly grows with n and for big values of n it is not quite efficient. Information about other faster but much more complicated approaches can be found in [Kachitvichyanukul and Schmeiser, 1988, Ahrens and Dieter, 1974, Devroye, 1986].

Algorithm 11: Generating a random variate of the beta binomial distribution BetaBino (α, β, n).

Input: $\alpha > 0$, $\beta > 0$ and $n > 0$; `/* the shape parameters of the beta binomial distribution */`

Output: X ; `/* a random variate of the beta binomial distribution` BetaBino (α, β, n) `*/`

`/* Generator */`

1 Generate B, a random variate from the beta distribution with parameters α and β, i.e. Beta (α, β), using Algorithm 4 ;

2 Generate X, a random variate from the binomial distribution with parameters n and $p = B$, i.e. Bino (n, B), using Algorithm 12 ;

3 **return** X ;

5.3.5 Random Variates of the Generalized Poisson Distribution

It is reported that random variates of the generalized Poisson distribution cannot be obtained through simple combination of more primitive random variates generation meth-

Algorithm 12: Generating a random variate of the binomial distribution $\text{Bino}(p, n)$.

Input: $0 \leq p \leq 1, n > 0$; /* the shape parameters of the binomial distribution */

Output: X ; /* a random variate of the binomial distribution $\text{Bino}(p, n)$ */

```
/* Initialization                                                    */
X ← 0;
i ← 1;
/* Generator                                                         */
while i ≤ n do
    /* generate B, a random variate of Bernoulli(p)                 */
    Generate U, a random number of the uniform distribution U[0, 1] ;
    if U ≤ p then
        B ← 1 ;
    else
        B ← 0 ;
    end
    /* sum Bernoulli random variates to form a binomial
       distribution random variate                                   */
    X ← X + B ;
    i ← i + 1 ;
end
return X ;
```

ods of the Poisson, binomial and normal distributions. Algorithm 13 is an acceptance-rejection method for generating random variates of the generalized Poisson distribution and is proposed by [Devroye, 1989]. The algorithm is based on a theorem which establishes an upper bound for the PDF of the Poisson distribution (given by Equation (5.10) in Section 5.1.5). The theorem and other technical details are discussed there as well.

5.3.6 Summary of Random Variate Generations

To sum up, in Section 5.3 we talked about how our six distributions can be successfully implemented and how we can generate their random variates. In this regard, we discussed about the inverse transform and the acceptance-rejection methods which are widely used to generate random variates. The challenge was that random variates of none of our distributions could be easily obtained through the inverse transform method. This is due the fact that the closed form of their inverse CDF does not exist. Therefore, random variate generation of these distributions are typically done through the acceptance-rejection method or by employing their specific statistical properties and

Algorithm 13: Generating a random variate of the generalized Poisson distribution $\mathrm{GenPois}(\mu, \lambda)$.

Input: $\mu > 0$ and $\lambda \in [0, 1]$; `/* shape parameters of the generalized Poisson distribution */`

Output: X ; `/* a random variate of the generalized Poisson distribution of` $\mathrm{GenPois}(\mu, \lambda)$ `*/`

`/* Initialization */`

$p_0 \leftarrow e^{-\mu}$;

$b \leftarrow \mu\, e^{2-\lambda-\min(\mu,\lambda)}\sqrt{\frac{2}{\pi}}$;

`/* Generator */`

Generate U, a random number of the uniform distribution $\mathrm{U}[0, 1]$;

repeat

  if $U \leq \frac{p_0}{p_0+b}$ **then**

    $X \leftarrow 0$;

    Accept $\leftarrow$ True ;

  else

    Generate two independent and identically distributed uniform random numbers V and W of $\mathrm{U}[0, 1]$;

    $X \leftarrow \left\lfloor \frac{1}{W^2} \right\rfloor$;

    Accept $\leftarrow \left[V\, b \left(\frac{1}{\sqrt{X}} - \frac{1}{\sqrt{X+1}} \right) \leq \frac{\mu(\lambda X+\mu)^{X-1} e^{-(\lambda X+\mu)}}{X!} \right]$;

  end

until *Accept*;

return X ;

relationships to other distributions.

Particularly, random variates of the beta binomial, the Yule, the Waring and the beta-negative binomial distributions could not be directly generated by tailored algorithms. The reason is that the PDFs of these distributions are complex and they also have different shape parameters which change the shape of the PDFs drastically, preventing a suitable dominating function in the acceptance-rejection method to be obtained. Therefore, we used other statistical properties of these distributions to generate their random variates. These properties let us generate the desired random variates, by appropriately combining the random variates of simpler distributions. In that direction, we introduced different elementary distributions as well as the appropriate algorithms for generating their random variates[24]. Finally, we combined the results in order to produce the random variates of these four distributions.

[24] In the case that any of these distributions is already implemented, in practice, one can directly plug it to other algorithms which use it.

5.4 Related Works

The presented results of this chapter focus on mathematically modeling the "evolution" of design models in terms of the applied edit operations between their subsequent revisions. The edit operations were defined at two abstraction levels. The first level was low-level graph edit operations and the second one was high-level (developer-friendly) edit operations. The measurements of frequencies of applied low-level edit operations lead to low-level difference metrics which are shown to be more appropriate than the static metrics counterparts in the context of model differencing, model versioning and model generation (see [Wenzel, 2010] and Chapter 4).

To best of our knowledge there is no similar work which addresses mathematical properties of difference metrics in practice and we could not find any research work which directly considers, how measured difference metrics on histories of real software systems behave and which properties they show. However, there are numerous metrics defined for software systems and obviously there are many applications for them, e.g. there is extensive amount of research on fault prediction, estimation of the development effort, cost estimation, quality estimation and prediction, maintenance, component fault classification, etc. using metrics [Kitchenham, 2010, Lanza and Marinescu, 2006, Fenton and Bieman, 2014]. The horizons even extensively broaden in combination of mining software repositories approaches which more or less employ metrics in their works, e.g. finding bugs, coupling components, software reuse, communication with respect to the development process, etc. [Kagdi et al., 2007].

In this section our focus will be narrowed to reviewing those works which can be considered more relevant to what we did in this chapter, i.e. mathematically modeling the evolution of design models in terms of low-level and high-level edit operations. To this aim, we categorized the most relevant works in two major categories. The first one considers the evolution of software systems motivated from the Lehman's laws of software evolution [Lehman, 1980]. Lehman tried to formulate how software systems evolve and what are their characteristics. The second category consists of the works which consider the analysis of different metrics or different features of software systems with respect to their mathematical properties. The proposed mathematical models were then used in different directions, e.g. understanding software systems and their properties, estimating development efforts, designing new languages or related virtual machines, identifying fault prone components, etc. In each category, we review the most relevant works without going into details of how they are used, since it would be out of our research scope.

Software Evolution The majority of the research done in this category is motivated from the pioneering work of Lehman for formulating the evolution of software systems [Lehman, 1980]. The laws of software evolution were themselves evolved over time [Lehman, 1996, Lehman et al., 1997] and many researchers were interested to evaluate different aspects of the laws in practice. Although our motivation is quite different, we review few works in this area which can indirectly be considered most related to our work. This area is a broad topic of interest on its own and the interested reader can find

more basic and interesting information in [Fernandez-Ramil et al., 2008, Herraiz, 2008, Vasa, 2010, Herraiz et al., 2013].

[Vasa et al., 2007a] studied the evolution of classes and interfaces in open-source Java systems based on static metrics of their released versions. 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. Between versions, identity of classes were established using their fully qualified name, therefore the renamed or moved classed were missed. The amount of change was measured based on value changes of 25 static metrics. They showed that more complex classes, i.e. the ones with more branch counts in their code, and more popular classes, i.e. the ones with more incoming and outgoing dependencies, are more likely to undergo changes. This research was continued in [Vasa et al., 2007b] and they consider additional static metrics. There, they show that the majority of changes happen on a small portion of all classes. They also analyze the history of classes based on a comparison between the static metric values counted in the final version of the system and those counted in preceding versions. In this regard, they measured the proportion of classes in the final version which were either changed or not modified since they were created. This was done through measuring the difference between the values of static metrics were identifiable in the final version to the earlier ones. The results show that one third of the classes are not changed since they are created. Moreover, just small amount of classes either change multiple times or go under substantial modification. In this direction, more detailed results are presented in [Vasa, 2010].

In the aforementioned works, neither any advanced statistics nor our proposed statistical models were considered. Moreover, the evolution was based on static metrics and not on difference metrics which are essential for generating models. As discussed in Section 4.1, using the static metrics has the drawback that it might be misleading. For instance if one measures the number of attributes in a class to be the same between two revisions, it does not convey that there is no change on attributes. Rather, the underlying amount of change might be considerable, e.g. 5 attributes are added, 5 are deleted and 5 might be renamed, resulting 15 changes in total which are not reflected by difference of zero between static metrics of *NOM-Attributes* between two revisions. Additionally, those works consider the versions of software systems while in this dissertation finer steps, i.e. revisions are considered. The changes were also computed between revisions rather between the final version and earlier ones.

In [Herraiz et al., 2007a], the authors considered many source codes files of FreeBSD, a Unix-like operating system, which were developed mostly in C, C++, Java, Perl and Python programming languages. They investigated different size and complexity metrics. Their analysis revealed that the Pareto distribution can model the measured metrics. The same distribution was showed to adequately model the source code file sizes using a sample set of many available software systems in the version 5.0.2 of the Debian Linux distribution [Herraiz et al., 2011]. The validation of the Lehman's laws of software evolution was the main topic of interest in [Herraiz, 2009, 2008]. There, the author considered many software systems in FreeBSD as well as many other systems available

at SourceForge.net and he computed different metrics on them. The results show that SLOC metric (Source Lines of Code) is a good indicator of software growth and size of files in SLOC and size of software project in terms of the number of its files, follow the Pareto distribution. Generally, it is concluded that there are cases which invalidate the proposed laws of software evolution.

Analysis of Metrics or Properties of Software Systems The reviewed publications in this category focus on mathematically modeling the static metrics or other features of software systems. The following works only use single system snapshots as the base for their analysis and the topic of system evolution, i.e. changes between versions or revisions, is not addressed at all.

[Wheeldon and Counsell, 2003] analyzed power law distributions for different forms of object-oriented couplings in the graph structure on three Java software systems; JDK (Java Development Kit), Apache Ant and Tomcat. The considered couplings were inheritance, aggregation (e.g. containment of fields, etc. by a class), interface implementation, parameter and return types. They extracted graph structures from the source code and computed the occurrences of the couplings. They identified that the coupling-related metrics obey the power law.

[Marchesi et al., 2004] considered the graph representation of four large SmallTalk projects which was principally a class-relationship representation of the systems. Similarly to [Wheeldon and Counsell, 2003], in such presentation the nodes are the elements of the systems and the edges show their interactions. They took classes as nodes and they considered two interaction of inheritance and dependency between them. They showed that class-relationships obey the power law.

[Potanin et al., 2005] studied the object graph of Java software systems. Using a tool, they took 60 snapshots of 36 running programs and created object graph. Object graphs are principally graphs where nodes are objects and edges represent their communicating relationships. The number of incoming and outgoing references showed to follow the power law in object graphs.

[Baxter et al., 2006] measured 17 static metrics on a sample set of Java systems. They considered the suitability of the power law as well as the log-normal and the exponential distributions. The analysis showed that the power law is more suitable than the others, although there are also cases that metrics do not obey the power law.

[Collberg et al., 2007] have studied static and structural metrics of a collection of Java jar-files for the aim of improving the design of future programming languages, compilers or virtual machines. Their detailed results show that the distribution of some the measured metrics are skewed with heavy tails. Except simple basic statistics, no advanced statistical models were reported in this work.

In [Concas et al., 2006, 2007], the authors studied different metrics related to the classes and methods of VisualWorks Smalltalk and compared them to those observed ones in Eclipse and JDK (Java Development Kit). All of the measured distributions showed heavy tails. They showed that majority of metrics obey the power law and the Pareto distribution. Some of the metrics reported to obey the log-normal distribution.

[Ichii et al., 2008] studied software component graphs, which essentially describe building units of a software and their relationships. The nodes in such graphs represent components, i.e. classes or interfaces, and edges represent use-relationships between components. Using different Java software systems as the sample set, they showed the distribution of in-degrees follows power law, but the distribution of out-degrees does not. They speculated that the out-degrees might follow the log-normal distribution.

[Louridas et al., 2008] studied the power law on dependency graphs of different software systems. The dependency graph between classes of Java systems, Perl packages, shared libraries in open source Linux distributions, dynamic link libraries of Microsoft Windows, Ruby library files and TEX modules were considered for the analysis. The results show that the power law is more pervasive in software systems than it was initially investigated by earlier works that just considered sample projects from a specific domain.

[Herraiz Tabernero et al., 2012] studied object-oriented static metrics on the most recent releases of a large collection of Java systems. They considered the suitability of the power law and the log-normal and the Pareto distributions on the metrics. Their results showed that some of the metrics could be modeled using the power law while others could be modeled by the Pareto distribution.

[Pani and Concas, 2012] considered five object-oriented metrics on five subsequent versions of Eclipse. They tested the suitability of the power law as well as the log-normal and the Pareto distributions. Roughly speaking, each metric could be fitted by at least one of those models.

[Shatnawi and Althebyan, 2013] considered eight static metrics and they investigated the presence of the power law on five Java systems. Five of the measured metrics showed heavy tails and the power law was hold on them.

The previous research works just considered (definite) single versions of systems, i.e. an snapshot of their histories, for the analysis and the history of systems were not considered. Contrary, the following research works consider different versions of systems and the history of the systems are considered. This issue should not be confused by analyzing the changes between histories. Although histories are considered, the presence of the power law or other mathematical models are just tested on each revision separately and the changes between versions, i.e. evolution, are not considered.

[Tamai and Nakatani, 1998, Tamai, 2002, Tamai and Nakatani, 2002] studied different object-oriented metrics on three software systems as the sample set. A heat exchange simulation system, a cash receipts transaction management system and a security management system were considered. All of those systems were written in SmallTalk and had 4, 4 and 14 versions respectively. They considered overall properties of systems as well as class and methods metrics which were calculated on versions. The Poisson, geometric and negative binomial distributions were considered to model the data and the results showed that just the negative binomial distribution was appropriate.

[Turnu et al., 2011] studied properties of four Java software systems. Several versions of Eclipse (12 versions), Netbeans (17 versions), JDK (Java Development Kit) (11 versions) and Java Ant (10 versions) were considered. Four properties regarding variables

and methods of classes were measured on them. They observed that the properties obey the power law. Moreover, they showed that the properties can also be modeled using the Yule process. Although they considered different versions of systems, their analysis were done solely on each version and the results were reported to be valid on individual revisions.

Summary As discussed, in majority of the aforementioned works the evolution was not considered. They worked on specific versions or snapshots of the systems instead of considering the changes between revisions or versions. Just in a few works, e.g. [Vasa et al., 2007b, Vasa, 2010], histories of model versions were considered. In contrast, we analyzed the revisions of systems in this dissertation. In comparison to versions, revisions are often quite finer steps in the development process. Therefore, the changes between revisions are better representatives for the evolution than the changes between versions.

Comparing the values of static metrics does not reflect the real amount of underlying changes especially in the domain of models and the model-driven engineering. Therefore, difference metrics were used in this dissertation and the evolution was measured in terms of these metrics rather using the static metrics. Moreover, the employed metrics by this dissertation are at two abstraction levels. The first level is based on low-level graph edit operations and the second level is based on high-level (developer-friendly) edit operations. For the first level we had 75, and for the second one we used 188 metrics. These metrics are much more detailed compared to static metrics employed by other research works.

In this dissertation, the changes were measured between subsequent revisions instead of using the final version as the base of comparison for all other versions, as done in [Vasa, 2010]. Such a comparison has much little benefit since much of the information is disregarded.

Similarly to what we found out in this research, many software metrics and properties show skewed distributions with heavy tails. We showed the presence of the power law on majority of difference metrics. We investigated the suitability of many statistical models on our difference metrics and we found out that the log-normal or the (continuous) Pareto distributions are not suitable in this case, instead the discrete Pareto distribution is more appropriate. We additionally found out that the Yule, the Waring, the beta-negative binomial, the beta binomial and the generalized Poisson distributions can be used to model the evolution with very good success rates. None of these distributions were used or reported to be used in the related literature. Therefore, it might be the case that such models are better than the existing ones in describing the properties of software systems, albeit more investigation is required.

5.5 Summary

In this chapter we addressed the question of how design models of software systems change and how we can mathematically model their differences. In this regard, the changes of design models were captured at two abstraction levels. The first level ex-

pressed the changes in terms of 75 low-level difference metrics. The second level captured the changes in terms of 188 high-level metrics.

We learned that the histogram of the calculated frequencies of both, the low-level and the high-level difference metrics are usually skewed with heavy tails. Statistical modeling of such changes was the first challenge. We tested the fitness of 60 distributions on all measured changes. Just six of those distributions were able to handle our data acceptably, although each with a different success rate. We proposed the detailed analysis of the changes (both low-level and high-level) using those six distributions. We showed where each distribution was successful, where it failed and how it can be used to statistically model the real evolutions. The negative binomial, the Waring, the beta-negative binomial and the generalized Poisson distributions are the most successful ones. We found out that they can describe almost any type of low-level and high-level changes with good success rates, making them the best fitting statistical models.

The other important question addressed in this chapter is how these statistical models can be used in practice to simulate real model evolution. The related literature did not offer algorithms for generating random variates of the negative binomial, the Waring and the beta-negative binomial distributions. We proposed an indirect method for generating their random variates based on mathematical properties of these distributions. In this regard, we also discussed the possible difficulties and the technical issues in detail.

CHAPTER 6

Time Series Analysis and Simulation of Design Models Evolution

As we discussed in Chapter 1, one contribution of this work is to mathematically model and simulate the evolution of design models for generating more realistic model histories. In this chapter we mathematically model and simulate the historical evolution of design models using time series. Time series theory deals with situations in which one has sequential measurements of interest through time. As we observed in Chapter 4, the evolution of design models was computed based on the measured occurrences of low-level and high-level edit operations between revisions of software design models. Such sequential observations of the applied edit operations between revisions can be principally modeled using time series.

As we will see later (Section 6.2), the measured evolution of our Java software systems shows a very erratic behavior which is a hurdle to easily employ time series models. Moreover, we will observe that the evolutions show different characteristics and they require different remedies. In order to handle the problem, it is quite essential to have a suitable rigid mathematical infrastructure which can be employed appropriately. In this regard, the required mathematical infrastructure of our analysis in is provided in Section 6.1. In Section 6.1.3, ARMA, GARCH and mixed ARMA-GARCH models are introduced in detail. We show how and why such models work in Section 6.1.2. Accordingly, the methodology of these time series models as well as their forecasting accuracies are discussed in Sections 6.1.4 and 6.1.5.

Section 6.2.1 discusses characteristics of the measured evolution and how the evolution can be mathematically transformed into an appropriate form for time series analysis. The transformed series were then modeled using ARMA and mixed ARMA-GARCH models in Sections 6.2.2 and 6.2.3.

The assessment of the proposed time series models is discussed in Section 6.3. To this end, a comparative study between these two kinds of time series models is presented in Section 6.3.1. The forecasting performance of these competitive models are also discussed in Section 6.3.2.

Section 6.4 is devoted to the simulation of the modeled evolution. General consideration regarding the simulation of time series models are discussed in Section 6.4.1. Section 6.4.2 describes how sequences of the proposed time series models are simulated while Section 6.4.3 discusses how the simulated sequences can be used to generate more realistic histories of design models.

The threats to the validity of our analyses are discussed in detail in Section 6.5. There we investigate different aspects, including accuracy in the measurement of changes, selection strategy for best time series models, forecasting performance as well as simulation. Related works are discussed in Section 6.5. The chapter ends with a summary in Section 6.7.

6.1 Time Series

In this section we provide the detailed theoretical background which is required by our time series analysis. There are many books which address time series, but each of them has its own perspective to the issue and each one expresses the required theory with a different level of abstraction and application focus. This makes it quite difficult to have a smooth and coherent theory which can be used to answer our research questions and to justify and prove the results. In what follows we provide the detailed theoretical background of our research. When needed, we refer the reader to the appropriate sources ordered by their importance of the discussed issues.

In the first step, here and in Section 6.1.1, we introduce the time series models. Section 6.1.2 discusses the general linear process and the Wold decomposition theorem which are the fundaments of the time series models and principally explains how and why such time series models work. Section 6.1.3 introduces ARMA, GARCH and mixed ARMA-GARCH models which we later need for our evolution analysis. Methodology of these time series models are discussed in detail in Section 6.1.4. Forecast accuracy of our time series models are discussed in Section 6.1.5.

Time series are sequential measurements or observations of a phenomenon of interest through time. A time series is usually denoted by $\{X_t, t \in T\}$ where T is the index set [Box et al., 2008]. The time dependency of the data gives time series a natural ordering which is used to study them.

If the measurements are done continuously in time, they are referred to as continuous time series, otherwise they are called discrete time series. In this work the latter case applies. The series is usually denoted by $x = x_{t_1}, x_{t_2}, \ldots, x_{t_N}$ when we have N successive observations at times $t_1, t_2, \ldots, t_N$. Please note that $\nabla t_i = t_i - t_{i-1}$ is not necessarily constant, although in some applications it is assumed so.

In order to extract useful information and statistics out of the data and to discover its underlying dynamics, time series methods are employed. These methods are also used in

order to forecast the future of the system. In time series methods it is usually assumed that the data obey some underlying parametric stochastic processes (i.e. a sequence of random variables) and the parameters are estimated in order to describe the behavior of the system. Principally it is assumed that an observed time series of $x_1, x_2, \ldots, x_N$ is a sample realization from an infinite population of such time series that could have been generated by the stochastic process $\{X_1, X_2, \ldots, X_k, \ldots\}$ [Box et al., 2008, Brockwell and Davis, 2006]. This property is used in order to create many time series realizations all with the same stochastic behavior (see Section 6.4).

6.1.1 Stationary Time Series

In the study of time series it is assumed that some statistical properties of the series are not changing with time, otherwise no statistical inference about the data is possible [Kirchgässner and Wolters, 2007]. In particular it is generally assumed that the underlying stochastic process is *stationary*, i.e. the process is in a statistical equilibrium and the main statistical features of the series does not change with time. This allows us to systematically capture those features and to mathematically model the series [Rachev et al., 2007]. Two classes of stationarity are defined, strictly stationary and weakly stationary.

Roughly speaking, in a strictly stationary stochastic process all statistical properties are not affected by shifts in time, i.e. a change of the time origin, and are only dependent on time differences. Formally, if the joint probability distributions of the observations $x_{t_1},\ldots,x_{t_n}$ and $x_{t_{1+h}}, \ldots, x_{t_{n+h}}$ are the same for all integers h and $n > 0$ and for all indices of $\{t_1, t_2, \ldots, t_n\}$, we call the process *strictly stationary*; i.e. $(x_{t_1}, \ldots, x_{t_n}) \stackrel{d}{=} \left(x_{t_{1+h}}, \ldots, x_{t_{n+h}}\right)$ [Box et al., 2008, Brockwell and Davis, 2002, Kirchgässner and Wolters, 2007, Montgomery et al., 2008].

The strictly stationary condition is too tight for practical applications and therefore the second class, weakly stationary, is used in practice. In weakly stationary processes we focus on the properties of first and second moments, i.e. mean and variance of data [Box et al., 2008]. The process is weakly stationary when there is no change in the trend of the data and there is also no change in the variance of the data through time, so if one plots them, data are fluctuating around a fixed level with a constant variation [Tsay, 2005].

A process is *mean stationary* if $\mathrm{E}[X_t]$ is constant for all t, i.e. $\mathrm{E}[X_t] = \mu$. Similarly, a process is called *variance stationary* if $\mathrm{Var}[X_t] = \sigma^2$ is constant and finite for all t. A *covariance stationary* process is a process in which $\mathrm{Cov}[X_t, X_s]$ is only dependent on the distance between t and s, i.e. dependent on $|s - t|$, and not on the actual points. The covariance stationarity results in the variance stationarity as the special case where $t = s$. A stochastic process is *weakly stationary*[1] when it is mean and covariance stationary [Kirchgässner and Wolters, 2007]. The weakly stationarity is only concerned with the unconditional mean and unconditional variance to be constant, but the conditional moments can vary with time which can be modeled and forecast.

[1] Some times is referred to as *second-order stationary* [Francq and Zakoian, 2010].

A *white noise* process, is an uncorrelated sequence of random variables $\{\varepsilon_t\}$ with mean 0 and the constant variance σ^2. $\{\varepsilon_t\}$ is weakly stationary and is denoted by $\{\varepsilon_t\} \sim \mathrm{WN}\left(0, \sigma^2\right)$ [Brockwell and Davis, 2002, Kirchgässner and Wolters, 2007]. Additionally when $\{\varepsilon_t\}$ is normally distributed it is called a *Gaussian white noise* [Montgomery et al., 2008].

Depending on the shape of the data, there are some methods for transforming non-stationary processes to stationary ones in order to stabilize their mean and their variance, e.g. trend elimination methods, differencing, cyclic/seasonal differencing, logarithm and power transformations, etc.

Differencing means that we use the series of $\{\nabla x_t\}$ instead of $\{x_t\}$ where $\nabla x_t = x_t - x_{t-1}$. $\{\nabla x_t\}$ usually has more stable moments. The *backshift operator* is defined by $\mathrm{B}\, x_t = x_{t-1}$. Thus $\nabla x_t = x_t - x_{t-1} = (1 - \mathrm{B}) x_t$ [Brockwell and Davis, 2002]. Powers of B and ∇ are defined similarly: $\mathrm{B}^k\, x_t = x_{t-k}$ and $\nabla^k x_t = \nabla(\nabla^{k-1} x_t)$ when $k \geq 1$ and $\mathrm{B}^0\, x_t = x_t$, $\nabla^0 x_t = x_t$. With this notation, polynomials of B and ∇ are treated like ordinary polynomials, e.g. $\nabla^2 x_t = \nabla(\nabla x_t) = (1 - \mathrm{B})(1 - \mathrm{B}) x_t = (1 - 2\,\mathrm{B} + \mathrm{B}^2) x_t = x_t - 2x_{t-1} + x_{t-2}$.

6.1.2 General Linear Process and the Wold Decomposition Theorem

Before going into the detail of time series models, we briefly provide the underlying theory which helps to grasp the concepts and materials which are provided in the following sections. This section is devoted to the groundbreaking Wold's theorem and the concept of causality and invertibility in time series, which are essential to find proper solutions for the weakly stationary time series.

A time series $\{x_t\}$ is said to be a *linear process* if for all t, it has the representation of [Brockwell and Davis, 2002]:

$$x_t = \sum_{j=-\infty}^{\infty} \psi_j \varepsilon_{t-j} \tag{6.1}$$

where $\{\varepsilon_t\} \sim \mathrm{WN}\left(0, \sigma^2\right)$ and $\{\psi_j\}$ is a sequence of constants with $\sum_{j=-\infty}^{\infty} |\psi_j| < \infty$, i.e. the coefficients are absolutely summable[2]. This ensures the convergence of the infinite sum of (6.1) [Brockwell and Davis, 2002].

A linear process is called a *moving average* or $\mathrm{MA}(\infty)$ if $\psi_j = 0$ for all $j < 0$, i.e. if we have $x_t = \sum_{j=0}^{\infty} \psi_j \varepsilon_{t-j}$ [Brockwell and Davis, 2002]. Moreover, when a process is defined based on the past values of ε_j's and is independent of the future values, we call it a causal process. Formally, we call the process $\{x_t\}$ *causal* or *future-independent*, if there exists a sequence of $\{\psi_j\}$ such that $x_t = \sum_{j=0}^{\infty} \psi_j \varepsilon_{t-j}$ with $\sum_{j=0}^{\infty} |\psi_j| < \infty$ [Chan, 2010]. In other words a process is causal if it has a moving average $\mathrm{MA}(\infty)$ presentation. The causal property is required for the prediction of the future of stationary processes, since the future will be just dependent to the past.

[2] It causes $\sum_{j=0}^{\infty} \psi_j^2 < \infty$, i.e. the absolute summability results in the square summability.

Using the backshift operator of B, (6.1) can be written more compactly as $x_t = \Psi(\mathrm{B})\,\varepsilon_t$, where $\Psi(\mathrm{B})$ is a series in terms of B and is defined by $\Psi(\mathrm{B}) = \sum_{j=-\infty}^{\infty} \psi_j\,\mathrm{B}^j$. The $\Psi(\mathrm{B})$ is also referred to as a *linear filter*, since it is applied on the "input" series of white noise in order to produce the "output" series of $\{x_t\}$. It is shown that the application of a linear filter to a stationary series delivers again a stationary output series [Brockwell and Davis, 2002]. Furthermore, in the case of applying two linear filters to a series, the application order is immaterial for the resulting series.

The groundbreaking Wold's decomposition theorem states that any nondeterministic stationary process $\{x_t\}$ has a linear process representation of [Brockwell and Davis, 2002, Box et al., 2008]:

$$x_t = v_t + \sum_{j=0}^{\infty} \psi_j \varepsilon_{t-j} \tag{6.2}$$

where $\psi_0 = 1$, $\sum_{j=0}^{\infty} \psi_j^2 < \infty$, $\{\varepsilon_t\} \sim \mathrm{WN}\left(0, \sigma^2\right)$, $\{v_t\}$ is deterministic[3] and $\mathrm{Cov}\left[x_s, v_t\right] = 0$ for all s and t. If $\{x_t\}$ is zero-mean stationary, then $v_t = 0$ for all t otherwise $v_t = \mu$ where μ is the mean of the process. Assuming that $\Psi(\mathrm{B}) = 1 + \sum_{j=1}^{\infty} \psi_j\,\mathrm{B}^j$, (6.2) can be written more compactly as $\tilde{x}_t = \Psi(\mathrm{B})\,\varepsilon_t$.

Let $\{\tilde{x}_t\}$ be the series of deviations of x_t from its mean μ, i.e. $\tilde{x}_t = x_t - \mu$, in other words, the Wold's theorem states that any zero-mean nondeterministic stationary process has an infinite moving average MA (∞) representation, i.e. it can be written as an infinite linear combination of white noise shocks. In order to find the Wold's representation, we have to fit infinite parameters of ψ_j's in (6.2) which is impossible, even when $\{x_t\}$ is finite [Hamilton, 1994]. In practice, the infinite series of (6.2) is truncated after q terms to get an approximation of the Wold's representation [Francq and Zakoian, 2010]. The truncated series is called a moving average process of order q and is denoted by MA (q). It is shown that the set of all moving average processes with finite orders are dense in the set of nondeterministic weakly stationary processes [Francq and Zakoian, 2010]. As we will see in Section 6.1.3.1, another typical and even more parsimonious approach is to consider the associated (backshift) series of $\sum_{j=0}^{\infty} \psi_j\,\mathrm{B}^j$, as the ratio of two finite order polynomials [Hamilton, 1994]:

$$\sum_{j=0}^{\infty} \psi_j\,\mathrm{B}^j = \frac{\Theta(\mathrm{B})}{\Phi(\mathrm{B})} = \frac{1 + \theta_1\,\mathrm{B} + \theta_2\,\mathrm{B}^2 + \cdots + \theta_q\,\mathrm{B}^q}{1 - \phi_1\,\mathrm{B} - \phi_2\,\mathrm{B}^2 - \cdots - \phi_p\,\mathrm{B}^p}$$

We should mention that under appropriate conditions [Box et al., 2008], $\tilde{x}_t$ can be written as a weighted sum of the past deviations of $\tilde{x}_j$'s plus a shock of ε_t, that is:

$$\tilde{x}_t = \varepsilon_t + \sum_{j=1}^{\infty} \pi_j \tilde{x}_{t-j} \tag{6.3}$$

Now let $\Psi(\mathrm{B}) = 1 + \sum_{j=1}^{\infty} \psi_j\,\mathrm{B}^j$ and $\Pi(\mathrm{B}) = 1 - \sum_{j=1}^{\infty} \pi_j\,\mathrm{B}^j$, therefore (6.2),(6.3) can be written in a more compact form as $\tilde{x}_t = \Psi(\mathrm{B})\,\varepsilon_t$ and $\Pi(\mathrm{B})\,\tilde{x}_t = \varepsilon_t$ respectively. It

[3] Here deterministic means that the future of the process is completely predictable from its past [Shumway and Stoffer, 2011].

Property	ARMA	GARCH	ARMA-GARCH
Conditional Mean	non-const.	const.	non-const.
Conditional Variance	const.	non-const.	non-const.

Table 6.1: Properties of time series models.

can be shown that we have $\Psi(\mathrm{B})\,\Pi(\mathrm{B}) = 1$ or equivalently we have $\Pi(\mathrm{B}) = \Psi(\mathrm{B})^{-1}$. This allows us to derive the coefficients of π's when we know the ψ's and vice versa. Therefore, having the history of each one, the history of the other one can be inferred. Finally, we call the linear process $\{x_t\}$ *invertible* when ε_t's can be expressed as linear combination of $\tilde{x}_t$'s, i.e. $\Pi(\mathrm{B})\,\tilde{x}_t = \varepsilon_t$.

To sum up this section, the linear process $\{x_t\}$ is stationary and causal if we have $\sum_{j=0}^{\infty} |\psi_j| < \infty$ and is invertible if $\sum_{j=0}^{\infty} |\pi_j| < \infty$. The Wold's theorem in addition to the aforementioned conditions lets us find an appropriate parsimonious representation (i.e. ARMA models) for weakly stationary time series (see the next section).

6.1.3 ARMA, GARCH and ARMA-GARCH Models

For studying the weakly stationary series, two classes of models, i.e. autoregressive moving average (ARMA) and autoregressive conditional heteroscedasticity (ARCH) models, are of special interest and have been extensively employed in practice. Before going into details of each class of models and their subfamilies, it is informative to consider their general properties.

Let $\mathcal{F}_t$ be the set of all of information up to time t. ARMA time series models are used to handle and model the conditional expectation, $\mathrm{E}[x_t \mid \mathcal{F}_{t-1}]$, of the series. In ARMA models, it is assumed that the underlying process is *homoscedastic* i.e. its conditional variance of $\mathrm{Var}[x_t \mid \mathcal{F}_{t-1}]$ is a constant and does not change with time. Such assumption does not hold in some application domains specially in econometrics time series where the conditional variance is volatile [Matteson and Ruppert, 2011, Cont, 2001]. When $\mathrm{Var}[x_t \mid \mathcal{F}_{t-1}]$ is not constant we call the series *heteroscedastic*. ARCH models are introduced in order to model the heteroscedasticity effect in data. ARCH models are later elaborated into generalized ARCH models (GARCH) which are more parsimonious. For ARCH and GARCH models, the conditional expectation is considered to be constant.

There are also frequent cases where both conditional mean and conditional variance of data are not constant. Mixed ARMA-GARCH models are used to model this characteristic in data. Table 6.1 depicts the previous discussion is a more compact manner [Jasiak]. Having the previous consideration in mind, we take a deeper look at ARMA, GARCH and mixed ARMA-GARCH models in the following sections.

6.1.3.1 ARMA and ARIMA Models

One of the most successful models that is used to analyze and stochastically model weakly stationary processes is the *autoregressive moving average* model (ARMA) [Box et al., 2008]. ARMA models are used to model the conditional expectation of time series. The unconditional mean and the unconditional variance are constant in these models. In such models, the dynamic of the system is assumed to be dependent on two factors: previous states of the process and random disturbances in the past.

Let $\{x_t\}$ be a weakly stationary time series with mean of μ and let $\{\tilde{x}_t\}$ be the corresponding mean-adjusted series i.e. $\tilde{x}_t = x_t - \mu$. Using ARMA (p, q), with p degree of dependency to the past observations and q degree of dependency to the past disturbances, the current state of the model, i.e. $\tilde{x}_t$, is defined by:

$$\tilde{x}_t = \sum_{i=1}^{p} \phi_i \, \tilde{x}_{t-i} + \varepsilon_t + \sum_{i=1}^{q} \theta_i \, \varepsilon_{t-i} \tag{6.4}$$

where $\tilde{x}_{t-i};\ i = 1, \ldots, p$ are the past observations, $\varepsilon_{t-i};\ i = 1, \ldots, q$ are the past disturbances, ε_t is the current disturbance; $\{\varepsilon_t\} \sim \mathrm{WN}\left(0, \sigma^2\right)$ and in practice, it is assumed to have the standard normal distribution. By definition, we should also have $\phi_p \neq 0$ and $\theta_q \neq 0$. Equation (6.4) has $p+q+2$ unknowns that should be estimated from data: ϕ_i's, θ_i's, μ and σ_ε^2 [Box et al., 2008]. When $p = 0$, we have MA (q) which is a purely *moving average* model; similarly when $q = 0$ we get AR (p), a purely *autoregressive* model. Using the backshift operator, (6.4) can be written as [Bisgaard and Kulachi, 2011, Tsay, 2005]:

$$\Phi_p(\mathrm{B}) \, \tilde{x}_t = \Theta_q(\mathrm{B}) \, \varepsilon_t \tag{6.5}$$

in which $\Phi_p(\mathrm{B}) = (1 - \phi_1 \, \mathrm{B} - \cdots - \phi_p \, \mathrm{B}^p)$ is the *autoregressive operator*, a polynomial of order p, and $\Theta_q(\mathrm{B}) = (1 + \theta_1 \, \mathrm{B} + \cdots + \theta_q \, \mathrm{B}^q)$ is the *moving average operator*, a polynomial of order q. As we discussed in Section 6.1.2, the ARMA representation of (6.5) is a parsimonious approximation of Wold's theorem given by (6.2).

We now discuss under which condition a suitable weakly stationary solution for (6.5) exists. When one considers ARMA models of the form (6.5) three problems arise [Shumway and Stoffer, 2011]: *First:* the models might have redundant parameters. *Second:* stationary AR models might not be causal, i.e. they might be future dependent. *Third:* MA models are not unique in such formulation, but their correlation structures are indistinguishable, thus one cannot clearly prefer one to the others. To overcome these problems, the weakly stationary solutions of (6.5) must be causal, invertible[4] (see Section 6.1.2) and with no common factors in AR and MA polynomials [Brockwell and Davis, 2002, Shumway and Stoffer, 2011].

It is proven that (6.5) has a unique stationary solution if and only if $\Phi_p\left(\mathrm{B}\right) \neq 0$ for all $|\mathrm{B}| = 1$. The solution of (6.5) is causal if $\Phi_p\left(\mathrm{B}\right) \neq 0$ for all $|\mathrm{B}| \leq 1$. Similarly, the solution of (6.5) is invertible if $\Theta_q\left(\mathrm{B}\right) \neq 0$ for all $|\mathrm{B}| \leq 1$. For the formal proofs of the

[4] Here invertibility can be considered equivalently as an MA (q) model can be represented as a AR (∞) by inverting its moving average operator.

stated theorems we refer the interested readers to [Brockwell and Davis, 2006]. To sum up our discussion, (6.5) has a unique causal invertible stationary solution if the roots of $\Phi_p(\mathrm{B})$ and $\Theta_q(\mathrm{B})$, all lie outside of the unit circle and they have no common factor.

Lastly, as discussed in Section 6.1.1, differencing[5] can be used to make a series stationary. If d times of differencing[6] is used to make $\{x_t\}$ stationary, since $\nabla^d \tilde{x}_t = \nabla^d x_t$ for $d \geq 1$, then $\nabla^d x_t$ can be used instead of $\tilde{x}_t$ in (6.4),(6.5) and we will have an *autoregressive integrated moving average* model, denoted by ARIMA (p, d, q), which is defined by [Box et al., 2008, Montgomery et al., 2008, Bisgaard and Kulachi, 2011]:

$$\Phi_p(\mathrm{B})\,(1-\mathrm{B})^d x_t = \Theta_q(\mathrm{B})\,\varepsilon_t \tag{6.6}$$

6.1.3.2 ARCH and GARCH Models

Before we introduce ARCH and GARCH models, it will be explanatory to consider the background of these models which roots in the financial econometrics. There, the study of values of assets over time is the main focus of financial time series analysis [Tsay, 2005]. For the analysis of values of assets, their returns are usually used instead of their prices. The *log return*, or simply the "return"[7], of an asset is defined as $r_t = \ln\left(\frac{P_t}{P_{t-1}}\right) = \ln P_t - \ln P_{t-1}$ where P_t is the price of an asset in time t. There are two reasons for using series of returns instead of series of the prices in analysis of financial markets [Campbell et al., 1997]: *a*) they are scale-free, which make them useful for investment decisions in competitive markets and *b*) they have more attractive statistical properties from theoretical and empirical points of view .

The analysis of returns has revealed statistical properties which are referred to as *stylized facts* [Cont, 2001, Bollerslev et al., 1994]. Volatility clustering, absence of autocorrelation, slow decay of the autocorrelation function of absolute returns and heavy tails of (unconditional) distribution of returns are the most important stylized facts.

The *autoregressive conditional heteroscedasticity* (ARCH) models are introduced by Engle [Engle, 1982] in order to deal with the stylized facts and also more importantly to capture the heteroscedasticity effect observable in financial time series. Such models are serially uncorrelated, have constant unconditional mean (more accurately the mean is zero) and constant unconditional variance but a non-constant conditional variance. Formally, let again $\{x_t\}$ be a time series[8], an ARCH (q) model, tries to model conditional

[5] There is also seasonal differencing for eliminating seasonal effects in data [Brockwell and Davis, 2002, Montgomery et al., 2008]. The seasonal differencing is not applicable to our data (see Section 6.2.1) and we do not discuss it here.

[6] The value of d is usually 0, 1 or at most 2 in practice [Box et al., 2008].

[7] There are also other less common definitions for *return*, each has applications in finance [Tsay, 2005], e.g. simple return is defined as: $R_t = \frac{P_t - P_{t-1}}{P_{t-1}} = \frac{P_t}{P_{t-1}} - 1$.

[8] $\{x_t\}$ is considered to be a mean-adjusted series in practice, i.e. its mean μ is subtracted from the series, since ARCH models are proved to have mean of zero.

variance of the series as a linear combination of q previous squared values of the series:

$$\mathrm{Var}\left[x_t \mid \mathcal{F}_{t-1}\right] = \sigma_t^2 = \omega + \sum_{i=1}^{q} \alpha_i x_{t-i}^2 \tag{6.7}$$

and the series $\{x_t\}$ is then modeled as:

$$x_t = \sigma_t \, \varepsilon_t \tag{6.8}$$

in which $\varepsilon_t \sim \mathrm{WN}\left(0, \sigma^2\right)$, $\omega > 0$ and $\alpha_i \geq 0 \, ; i = 1, \ldots, q$. In practice, ε_t is considered to be of a standard normal distribution or a standardized Student's t-distribution [Tsay, 2013].

The aforementioned conditions on ω and α_i's guarantees the positiveness of the conditional variance. The $\mathrm{ARCH}\,(q)$ model with $\omega > 0$ and $\alpha_i \geq 0 \, ; i = 1, \ldots, q$ is covariance stationary if and only if all the roots of its characteristic equation are outside of the unit circle [Engle, 1982]. Considering (6.7) and (6.8), intuitively, when values of $|x_{t-i}|$ are bigger then σ_t^2 is bigger and x_t has more volatility. Therefore, the ARCH models can be used to capture the heteroscedasticity effect.

The ARCH models have also some weaknesses [Tsay, 2005]. For example, the requirements on ω and α_i's for capturing the kurtosis of the data is complicated for models of high orders. Additionally, the ARCH models need many terms in (6.7) in order to sufficiently capture the volatility in practice. A more parsimonious class of models, known as *generalized autoregressive conditional heteroscedasticity* models (GARCH), is proposed by Bollerslev [Bollerslev, 1986] in order to answer such drawbacks. In $\mathrm{GARCH}\,(p, q)$ models, the current conditional variance is modeled as a linear combination of q previous squared values of the series and p previous squared values of the conditional variances:

$$\sigma_t^2 = \omega + \sum_{i=1}^{q} \alpha_i x_{t-i}^2 + \sum_{i=1}^{p} \beta_i \sigma_{t-i}^2 \tag{6.9}$$

or in terms of the backshift operator, more compactly as:

$$\sigma_t^2 = \omega + \boldsymbol{\alpha}\left(\mathrm{B}\right) x_t^2 + \boldsymbol{\beta}\left(\mathrm{B}\right) \sigma_t^2 \tag{6.10}$$

where $\boldsymbol{\alpha}\left(\mathrm{B}\right) = \sum_{i=1}^{q} \alpha_i \, \mathrm{B}^i$ and $\boldsymbol{\beta}\left(\mathrm{B}\right) = \sum_{i=1}^{p} \beta_i \, \mathrm{B}^i$. In GARCH models, the series is again to be assumed to have the form of (6.8) with $\varepsilon_t \sim \mathrm{WN}\left(0, \sigma^2\right)$. The parameter estimation of a GARCH model is addressed in [Francq and Zakoian, 2010, Enders, 2010].

It is shown [Bollerslev, 1986, Francq and Zakoian, 2010, Lindner, 2009] that the $\mathrm{GARCH}\,(p, q)$ model defined by (6.8), (6.9) has a unique weakly stationary solution if and only if $\omega > 0$ and $\boldsymbol{\alpha}\left(1\right) + \boldsymbol{\beta}\left(1\right) < 1$. Positiveness of the conditional variance additionally requires that $\omega > 0$, $\alpha_1, \ldots, \alpha_q > 0$ and $\beta_i, \ldots, \beta_p > 0$. Moreover, for a $\mathrm{GARCH}\,(p, q)$ process, the unconditional mean is zero, the unconditional variance is $\omega \left(1 - \boldsymbol{\alpha}\left(1\right) - \boldsymbol{\beta}\left(1\right)\right)^{-1}$ and the covariance between two different lags is zero. In fact, the non-constant conditional variance causes the series not to be independent, but the constant conditional mean causes the series to be uncorrelated [Ruppert, 2011]. More

theoretical information about GARCH processes including their distributional properties as well as their moments can be found in [Lindner, 2009].

It is shown that a GARCH (p, q) model can be written as a ARCH (∞) [Francq and Zakoian, 2010] in which the weights exponentially decay for larger lags [Engle and Bollerslev, 1986]. Thus a low order GARCH model have similar properties of high order ARCH models with the advantage that far fewer parameters have to be estimated. This is quite favorable since it avoids the constraints imposed by non-negativity conditions of the conditional variance.

It is also informative to see that a GARCH model can be represented as an application of ARMA model to the $\left\{x^2\right\}$ series [Bollerslev, 1986, Tsay, 2005]. In Section 6.1.4.2 we will see that it motivates the use of ACF and PACF for determining degrees of GARCH models [Bollerslev, 1988, Engle and Bollerslev, 1986].

Let $\eta_t = x_t^2 - \sigma_t^2$, it follows that $\sigma_{t-i}^2 = x_{t-i}^2 - \eta_{t-i}$; $i = 0, \ldots, p$. Substituting them in the Equation (6.9) results in:

$$x_t^2 = \omega + \sum_{i=1}^{\max(q,p)} (\alpha_i + \beta_i)\, x_{t-i}^2 + \eta_t - \sum_{i=1}^{p} \beta_i\, \eta_{t-i} \tag{6.11}$$

in which $\alpha_i = 0$ for $i > q$ and $\beta_i = 0$ for $i > p$. It is shown that $\mathrm{E}[\eta_t] = 0$ and $\mathrm{Cov}[\eta_t, \eta_{t-i}] = 0$ for $i \geq 1$, The $\{\eta_t\}$ is not necessarily an independent and identically distributed (iid) sequence. In other words, a GARCH (p, q) can be regarded as an ARMA $(\max(q, p), p)$ process with autoregressive coefficients of $\phi_i = \alpha_i + \beta_i$; $i = 1, \ldots \max(q, p)$ and moving average coefficients of $\theta_i = -\beta_i$; $i = 1, \ldots q$ (see Equation (6.4)) [Bollerslev, 1988].

6.1.3.3 ARMA-GARCH Models

In Sections 6.1.3.1 and 6.1.3.2 we studied ARMA and GARCH models respectively. ARMA models were used to model the conditional mean while GARCH models were employed to model the conditional variance in data (see Table 6.1). There are cases when the series shows correlation both in the series itself and in the square power of the series, i.e. both properties of ARMA and GARCH models are observable. In such cases, the series has a non-constant conditional mean and a non-constant conditional variance. Mixed ARMA-GARCH models are used to model both conditional mean and conditional variance when they depend on the past [Ruppert, 2011, Rachev et al., 2007]. In mixed ARMA-GARCH models, the disturbance term (ε_t) in Equations (6.4),(6.5) is not a white noise any more but in fact a GARCH process.

Formally, let $\{\tilde{x}_t\}$ be a mean adjusted time series, an ARMA $(p_A, q_A)-$GARCH (p_G, q_G) model is defined as [Ling, 2007]:

$$\tilde{x}_t = \sum_{i=1}^{p_A} \phi_i\, \tilde{x}_{t-i} + \varepsilon_t + \sum_{i=1}^{q_A} \theta_i\, \varepsilon_{t-i} \tag{6.12}$$

where

$$\varepsilon_t = \sigma_t \eta_t,$$
$$\sigma_t^2 = \omega + \sum_{i=1}^{q_G} \alpha_i \tilde{x}_{t-i}^2 + \sum_{i=1}^{p_G} \beta_i \sigma_{t-i}^2 \tag{6.13}$$

and $\eta_t \sim \text{WN}\left(0, \sigma^2\right)$ which may be considered to have the standard normal distribution. The conditions on ϕ_i's, θ_i's, ω, α_i's and β_i's are the same as the ones earlier mentioned for ARMA and GARCH models separately. The parameters of a mixed ARMA-GARCH model can be estimated using maximum likelihood estimation method [Francq and Zakoian, 2004].

6.1.4 Methodology for Time Series Modeling

Up to this point, we have discussed theoretical aspects of the time series and we have introduced ARMA, GARCH and mixed ARMA-GARCH models without showing how they are used in practice. In this section we briefly review the methodology of these models and we discuss technical aspects of their practical application.

6.1.4.1 Methodology of ARMA and ARIMA Models

The methodology used for studying of the ARIMA models is usually referred to as the Box-Jenkins methodology, which we explain next[9].

Phase I - Model Identification

S1 *Data Preparation:* In this step the original data should be investigated for adequacy of use for time series analysis. Data should be transformed into a weakly stationary series by removing any deterministic patterns such as trends, seasonal effects, etc. The mean and the variance of the data should be stabilized usually through employing proper transformations.

S2 *Model Selection:* Having a weakly stationary series, the degree of the ARMA (p, q), i.e. p and q, should be estimated by investigating the *autocorrelation function* (ACF) and the *partial autocorrelation function* (PACF) of the data. The candidates values for p and q are suggested by examining the patterns in the ACF and PACF and subsequently comparing them to the theoretical ones.

Now let S_{AR} and S_{MA} be the sets of values suggested for p and q respectively. The set of candidate models are created as [Wei, 2006, Makridakis et al., 1998, Bisgaard and Kulachi, 2011]:

$$M = \{\text{ARMA}\,(p, q) \mid (p, q) \in S_{AR} \times S_{MA} - \{(0, 0)\}\} \tag{6.14}$$

[9] The overall structure is adjusted according to [Makridakis et al., 1998, Rachev et al., 2007].

Due to the estimation process of the sample ACF and PACF from the data, there are differences in the observed patterns in the sample ACFs and PACFs in comparison to their theoretical counterparts. Additionally, the parameter estimation (S3) of a time series is a computationally complex task. This is due to the fact that parameters must fulfill the model's constraints. Therefore, the results are not necessarily completely equate with the theoretical discussions in order to fully capture the dynamics of the system. In these situations, a more conservative approach is to consider more candidate models by setting $S_{AR} = \{0, \ldots, p_{max}\}$ and $S_{MA} = \{0, \ldots, q_{max}\}$. The values of p_{max} and q_{max} can be specified using sample ACFs and PACFs considering that there are not much significant correlations after p_{max} lag in ACF and q_{max} lag in PACF.

Phase 2 - Model Estimation and Testing

S3 *Estimation:* Getting M as the set of candidate models, $p + q + 2$ parameters for each model should be estimated (Section 6.1.3.1). There are different methods for the estimation of parameters. Two of them which are frequently used are *non-linear least square* and *maximum likelihood estimate*[10] (MLE) methods. The latter is shown to be superior [Box et al., 2008].

S4 *Diagnostics:* After estimating the parameters of the candidate models, it should be checked whether all relevant information of the data has been captured by a candidate model and if the model is suitable. In this regard, the deviation of the estimated values by the model from the real observations, i.e. the residuals of model, should not show any systematic pattern that can be used for their prediction [Rachev et al., 2007]. Ideal residuals are Gaussian white noise with almost no correlation [Montgomery et al., 2008] (see Section 6.1.3).

Anderson-Darling test can be used to check the normality of residuals[11]. The ACF and the PACF of the residuals as well as the *Ljung-Box Q-test*[12] [13] are used to test whether there is correlation between the residuals [Tsay, 2005, Bisgaard and Kulachi, 2011, Montgomery et al., 2008].

If a model is adequate in this respect, it will remain in M. If $M = \{\}$ we can go to Step S2 and try to find new candidate models or stop and concluding that there is no suitable ARMA model capable of describing the dynamics of the data.

S5 *Best Model Selection:* If there is more than one candidate in M, we will have some suitable models to describe the dynamics of the data. The superior model

[10] Since in the MLE method the estimation cannot be done by solving systems of linear equations, the surface of the likelihood function should be numerically searched for the maximum. In this regards, the Yule-Walker and the Hannan-Rissanen algorithms [Brockwell and Davis, 2002] provide good estimation of the parameters which can be used by the MLE method as the starting points of the numerical search.

[11] Alternatively, one can use other tests such as Kolmogorov-Smirnov and Lilliefors which are less powerful [Razali and Wah, 2011].

[12] See Sections 6.1.4.2 and 6.2.2 for more information about the Ljung-Box Q-test.

[13] Box-Pierce is an alternative but less powerful test in this regard [Montgomery et al., 2008].

in M is the one with lower *Akaike information criterion* (AIC)[14]. AIC rewards the goodness of the fit and accuracy, and penalize the overfitting or complexity of models [Tsay, 2005, Wei, 2006, Box et al., 2008].

There are cases where differences between AIC of models are negligible, in these cases it might be more appropriate to have a hold-out set of the last few observations and assess the power of models by comparing the forecast for the hold-out set with real observations in the set and take the model with finest accuracy as the best model.

Phase 3 - Applications

S6 *Forecasting and Simulation:* The best model can be used to predict the future of the system or it can be used to simulate the dynamics of the system and produce infinite similar time series all obeying the same characteristics. Minimal mean squared error method is frequently used in forecasting [Kirchgässner and Wolters, 2007, Montgomery et al., 2008]. More information about forecasting, calculation of them and their errors bands are addressed in [Brockwell and Davis, 2002, 2006].

6.1.4.2 Methodology of GARCH Models

Regarding the methodology of GARCH models, the following steps are usually practiced in the related literature. The steps are similar to the ones that we mentioned for ARMA methodology, but in each step different criteria should be checked. At first, the data should be transformed into weakly stationary. The next step is to detect the existence of the heteroscedasticity effect, i.e. if the conditional variance is not constant. This can be done using three methods.

The first simple method is to use the ACF plot of the squared series in order to detect significant positive correlations. This is due to the fact that a GARCH (p, q) can be written as an ARMA model of squared series as given by (6.11) (see Section 6.1.3.2). Additionally, considering (6.7) and (6.9), the bigger values of x_{t-i}^2 will have more effect in the variance of the series and therefore bigger magnitude for x_t is expected [Ruppert, 2011]. McLeod and Li first applied this approach on the squared residuals of ARMA models [McLeod and Li, 1983]. As the second method, the former investigation of correlation in the squared series can be more accurately accomplished using the *Ljung-Box Q-test* [Ljung and Box, 1978]. The null hypothesis of the test assumes that the first h lags of the ACF of $\left\{x_t^2\right\}$ are not significantly different from zero. As the third method, the more formal test for detecting the heteroscedasticity effect is the *Engle's ARCH test* [Engle, 1982]. We will provide detailed information about the Ljung-Box and Engle tests in Section 6.2.2 when we apply them to our data.

After proving the existence of the heteroscedasticity effect, the next step in to determine the suitable degrees for ARCH or GARCH models and form the set of possible candidate models for capturing the effect. In the case of an ARCH model, Tsay [Tsay,

[14] One can also use the *Bayesian information criterion* (BIC) [Schwarz, 1978]. Although in our problem, it is not appropriate (see Section 6.5.2).

2005] suggests to use the PACF of the $\{x_t^2\}$ in order to determine the order of the model. He uses ARMA representation of GARCH model given by (6.11). Setting $p = 0$, (6.11) becomes $x_t^2 = \omega + \alpha_1 x_{t-1}^2 + \ldots + \alpha_q x_{t-q}^2 + \eta_t$ which is an AR (q) model of $\{x_t^2\}$. Therefore, the PACF of $\{x_t^2\}$ can be used to determine the order of q (see Section 6.1.4.1). He warns that the PACF of $\{x_t^2\}$ might not be quite efficient since $\{\eta_t\}$ is not identically distributed.

In the case of GARCH models, it is pointed out that accurately determining the degrees are difficult [Tsay, 2005]. Using the ARMA representation of a GARCH model (6.11), Wei [Wei, 2006] argues that the ACF and PACF patterns of $\{x_t^2\}$ will show patterns of exponential decay and similar techniques discussed for ARMA models can be used to detect the proper degrees. As we mentioned in Section 6.1.3.2, a GARCH (p, q) model can be written as an ARCH (∞). Therefore, usually several low order GARCH models are used as candidates for capturing the heteroscedasticity in practice [Enders, 2010, Tsay, 2005]. The orders of p and q are usually two or one [Guidolin, Accessed: July, 2014, Rossi, 2004].

The next step will be the estimation of GARCH models. The least square, maximum likelihood and quasi-maximum likelihood methods are used for estimating the parameters of an ARCH or a GARCH model [Francq and Zakoian, 2010, Bollerslev et al., 1994, Enders, 2010].

The final steps are diagnosing the estimated models as well as selecting the best model. Similar to ARMA models, the "standardized residuals" of GARCH models should be Gaussian white noise with no correlations[15]. The *standardized residuals* of a GARCH model are calculated by dividing the inferred residuals of the model by its inferred conditional standard deviation [Tsay, 2005, Enders, 2010]. The Akaike (AIC) and Bayesian (BIC) information criteria are also used to select models in practice [Tsay, 2013, Zivot and Wang, 2006, Ruppert, 2011].

6.1.4.3 Methodology of ARMA-GARCH Models

The methodology for ARMA-GARCH models are principally the same as we mentioned for ARMA and GARCH models in Sections 6.1.4.1 and 6.1.4.2 respectively. In practice the same steps are taken to investigate the suitability of ARMA and GARCH models for the data and to detect the proper orders for models.

In order to estimate an ARMA-GARCH model, two methods of Maximum Likelihood and Quasi-Maximum Likelihood are respectively discussed in [Francq and Zakoian, 2004] and [Francq and Zakoian, 2010]. A practical implementation for estimating the parameters of a model is addressed in [Wurtz et al., 2006].

Similarly to GARCH models, to diagnose an estimated ARMA-GARCH model the standardized residuals are used (see Section 6.1.4.2) and AIC and BIC are employed for selecting the proper model.

[15] In this work we considered GARCH models which have the Gaussian white noise, but as we mentioned in Section 6.1.3.2, Student's t-distribution is also used in other application domains especially in the econometrics literature.

6.1.5 Accuracy of Forecasts

The accuracy of approximations and forecasts as well as proper measurements of errors are of special interest. It is well established that the choice of error measures is dependent on the situation, nature of the problem, the purpose of predictions as well as the needs of decision makers [Makridakis and Hibon, 1995]. Evaluation of different error measures, their advantages and disadvantages have been thoroughly studied in [Armstrong and Collopy, 1992, Makridakis and Hibon, 1995, Hyndman and Koehler, 2006]. Armstrong [Armstrong, 2001] generally recommends the error measures which are not affected by scale, are not biased (not giving too much weight to some observations and their forecasts) and have low sensitivity to outliers, although always a trade-off between these incompatible goals is unavoidable [Makridakis and Hibon, 1995].

Two of the mostly used measures which are employed by academic and practitioners are the mean absolute percentage error (MAPE) and mean squared error (MSE) [Armstrong, 2001]. Both methods are reported to have disadvantages that the amplitude of the measurements are not properly considered [Armstrong and Collopy, 1992, Makridakis and Hibon, 1995, Hyndman and Koehler, 2006]. Before going into more details, we first provide the requirements.

Let $\hat{x}$ be the forecast of observation x, the *error* is defined as $e = x - \hat{x}$. The *relative error*, in percent, is defined by $\delta_x = |x - \hat{x}|/|x| \times 100$ where $x \neq 0$. The drawback of the relative error is that it is adversely affected by small values. For instance suppose that $x = 1, \hat{x} = 2$ and $y = 100, \hat{y} = 101$ then $\delta_x = 100\%$ and $\delta_y = 1\%$ although for both the absolute error is 1. Therefore, the MAPE which is defined as MAPE $= 100 \left(\sum_{i=1}^{n} |x_i - \hat{x}_i| \,/\, |x_i|\right) /n$, is not immune to the magnitude of measurements.

Despite the interesting properties of the MSE, defined as MSE $= \sum_{i=1}^{n} (x_i - \hat{x}_i)^2 /n$, it has the disadvantage that it is not a scale free measure which makes it inappropriate for comparing the models on various data sets with different standard deviations and scales [Hyndman and Koehler, 2006, Lughofer, 2011].

We now introduce the proper way of assessing the accuracy forecast for our analysis. Both of our measured low-level and high-level changes show very volatile behavior and the magnitude of changes for each project and within the projects are quite erratic (see Section 6.2.1, Table 6.4 and Figure 6.1). The same issue is also reported in [Zhou et al., 2006] when predicting highly volatile internet traffic data. A similar approach to ours is also proposed there as well as in [Billah et al., 2006].

To solve the aforementioned problems and handle volatile data, *normalized relative error* (NRE) has been introduced. Suppose that X is a set of possible outcomes for x, then NRE (in percent) is defined as:

$$\text{NRE} = 100 \times \frac{|x - \hat{x}|}{\max(X) - \min(X)}$$

Similarly for $x_1, x_2, \ldots, x_n$ observations and their corresponding forecasts, *normalized mean squared error* (NMSE) is defined[16] by [Lughofer, 2011]:

[16] Another way of normalizing the MSE is by dividing it to the variance of the data.

$$\text{NMSE} = \frac{1}{n}\sum_{i=1}^{n}\frac{(x_i - \hat{x}_i)^2}{(\max(X) - \min(X))^2} \tag{6.15}$$

The NRE and NMSE have the advantage of being scale free and not affected by small measurements (not biased). Additionally both of them are more practical since they are normalized by their ranges [Lughofer, 2011]. Equation (6.15) is used to assess the quality of forecasts by time series models in Section 6.3.2.1. Similar to our arguments, Billah et al [Billah et al., 2006] and Zhou et al [Zhou et al., 2006] argue that the normalized version of MSE is much more suitable for data with high volatility and different variances.

6.2 Modeling the Evolution

In Chapter 4 we showed that the changes of design models can be represented at two different abstraction levels of low-level and high-level changes. For low-level changes we have 75 low-level difference metrics which are defined as the application of 5 edit operations of addition, deletion, move, reference change and update onto 15 different model element types (packages, interfaces, classes, etc.). For high-level changes, we had 188 edit operations which their occurrences were considered as high-level difference metrics. They were categorized into creates, deletes, moves, set/unset, modifying non-containment references and refactorings.

Let $l_{p,i}$ be the sum of all of low-level metrics computed between design models of revisions i and $i+1$ in project p. L_p, which is defined as the sequence of $l_{p,1}, l_{p,2}, \ldots, l_{p,n_p}$, is our data set for the analysis of the evolution of project p using low-level difference metrics (n_p is the length of project p). $H_p = h_{p,1}, h_{p,2}, \ldots, h_{p,n_p}$ is defined analogously for analysis of the evolution of p, using high-level difference metrics.

Regarding the theory and methodology of time series analysis in Section 6.1.4, the steps S4 and S5, i.e. "diagnostics" and "best model selection" are sufficient for a time series model to be valid. But in order to check the performance of a time series model in practice, we use "out of sample evaluation" for each proposed model. In this regard, we partitioned our measured sequence of $\{x_1, x_2, \ldots, x_N\}$ into two disjoint subsets. The first set, called "base-set", consists of the observations of x_1 up to and including x_{N-6}. The second one, which is called the "hold-out" set, consists of the last six observations. We tried to estimate our time series models on the base-set. The fitness of the proposed models is then checked by comparing their six step forecasts with the actual observations in the hold-out set. In Section 6.5.3 we argue why we chose a hold-out set of length six. In what follows we report our analysis and measured statistics based on the hold-out sets of length six. All of our analysis and calculations are done using Mathworks Matlab® R2013b computational engine [MathWorks Company, 2013].

6.2.1 Data Description and Transformation

As mentioned earlier, our measurements are series of the total number of changes applied between revisions of a software system p, which are calculated separately for low-level

(L_p) and high-level (H_p) changes. In this section we discuss general properties of our measurements and discuss how we transform them into weakly stationary form.

Both of L_p's and H_p's show high volatility. In other words, in the calculated series one obeserves both big and small changes together. As an illustration, Figure 6.1 shows the series of low-level changes for the ASM project. Moreover, the data do not show cyclic or seasonal patterns.

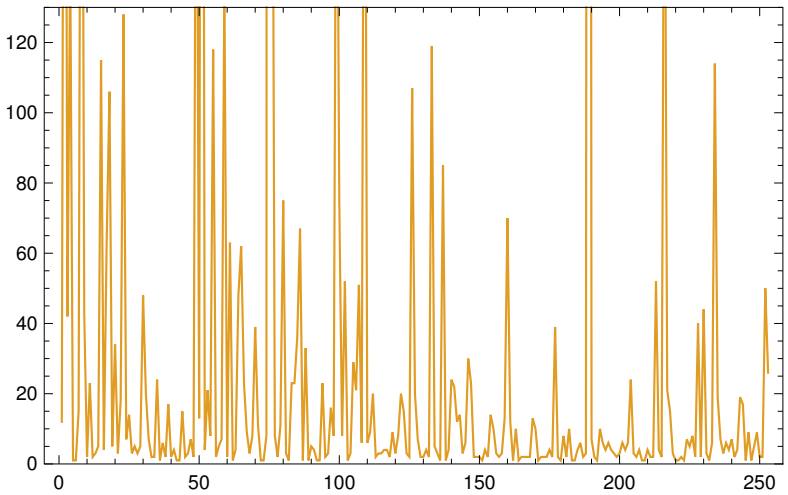


Figure 6.1: ASM project - Total number of changes in the hold-out set. For better visibility, the y-axis is limited to 130 and bigger values are not shown.

Another feature is high variation in the data, i.e. the variances of the measured data are big. For instance, Figure 6.2 shows the empirical variance of the low-level changes as a function of sample size (revisions) for the Struts project. As shown, the variance is erratic and it takes big values. Table 6.4 provides basic statistics about the measured data both for low-level and high-level changes. As shown there the variance of the data is quite big. Additionally, we observed that the distributions of our observations are not symmetric and are skewed toward right (see Chapter 5).

Due to the aforementioned characteristics, the data are not stationary, inappropriate for the time series analysis and therefore have to be transformed into weakly stationary form. In order to stabilize the variance of the data, power and logarithmic transformations are typically employed. The logarithmic transformation is usually used in the financial time series analysis (see Section 6.1.3.2).

For stabilizing the variance, we used the *Box-Cox transformation* (BCT) which gives the logarithmic transformation as its special case[17]. BCT was proposed by Box and Cox [Box and Cox, 1964] with regard to the analysis of linear models in order to stabilize the variance of errors, normalize their distributions and reduce their heteroscedasticity

[17] Alternative, but in our work less suitable method is the logarithmic transformation. Our experiments revealed that the logarithmic transformation is not appropriate, since the residuals of the time series models will not be normally distributed.

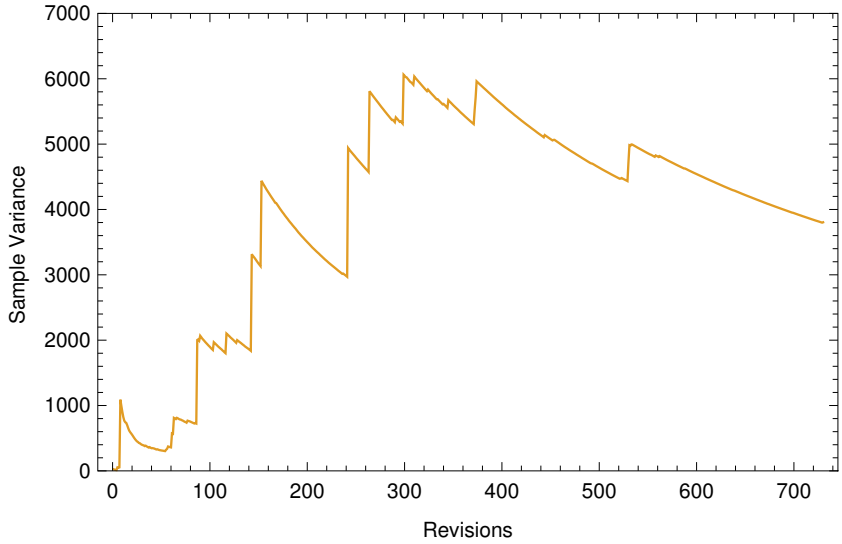


Figure 6.2: Project Struts - Empirical variance of low-level changes as a function of sample size (revisions).

[18]. Principally, BCT tries to transform the data in a way that the transformed data have the normal distribution. Suppose that positive $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ is given, the Box-Cox transformation of $x_i > 0$, i.e. $\tau(x_i; \lambda)$, is defined by[19]:

$$\tau(x_i; \lambda) = x_i^{(\lambda)} = \begin{cases} \frac{x_i^{\lambda} - 1}{\lambda} & , \lambda \neq 0 \\ \ln(x_i) & , \lambda = 0 \end{cases} \tag{6.16}$$

where λ is the optimum transformation parameter and is obtained by maximizing the logarithm of the likelihood function of the data, given by [Viktor, 2010, Porunov, 2010]:

$$f(\mathbf{x}; \lambda) = -\frac{n}{2} \ln \left[\sum_{i=1}^{n} \frac{\left(x_i^{(\lambda)} - \bar{x}(\lambda) \right)^2}{n} \right] + (\lambda - 1) \sum_{i=1}^{n} \ln(x_i)$$

where $\bar{x}(\lambda) = \frac{1}{n} \sum_{i=0}^{n} x_i^{(\lambda)}$.

As we mentioned in Section 6.1.3.2, in the analysis of financial time series, a very common transformation of the series $\{x_t\}$ is the series of $y_t = \ln x_t - \ln x_{t-1}$. In other words, first a logarithmic transformation and then a differencing is applied [Tsay, 2005, Enders, 2010, Hamilton, 1994]. Similarly, we successfully transformed all of our sample data sets into weakly stationary by applying the following three step transformations which we abbreviate it as the BCDM transformation:

[18] A detailed bibliography of the research related to the Box-Cox transformation can be found in [Sakia, 1992].

[19] Non-positive data can be first shifted by adding a suitable positive number.

Project	Low-level Changes		High-Level Changes	
	Max Log-Like.	**λ Opt.**	**Max Log-Like.**	**λ Opt.**
ASM	-575.934	-0.259122	-554.641	-0.265261
CheckStyle	-1899.74	-0.206358	-1785.84	-0.217717
DataVision	-38.8586	-0.362303	-37.4666	-0.346967
FreeMarker	-845.015	-0.073134	-820.550	-0.050067
HSQLDB	-3328.74	-0.511699	-3257.16	-0.491147
Jameleon	-694.797	0.073503	-686.145	0.089726
JFreeChart	-971.206	-0.117241	-946.980	-0.126192
Maven	-1523.80	-0.257195	-1364.48	-0.263548
Struts	-1490.00	-0.221296	-1405.69	-0.226861

Table 6.2: Box-Cox transformation - Optimum values of λ and the maximum of the corresponding logarithm-likelihood function.

(T1) A Box-Cox transformation using the optimum λ.

(T2) A Difference transformation of degree One.

(T3) The series Mean of the previous step is subtracted.

The optimum values of λ used in our BCDM transformations as well as the corresponding maximum values of the logarithm of the likelihood function is given in Table 6.2.

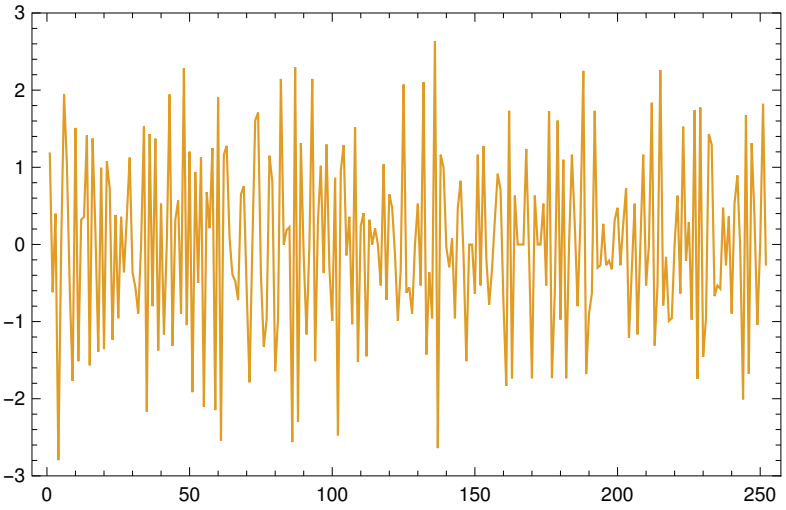


Figure 6.3: ASM project - Fully transformed series using BCDM transformation.

Figure 6.3 shows the transformed low-level changes for the ASM project. Comparing it to Figure 6.1, apparently the variance of the data is stabilized and the series is fluctuat-

Project	Low-level Changes		High-Level Changes	
	p-value	**Stat.**	**p-value**	**Stat.**
ASM	0.2696	2.3230	0.3020	-7.695
CheckStyle	0.5000	0.2216	0.5000	0.1211
DataVision	0.5000	0.8079	0.4436	0.9805
FreeMarker	0.5000	0.6975	0.5000	0.8083
HSQLDB	<0.001	113.75	<0.001	128.65
Jameleon	0.0018	21.029	0.0029	17.904
JFreeChart	0.0304	7.1826	0.0253	7.7394
Maven	0.3623	1.9200	0.0890	4.6628
Struts	0.3151	2.1992	0.3509	1.9811

Table 6.3: Jarque-Bera test of normality for the transformed data.

ing around the mean level of zero. Thus, the transformed series is weakly stationary and the time series analysis can be applied on it (see Section 6.1.3). Table 6.4 contains the summary statistics for the original as well as the BCDM transformed series of low-level and high-level changes. For all projects, the variance and higher moments (skewness and kurtosis) of the data have been stabilized after the BCDM transformation and as shown in the table, their calculated values are near to ones of a normal distribution. To investigate that, we used the Jarque-Bera test of normality which considers the skewness and kurtosis of the data [Rachev et al., 2007]. The null hypothesis of the test is that the data obeys a normal distribution with an unknown mean and variance. As shown in Table 6.3, the results indicate that the null hypothesis is not rejected at the significance level of 0.01 for all projects except HSQLDB and Jameleon. Later in Section 6.2.3 we will show that the effect of non-normality of the transformed data will degrade the time series models with respect to the number of valid candidate models that have normal residuals.

6.2.2 Time Series Models of Evolution

As we mentioned earlier, our main intention was to mathematically model the evolution of software systems at the abstraction level of design models and later to use it for generating more realistic model histories. The first step in the time series analysis is to transform the data into weakly stationary form (see Section 6.1.4.1). We observed that our data sets had to be transformed into weakly stationary form in order to let us study their dynamics.

The next step (S2 in Section 6.1.4.1) is to find a set of candidate time series models by analyzing the ACF and PACF of the data. The sample ACFs and PACFs should be compared with the theoretical ones of ARMA models in order to find some proper candidate degrees for ARMA (p, q) models. For all projects, we carefully analyzed the sample

Project	Change Type	Data Kind	Mean	Variance	Std. Dev.	Skewness	Kurtosis	Min	Max
ASM	Low Level	Orig. Obser.	35.72	21455.7	146.48	10.29	127.13	1	1973
		BCDM Tran.	≈ 0	1.3	1.14	0.004	2.53	-2.789	2.894
	High Level	Orig. Obser.	31.21	12790.8	113.096	8.64	93.99	1	1404
		BCDM Trans.	≈ 0	1.26	1.121	-0.0283	2.556	-2.757	2.857
CheckStyle	Low Level	Orig. Obser.	15.78	4286.44	65.47	17.55	354.81	1	1366
		BCDM Tran.	≈ 0	1.280	1.131	-0.0349	3.020	-3.266	3.762
	High Level	Orig. Obser.	13.17	2506.91	50.07	22.104	599.24	1	1406
		BCDM Trans.	≈ 0	1.205	1.098	0.006	2.948	-3.212	2.987
DataVision	Low Level	Orig. Obser.	15.52	781.51	27.95	2.121	6.209	1	105
		BCDM Tran.	≈ 0	1.062	1.031	-0.158	2.158	-2.102	1.576
	High Level	Orig. Obser.	14.04	590.54	24.301	1.996	5.533	1	86
		BCDM Trans.	≈ 0	1.130	1.062	-0.152	2.057	-2.165	1.595
FreeMarker	Low Level	Orig. Obser.	26.18	4747.62	68.90	7.916	80.644	1	851
		BCDM Tran.	≈ 0	1.94	1.393	0.032	3.213	-4.336	4.276
	High Level	Orig. Obser.	22.10	2204.27	46.95	6.401	55.388	1	475
		BCDM Trans.	≈ 0	2.06	1.436	0.063	3.205	-4.570	4.648
HSQLDB	Low Level	Orig. Obser.	101.50	153553	391.858	12.104	162.326	8	5884
		BCDM Tran.	≈ 0	0.009	0.097	0.002	4.689	-4.011	4.367
	High Level	Orig. Obser.	89.62	109683	331.184	14.083	222.714	8	5884
		BCDM Trans.	≈ 0	0.011	0.103	-0.0285	4.795	-4.252	4.638
Jameleon	Low Level	Orig. Obser.	21.35	4562.61	67.55	14.865	238.43	1	1107
		BCDM Tran.	≈ 0	2.886	1.699	-0.0898	4.328	-6.765	6.777
	High Level	Orig. Obser.	20.25	3539.65	59.49	14.626	233.051	1	971
		BCDM Trans.	≈ 0	3.197	1.788	-0.0718	4.228	-7.061	7.074
JFreeChart	Low Level	Orig. Obser.	30.54	3977.07	63.06	4.542	29.465	1	599
		BCDM Tran.	≈ 0	1.489	1.220	0.032	3.687	-4.148	4.141
	High Level	Orig. Obser.	29.23	4045.08	63.60	5.015	35.981	1	646
		BCDM Trans.	≈ 0	1.481	1.217	0.0186	3.715	-4.061	4.054
Maven	Low Level	Orig. Obser.	27.17	17424.6	132.002	13.649	245.783	1	2735
		BCDM Tran.	≈ 0	1.299	1.140	-0.001	2.756	-3.044	3.391
	High Level	Orig. Obser.	16.86	3815.49	61.77	9.946	131.873	1	1063
		BCDM Trans.	≈ 0	1.232	1.110	-0.015	2.622	-2.999	2.966
Struts	Low Level	Orig. Obser.	21.019	3789.49	61.559	6.613	55.246	1	715
		BCDM Tran.	≈ 0	1.403	1.184	0.007	2.732	-3.149	3.272
	High Level	Orig. Obser.	16.936	1960.51	44.28	6.236	50.263	1	487
		BCDM Trans.	≈ 0	1.334	1.155	0.037	2.757	-3.075	3.248

Table 6.4: Project summary - Basic statistics (based on the hold-out set).

ACF and PACF plots of low-level and high-level changes. We investigated whether at the significance of 0.05, the lags in the plots are significantly different from zero. For most of our data sets it turned out that, although after the first few beginning lags the depicted correlation in the ACFs and PACFs plots cut off to the confidence band (not significant anymore), again they show significant differences from zero few lags later. In other words, we conclude that both of lower and higher order ARMA models should be taken as candidates for further analysis and the suitability of them should be further investigated in the diagnostics step (S4 in Section 6.1.4.1). Therefore[20], based on our

[20] Another reason is that we want to objectively compare the effects of the transformation as well as the best model selection strategy (using AIC or BIC) on the candidate time series models and their residuals (see the materials later in this section and see Sections 6.2.3 and 6.5.2).

observations, for all sample projects except DataVision we set the p_{max} and q_{max} to 25 and we formed the set of our ARMA candidate models of M as explained by Equation (6.14) in step S2 of Section 6.1.4.1. For the DataVision project, we set p_{max} and q_{max} to 15. The reason for that is that DataVision is the smallest project in our sample set and it has totally 29 revisions (see Table 4.3).

The next important issue to consider, is that whether the data sets show any heteroscedasticity effect, i.e. if their conditional variances are not constant. Such property was the main motivation behind the GARCH models in econometrics (see Section 6.1.3.2). In the econometrics literature, *volatility clustering* refers to the phenomenon that large variation in the data is more likely followed by large variation than the small ones [Cont, 2001]. This phenomenon does not indicate the lack of stationarity but shows the dependence in conditional variances of the series [Ruppert, 2011]. The *ARCH effect* refers to the effect of dependence of the conditional variance to its previous values [Tsay, 2005].

As we discussed in Section 6.1.4.2, there are two methods for detecting the ARCH effect and the existence of volatility clustering in data. The first method is based on the existence of significant autocorrelation in the squared series of $\{x_t^2\}$. When there are significant positive correlations, we can conclude that the ARCH effect exists in the data and we can employ the GARCH models. To test for the significant correlations, the *Ljung-Box Q-test* is frequently employed[21] [Ljung and Box, 1978]. The test statistic is given as:

$$\mathrm{Q}(h) = n(n+2)\sum_{k=1}^{h}\frac{\hat{\rho}_k^2}{n-k} \tag{6.17}$$

in which n is the length of data and $\hat{\rho}_k$ is the lag-k sample autocorrelation coefficient. The test statistic has an asymptotic χ^2 distribution with h degrees of freedom when the data is serially uncorrelated. The null and alternative hypotheses are $\mathcal{H}_0 : \rho_1 = \cdots = \rho_h = 0$ and $\mathcal{H}_a : \rho_k \neq 0$ for some $k \in \{1, \ldots, h\}$; respectively.

It is shown that the power of the test is affected by the number of lags (h) used in the calculation of the statistics. It seems that there is no common agreement on the appropriate number of lags which should be used in the test. Tsay [Tsay, 2005] suggests $h = \ln(n)$ based on simulation studies. Hyndman and Athanasopoulos [Hyndman and Athanasopoulos, 2013] suggest $h = 10$ for non-seasonal data and in [MathWorks Company, 2013] $h = \min(20, n-1)$ is suggested.

The second method which is more formal is the *Engle's ARCH test* [Engle, 1982]. Similar to Equation (6.7) in Section 6.1.3.2, when there is an autocorrelation in the squared series, then for some $\alpha_k \neq 0$ we have ${x_t}^2 = \alpha_0 + \alpha_1 {x_{t-1}}^2 + \cdots + \alpha_h {x_{t-h}}^2$. Similarly to the Ljung-Box test, the null and alternative hypothesis are defined as $\mathcal{H}_0 : \alpha_1 = \cdots = \alpha_h = 0$ and $\mathcal{H}_a : \alpha_k \neq 0$ for some $k \in \{1, \ldots, h\}$. The statistic of the Engle's ARCH test is provided in [Engle, 1982, Tsay, 2005] and is showed that have a χ^2 distribution with h degrees of freedom.

[21] The other but less powerful alternative is the Box-Pierce test.

We might ask that for which lag of h the test is more powerful. A simple practical procedure is suggested [MathWorks Company, 2013]. Consider ARCH (k) models with $k \in \{1, ..., h_{max}\}$, fit them to the data and then choose the degree of the one with the smallest AIC (Akaike information criterion), i.e. $h = \arg\min_k \text{AIC}(\text{ARCH}(k))$. The h_{max} is usually assumed to be at least $p + q$ for models of GARCH (p, q) since they are locally as good as ARCH $(p + q)$ models. For both of ARCH and GARCH models we can also consider the ACF of the squared series in order to choose the h_{max}.

Let us go back to our data sets, for which we wanted to check the presence of the ARCH effect. Figure 6.4 shows the ACF plot of the squared data for the high-level changes of the Jameleon project. At the level of 0.05, there are significant correlations in the first, sixteenth and twenty fourth lags, indicating the existence of the ARCH effect. All of our data sets of low-level and high-level changes except for the projects DataVision and Struts, show similar behavior which shows the presence of the ARCH effect.

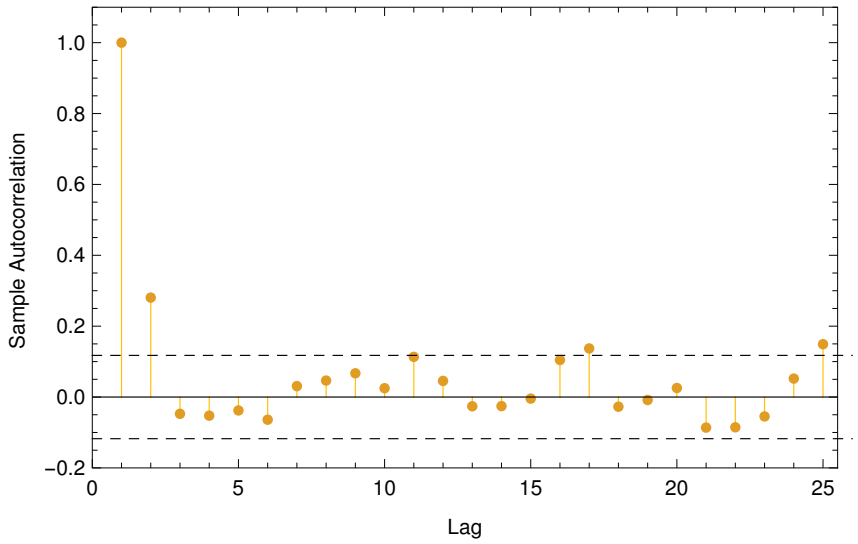


Figure 6.4: Project Jameleon - ACF plot of the squared high-level changes.

In order to approve the existence of the ARCH effect, we employed the more formal method of the Engle's ARCH test. As we described earlier we chose $h_{max} = 25$ and we progressively fitted the ARCH models up to the degree of h_{max}. The choice of 25 was based to the fact that mostly no significant correlation was observable in the ACF plot of the squared data sets after the twenty-fifth lag. We then computed the log-likelihoods of the models in order to calculate their AIC values. It turned our that for all of our data sets the ARCH (1) model delivers the best fit with the lowest AIC value. This result was not surprising since for all of our data sets, the most significant correlation was measured at the first lag. Therefore, for all of our data sets, the suitable lag of the Engle's ARCH test is $h = 1$. Table 6.5 shows the results of our ARCH test. The

Project	Low-Level Changes			High-Level Changes		
	AIC	**p-value**	**Stat.**	**AIC**	**p-value**	**Stat.**
ASM	785.53	2.3295×10^{-2}	5.1464	773.36	1.0042×10^{-2}	6.6274
CheckStyle	3057.5	4.2400×10^{-10}	38.999	3000.1	4.6200×10^{-10}	38.830
DataVision	62.376	1.0893×10^{-1} $\diamond$	2.5697	68.038	1.6100×10^{-1} $\diamond$	1.9648
FreeMarker	1137.1	1.8400×10^{-11}	45.135	1158.9	5.2800×10^{-13}	52.100
HSQLDB	-1882.7	≈ 0	75.617	-1782.2	≈ 0	77.296
Jameleon	1067.7	7.5100×10^{-06}	20.058	1095.6	3.0500×10^{-06}	21.782
JFreeChart	1145.2	4.4000×10^{-05}	16.691	1138.1	4.5800×10^{-07}	25.433
Maven	2385.9	9.9200×10^{-06}	19.527	2332.8	1.5100×10^{-08}	32.036
Struts	2321.7	6.4139×10^{-2} $\diamond$	3.4270	2284.6	1.0398×10^{-1} $\diamond$	2.6435

Table 6.5: Engle's ARCH test of heteroscedasticity ($\diamond$: Significant at 0.05).

"AIC" columns correspond to the ARCH (1) model which had the lowest AIC value for all data sets. The columns "p-value" and "Stat." show the p-values and the statistics of the test respectively. At the significance level of 0.05, the null hypothesis of no ARCH effect cannot be rejected for just the low-level and high-level changes of DataVision and Struts projects[22], indicating that the other data sets exhibit the ARCH effect.

Now that the presence of the ARCH effect and volatility clustering is shown by the Engle's ARCH test, the next step will be determining the proper order of the GARCH models. According to that, the set of candidate models is then formed in order to capture the heteroscedasticity effect in the data. As we showed earlier the best model that was used in the Engle's ARCH test was the ARCH (1) model. Our observations also approves the presence of significant correlation in the first lag/first few lags of the ACF plot of the squared data. Therefore we would expect that a low order GARCH model could be used to properly model the conditional variance of our data. In this regard we consider the GARCH models of orders of maximum two. This selection is also strongly supported by the fact that a GARCH (p, q) model even with small orders can be represented as an ARCH (∞) model (see Section 6.1.3.2). To see that [Guidolin, Accessed: July, 2014, Rossi, 2004], consider the GARCH $(1, 1)$ model with conditional variance of $\sigma_{t+1}^2 = \omega + \alpha x_t^2 + \beta\sigma_t^2$. By recursively substituting the equation in itself we obtain:

$$\sigma_{t+1}^2 = \omega \sum_{j=0}^{\infty} \beta^j + \alpha \sum_{j=0}^{\infty} \beta^j x_{t-j}^2 + \lim_{j\to\infty} \beta^j \sigma_{t-j}^2$$

Since ω, α, β are positive and $\alpha+\beta < 1$, we have $0 < \beta < 1$. Provided that the series had started far enough in the past we get $\lim_{j\to\infty} \beta^j \sigma_{t-j}^2 = 0$ and $\omega \sum_{j=0}^{\infty} \beta^j = \omega / (1 - \beta)$. Therefore we have:

$$\sigma_{t+1}^2 = \frac{\omega}{1-\beta} + \text{ARCH}(\infty)$$

[22] The same results can be obtained by employing the Ljung-Box Q-test on the squared data.

in which coefficients of the ARCH model decay according to the powers of β.

To sum up our actions for forming the candidate models, based on the aforementioned discussions, the ARMA models with degrees of up to 25 (except for the small project of DataVision which we chose 15) and the GARCH models with degrees of up to 2 will be the proper choices to model the evolution of design models in our sample set of Java software systems. We observed that all our data sets (except the ones associated to the projects DataVision and Struts) show the heteroscedasticity effect and therefore the mixed ARMA-GARCH models should be formally used to model the evolution.

Since the ARCH effect is more significant in the first lag/first few beginning lags and is not persistent for a long time, we are also interested to see if the heteroscedasticity effect can satisfactorily be modeled by pure ARMA models or not, i.e. with ARMA models that have constant variance instead of a GARCH variance. Moreover, another interesting research question is the forecast performance of pure ARMA and the mixed ARMA-GARCH models. In this regard, we present our findings in Section 6.3 where we assess and compare ARMA and ARMA-GARCH models. We also discuss the forecast performance and other aspects of the employed models there.

We are also interested to see that how many of the considered ARMA and mixed ARMA-GARCH candidate models will pass the diagnostic tests after their parameters are estimated. Whether the transformation has any effect in the performance of ARMA and mixed ARMA-GARCH models and if so how such an effect will look. These issues are covered in the next section, i.e. Section 6.2.3.

According to the above remarks, we thus considered two sets of candidate models. The first set is the set of ARMA candidate models with degrees of up to 25. We denote this set by M_1. M_1 has a total of 675 ARMA models. The second set of candidate models, M_2, consists of the mixed ARMA-GARCH models in which the degree of the ARMA part is up to 25 and the degree of the conditional variance part (GARCH part[23]) is of up to 2. The set M_2 has totally 4050 models. Since for the small project of DataVision we used models of up to 15 degree, the sets of M_1 and M_2 have 255 and 1530 models respectively. In the next section, we will talk about the estimation of the candidate models in sets M_1 and M_2 and check their adequacy for the requirements mentioned in steps S4 and S5 of Section 6.1.4.1.

6.2.3 Estimation and Diagnostics of the Time Series Models

Up to now, we described how we examined the transformed data for forming sets of suitable candidate models that can describe the evolution. We formed the two candidate sets of M_1 and M_2. The set M_1 consists of pure ARMA models i.e. ARMA models which have a constant variance term in their formulation. The set M_2 is the set of mixed ARMA-GARCH models, namely ARMA models with a GARCH variance formulation.

For each project p, we estimated the parameters of models in candidate sets of M_1 and M_2, on the data sets of low-level (L_p) and high-level changes (H_p). The estimation was done using the maximum likelihood estimation method.

[23] The GARCH part has totally 6 models.

Project	Low-level Changes		High-Level Changes	
	M_1	M_2	M_1	M_2
ASM	98.07%	97.51%	98.96%	98.49%
CheckStyle	33.93%	21.93%	16.74%	14.07%
DataVision	9.08%	NC	8.40%	NC
FreeMarker	98.67%	95.21%	99.11%	98.27%
HSQLDB	0%	0.15%	0%	0.05%
Jameleon	2.37%	3.58%	3.11%	4.47%
JFreeChart	77.04%	86.17%	86.67%	87.06%
Maven	9.93%	7.68%	4.44%	3.14%
Struts	8.89%	NC	4.44%	NC

Table 6.6: Percent of candidate models in candidate sets of M_1 and M_2 with normal residuals (NC: Not Considered).

As we described in Section 6.1.4, the estimated time series models should be investigated to see whether they meet the requirements mentioned in the ARMA and GARCH methodologies. The first requirement is that the inferred standardized residuals of the models should be Gaussian white noise with no correlation. The residuals should show no systematic pattern that helps one to predict their values from their corresponding observations.

For candidate models in sets of M_1 and M_2, we first inferred their standardized residuals[24]. We took the significance level of 0.01 and we then filtered out the candidate models whose standardized residuals were not normal at that level. We used the Anderson-Darling test of normality[25]. Table 6.6 shows the number of candidate models, in percent, which met the normality requirement of their standardized residuals.

Investigating the information provided in Table 6.6, we see that the number of candidate models with normal residuals are just a few for projects HSQLDB and Jameleon. In order to describe the reason of this phenomenon, we investigated descriptive statistics of the transformed data sets reported in Table 6.4. We observed that for projects HSQLDB and Jameleon, the kurtosis of the transformed data sets of low-level and high-level changes are bigger than 4. We know that the normal distribution of $\mathrm{N}\left(\mu, \sigma^2\right)$ has the kurtosis of 3 and greater values suggest that the transformed data sets might be no longer normal. This is confirmed by the p-values of the Jarque-Beta test of normality which is given in Table 6.3. Although, for both of HSQLDB and Jameleon, the normality of transformed data sets are rejected at the significance level of 0.01, the p-value for HSQLDB is quite smaller than the p-value of the Jameleon.

[24] Please see Section 6.1.4.3 to see how standardized residuals are computed for mixed ARMA-GARCH models.

[25] The Anderson-Darling test is shown to be more powerful in comparison to other alternative tests of normality [Razali and Wah, 2011].

Moreover, for HSQLDB, the kurtosis of the transformed series of low-level and high-level changes are approximately 4.7 and 4.8 while for Jameleon they are 4.3 and 4.2 respectively. Therefore, more candidate models with normal residuals are reported for Jameleon than HSQLDB. Figure 6.5 shows the box-plot of the kurtosis of standardized residuals for all candidate models of both low-level and high-level changes of the HSQLDB project. As we see, just few models in the set of mixed ARMA-GARCH models (M_2) have kurtosis near to 3 (kurtosis value of the normal distribution) and their normality is not rejected by the Anderson-Darling test. We conclude that far from the normality of the transformed times series have deteriorating effect on the number of the candidate models which have normal standardized residuals. Moreover, as Figure 6.5 shows, having a GARCH variance instead of a constant variance in ARMA models, i.e. having mixed ARMA-GARCH models, provides more flexibility that causes more candidate models have standardized residuals with near normal distribution.

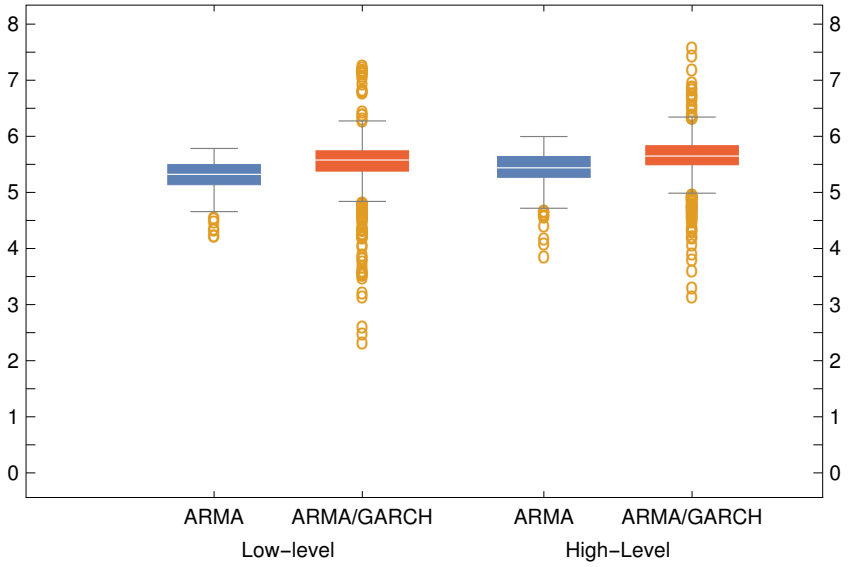


Figure 6.5: Project HSQLDB - Kurtosis box-plot of the standardized residuals of candidate models (the y-axis is limited to 8).

Having the candidate models with normal standard residuals, the next step is to select the most suitable model that can capture the dynamics of the evolution. As we described in Section S2, Akaike information criterion (AIC) can be used to select the best model [Akaike, 1974]. AIC is defined by[26]:

$$\mathrm{AIC} = -2\ln(\Lambda) + 2k \tag{6.18}$$

[26] In some literature, the value of AIC given in (6.18) is divided by the number of observations, e.g. [Tsay, 2005, Kirchgässner and Wolters, 2007]. This will not harm, since the smaller of two values will not be affected by such a rescaling [Enders, 2010].

in which Λ is the maximum of the likelihood function for the estimated model and k is the number of the parameters in the estimated model. AIC can be used to compare any models that are fitted to the same data and it does not need to test any hypothesis in this regard. In contrast, other model comparison tests such as the Lagrange multiplier or the Wald tests, should be just used to compare hierarchically nested models. AIC belongs to likelihood based assessment of fitness methods which penalize the complexity (number of parameters) of models. These methods differ in the way they penalize the complexity of models. Another test in this class which is frequently used in the Bayesian information criterion (BIC) [Schwarz, 1978]. We will compare the results of our analysis using BIC in Section 6.5.2 and we show that the models which are selected based on BIC will fail to pass the diagnostic step since they show some patterns in their residuals.

When comparing models using AIC, the model with lowest AIC value is regarded to be better. AIC does not provide any information whether other aspects of the fitted models are appropriate or not and they have to be tested separately. We sorted the candidate models, whose residuals were normal, with regard to their calculated AIC values. For each of the sample projects, we progressively selected the models with the lowest AIC. If the model was adequate regarding the absence of the serial correlation and absence of any observable pattern in its residuals, then the model was selected as the most appropriate model to capture the evolution, otherwise the next model with lower AIC value was tested and the procedure repeated. For mixed ARMA-GARCH models, the absence of serial correlation in the squared residuals was also considered. In practice we just needed to test very few models in the beginning of our sorted list (most of the times the first model), since they met all the diagnostic requirements.

Table 6.7 provides the information about the selected models which best capture the dynamics of the evolution at the low-level and high-level changes. For projects DataVision and Struts no mixed ARMA-GARCH model were considered since they showed no heteroscedasticity effect using the Engle's ARCH test. Moreover, there is no best ARMA model reported for the high-level changes of the project Maven, since the estimated candidate models did not pass the diagnostic step.

In order to test that the correlation effect is properly captured by the selected models, we used the Ljung-Box Q-test (see Section 6.2.2). The test statistics is given by (6.17). As we discussed earlier, the power of the test is affected by the number of lags, h, used in the test. In this regards $h = \ln(n)$, $h = 10$ and $h = \min(n, 20)$ were suggested in different literature (n is the length of observations) [Tsay, 2005, Hyndman and Athanasopoulos, 2013, MathWorks Company, 2013].

According to Table 4.3, $5 < \ln(n) < 7$ for all projects except the small project of DataVision. Due to the previous discussions, we decided to test the correlation effect at lags 5, 10, 15 and 20. Tables 6.10 and 6.11 provide the p-values of the Ljung-Box test on standardized residuals as well as squared standardized residuals of the selected models. As shown, the Ljung-Box Q-test does not show any violation of the absence of correlation for the residuals of the selected best models. The only exception is the residuals of the best ARMA model for the Struts project[27]. We will cover this issue

[27] Please remind that for all projects except DataVisoin and Struts, the appropriate models are mixed

Project	Selected Model	Low-level Changes		High-Level Changes	
		Model Degree	AIC	Model Degree	AIC
ASM	ARMA	(19,20)	612.143	(17,11)	602.390
	ARMA-GARCH	(17,20)-(2,1)	609.423	(19,18)-(0,2)	592.601
CheckStyle	ARMA	(4,13)	2508.046	(13,14)	2426.410
	ARMA-GARCH	(17,17)-(2,1)	2468.285	(20,22)-(2,2)	2382.077
DataVision	ARMA	(9,14)	23.157	(9,10)	44.682
	ARMA-GARCH	Not Considered		Not Considered	
FreeMarker	ARMA	(19,18)	963.812	(25,21)	979.089
	ARMA-GARCH	(20,25)-(0,2)	959.509	(19,23)-(1,2)	969.024
Jameleon	ARMA	(7,17)	929.157	(18,15)	945.673
	ARMA-GARCH	(13,21)-(2,2)	923.938	(24,25)-(2,2)	925.236
JFreeChart	ARMA	(19,17)	939.431	(17,15)	928.585
	ARMA-GARCH	(19,17)-(2,1)	922.540	(16,20)-(1,2)	921.799
Maven	ARMA	(5,21)	2119.262	-	-
	ARMA-GARCH	(23,19)-(2,1)	2114.048	(6,18)-(0,1)	2133.804
Struts	ARMA	(2,11)	2079.437	(6,24)	2063.681
	ARMA-GARCH	Not Considered		Not Considered	

Table 6.7: All projects - Selected time series model.

in detail in Section 6.3.1, but for now we generally conclude that the residuals of the selected best models do not show correlation.

Before closing this section, we should be sure that the estimated ARMA models are weakly stationary and invertible (see Section 6.1.3.1). For this, the roots of the associated polynomials of the estimated model should lay out side of the unit circle. This requirement is met by most of software packages which can be used to estimate the parameters of the proposed time series models, such as Matlab and R, since they mostly use constraint optimization methods during parameter estimation of the models through the maximum likelihood estimation method. The same strategy also applies on the constraints of GARCH models (see Section 6.1.3.2).

6.3 Assessing the Time Series Models

Earlier in Section 6.2.2, based on the transformed data, we employed ARMA-GARCH models for all projects except DataVision and Struts. There, additionally we considered ARMA models on those projects to see if simpler ARMA models are satisfactorily enough to handle the dynamics of the design model evolution. In this section we compare these

ARMA-GARCH models and for these two projects ARMA models are considered to be appropriate.

two classes of models and we investigate their performance and their properties against each other. First, in Section 6.3.1, we investigate how well each of these two classes can properly capture the dynamics of the evolution. Later in Section 6.3.2 we investigate the forecasting performance of the proposed ARMA and ARMA-GRACH models.

6.3.1 Comparing ARMA and ARMA-GARCH Models

In section 6.2.3 we showed that the best ARMA-GARCH models of all projects with heteroscedasticity effect have successfully passed the diagnostic tests of having normal residuals and their residuals did not show any correlation.

Moreover, for those projects we showed that the heteroscedasticity effect is mostly present in the first lag or the very first few lags. The interesting research question was that if simpler ARMA models can handle the heteroscedasticity effect and if they successfully pass the diagnostic steps. For this purpose, here we additionally considered ARMA models for those projects. Here, we compare the selected best ARMA models of those projects with their mixed ARMA-GARCH counterpart models.

As shown in Tables 6.10 and 6.11, for low-level changes of the Maven project, the null hypothesis of the Ljung-Box Q-test for the residuals is rejected at significance level of 0.05 for lags 5 and 10, indicating that some correlation between residuals exist. Since the null hypothesis is not rejected for lags of 15 and 20, we do not see any reason to disregard the usability of a simpler ARMA model in this case. This belief was also supported when we visually inspected the ACF of the residuals. However, we should also mention that the best ARMA-GARCH model for the low-level change of Maven is superior in this regard, removing all correlation in the residuals.

A similar story happens for the squared residuals of the best ARMA model when we consider high-level changes of the CheckStyle project. The null hypothesis of the Ljung-Box Q-test was rejected at lags 10, 15 and 20 at the significance level of 0.05. Visually inspecting the ACF of the squared residuals revealed that there is significant peak at the 10th lag which causes the null hypothesis of the test be rejected. Similarly, the mixed ARMA-GARCH model is more superior in this case.

Before closing this section, we go back to the project Struts for which we did not see any ARCH effect in low-level and high-level changes (see Section 6.2.2 and Table 6.5). As we saw in Section 6.2.3, the residuals for the best ARMA model of low-level changes show correlation using the Ljung-Box Q-test for all considered lags indicating the chosen ARMA model has correlation in its residuals. Visually investigating the ACF of the residuals, we saw that there are significant (but quite small) correlation in the first three lags causing the null hypothesis of the test be rejected. As we discussed earlier, this does not much degrade the appropriateness of the selected best ARMA model. Moreover, we additionally investigated the ability of the mixed ARMA-GARCH models on the low-level changes of the project Struts. The residuals of the selected best ARMA-GARCH model showed no correlation using the Ljung-Box Q-test.

Since the ARCH effect is just more significant in the first lag or the very first few lags of the transformed data sets, we can generally conclude that simpler ARMA models can be used more or less in practice to capture the dynamics of the evolution in design

models of the Java software systems. But we insist that mixed ARMA-GARCH models are the right and more appropriate time series models to mathematically model the evolution.

6.3.2 Forecasting Performance of the Time Series Models

The forecasting of a time series model is a topic of interest in many practical applications and in this section we investigate the forecasting performance of the proposed time series models.

Regarding the theory of time series (see Section 6.1), correctly performing the steps mentioned in the methodology of time series is sufficient for a time series model to be adequate for fulfilling the theoretical requirements and no further step is required. But in practice, it would be quite useful if we study the forecasting performance of the models. Although forecasting the evolution was not the main motivation behind this research work, we believe that it would be enlightening for other researchers that we devote this section to the study of forecasting performance of the selected time series model. In this regard, in Section 6.3.2.1, we investigate the forecasting accuracies of the proposed time series models based on the out of sampling technique. Later in Section 6.3.2.2, we compare the forecasting performance of the proposed ARMA and mixed ARMA-GARCH models.

6.3.2.1 Accuracies of Forecasts

As we discussed earlier in Section 6.1.5, due to high volatility in the measured low-level and high-level changes, the proper way of interpreting the forecasts' accuracy is by considering the magnitude of changes. In this regard the normalized mean squared errors will be more appropriate in our case (Equation (6.15)).

Based on our hold-out set (see Section 6.2), we used six step forecast of the selected time series models as presented in Table 6.7. The forecasts were transformed back into the original form by reversing the steps of of the BCDM transformation described in Section 6.2.1. We used out of sampling technique, i.e. we compared the forecasts with the actual observations in the corresponding hold-out sets using the normalized mean squared error (NMSE) given by (6.15). Equation (6.15) needs the maximum of the observed values in the base-set. The computed errors are reported in column "NMSE (Max)" of the table. The errors are quite low, both for low-level and high-level changes.

Computing the NMSE using the maximum of the observations is quite legitimate, since the measurements of the low-level and high-level changes are of high accuracy (see Section 6.5) and the calculated maximum values are not due to measurement errors but are due to those commits to the repositories which contain enormous amount of changes. But one might argue that the maximum and very big observations are not quite frequent and what would be the NMSE values if we ignore them. In this regard, we tried to exclude big values out of our observations and see what the NMSE values will be. We used the k-mean clustering algorithm [Gan et al., 2007] to detect not-frequent very big changes. For low-level and high-level changes of all projects, the ratio

Project	Selected Model	Low-level Changes		High-Level Changes	
		NMSE (Max)	**NMSE (Q98)**	**NMSE (Max)**	**NMSE (Q98)**
ASM	ARMA	2.0621×10^{-3}	9.8670×10^{-2}	4.2223×10^{-3}	9.9263×10^{-2}
	ARMA-GARCH	2.0640×10^{-3}	9.8760×10^{-2}	4.2185×10^{-3}	9.9174×10^{-2}
CheckStyle	ARMA	4.2116×10^{-6}	8.0324×10^{-4}	5.8528×10^{-6}	1.7497×10^{-3}
	ARMA-GARCH	2.9310×10^{-6}	5.5901×10^{-4}	2.6559×10^{-6}	7.9398×10^{-4}
DataVision	ARMA	2.9413×10^{-4}	2.9413×10^{-4}	2.1748×10^{-4}	2.1748×10^{-4}
	ARMA-GARCH	Not Considered		Not Considered	
FreeMarker	ARMA	9.9039×10^{-5}	2.0454×10^{-3}	6.3575×10^{-4}	6.1532×10^{-3}
	ARMA-GARCH	6.1144×10^{-5}	1.2628×10^{-3}	1.9749×10^{-4}	1.9114×10^{-3}
Jameleon	ARMA	6.5411×10^{-5}	1.1405×10^{-2}	8.0775×10^{-5}	1.2845×10^{-2}
	ARMA-GARCH	1.0002×10^{-4}	1.7439×10^{-2}	1.9526×10^{-4}	3.1051×10^{-2}
JFreeChart	ARMA	9.3903×10^{-4}	5.5625×10^{-3}	3.2103×10^{-4}	2.8284×10^{-3}
	ARMA-GARCH	2.2280×10^{-4}	1.3198×10^{-3}	4.3750×10^{-4}	3.8546×10^{-3}
Maven	ARMA	3.3037×10^{-3}	2.6373×10^{-1}	-	-
	ARMA-GARCH	3.2770×10^{-3}	2.6160×10^{-1}	2.6088×10^{-2}	1.9448×10^{-0}
Struts	ARMA	8.2014×10^{-4}	1.2269×10^{-2}	1.6722×10^{-3}	1.9467×10^{-2}
	ARMA-GARCH	Not Considered		Not Considered	

Table 6.8: Normalized mean squared error of forecasts for all projects.

of the number of the clustered outliers to the total number of observations were typically less than 2%. Therefore we used the 0.98-quantile (Q98) of the data instead of their maximum in (6.15). The "NMSE (Q98)" columns in Table 6.8, show the NMSE values calculated using Q98 instead of the maximum. In either of the cases, the calculated NMSE are quite small (typically few percent) and we conclude that the selected time series models are providing acceptable forecasts of their corresponding hold-out sets.

6.3.2.2 Comparing Accuracies of Forecasts

In practice it is quite common, as in this work, that we have different competing models for forecasting of the same observations. In such situations we are interested to compare the predictive performance of two or more models, weather one is performing better than the others or not[28].

[28] This notion is different from the fact that in some situations, we have two hierarchically nested models in which one is simpler than the other and we are interested to see if the simpler model is capable of sufficiently accounting the relevant patterns and factors in the data. We call the simpler (restricted) model "hierarchically nested" in the complex (unrestricted) model if the complex one can be obtained by adding some parameters to the simple one, e.g. ARMA $(2, 3)$ is hierarchically nested in ARMA $(2, 4)$ − GARCH $(0, 1)$ but not in ARMA $(2, 2)$.

Although adding more parameters to a model causes higher values for the likelihood function, but in a point there will be no significant improvement in the ability of the model to better capture the characteristics of the given data set. The likelihood ratio, Lagrange multiplier and Wald tests are frequently used in this regard which are asymptotically equivalent [Greene, 2002, Hamilton, 1994].

In our case, just regarding the predictive performance of the selected time series models, we are interested to see if ARMA and mixed ARMA-GARCH models are performing equally good or not. In other words, despite the fact that the residuals of the selected ARMA models might be correlated and not fully capture the dynamics of the data, we will study their predictive performance with the more suitable mixed ARMA-GARCH models. This is consistent with our previous results in Sections 6.2.2 and 6.3.1 that we showed the ARCH effect is not persistent over a long period and therefore a simpler ARMA models might forecast as good as a more complex ARMA-GARCH model.

To do a comparative analysis, suppose that T actual observations are given by $\{x_t\}_{t=1}^{T}$ and the forecasts of two competing models are denoted by $\{\hat{x}_{1,t}\}_{t=1}^{T}$ and $\{\hat{x}_{2,t}\}_{t=1}^{T}$. Let $\{e_{i,t}\}_{t=1}^{T}$; $i = 1, 2$, be the corresponding forecasts errors with $e_{i,t} = \hat{x}_{i,t} - x_{i,t}$. A function g is called a *loss function* if it is a function of the forecast errors, i.e. $g(e_{i,t})$, and takes value of zero when there is no error in the forecasts. Typically g is the square or the absolute functions, where it is referred to as *square-error loss function* and *absolute error loss function* respectively. The *loss differential* of two forecasts is defined as $d_t = g(e_{1,t}) - g(e_{2,t})$; $t = 1, \ldots, T$.

In the case that the loss differential of two forecasts has zero expectation, i.e. $\mathrm{E}[d_t] = 0$ or equivalently $\mathrm{E}[g(e_{1,t})] = \mathrm{E}[g(e_{2,t})]$, we can consider the two forecasts equally good. The *Diebold-Mariano* (DM) test [Diebold and Mariano, 1995] was proposed to assess the predictive accuracies of two forecasts. The test allows the forecasts to be non-zero in mean, non-Gaussian as well as correlated, which makes the test very useful in practice. Moreover, the test is a model-free test which, contrary to model-based tests, does not need that the models generating the forecasts to be available.

Harvey et al. [Harvey et al., 1997] modified the test to address the shortcoming of the original formulation for small sample sizes, a situation which is quite frequent in practice. The null and alternative hypotheses of the *modified Diebold-Mariano* (MDM) test are $\mathcal{H}_0 : \mathrm{E}[d_t] = 0$; $t = 1, \ldots, T$ and $\mathcal{H}_a : \mathrm{E}[d_t] \neq 0$ respectively. If $\bar{d} = \left(\sum_{t=1}^{T} d_t\right) / T$ is the mean and $\hat{V}\left(\bar{d}\right)$ is the estimated variance of the loss differentials, the statistics of the test is given by:

$$\mathrm{S}_{\mathrm{MDM}} = \sqrt{\frac{T + 1 - 2h + h\left(h-1\right)/T}{T}}\,\mathrm{S}_{\mathrm{DM}} \tag{6.19}$$

in which $\mathrm{S}_{\mathrm{DM}} = \bar{d}/\sqrt{\hat{V}\left(\bar{d}\right)}$ is the statistics of the original Diebold-Mariano test and h is the forecast horizon (h-step forecasts). The variance of the loss differentials can be obtained by:

$$\hat{V}\left(\bar{d}\right) = \frac{1}{T}\left(\gamma_0 + 2\sum_{k=1}^{h-1}\gamma_k\right)$$

In our case, the selected ARMA and mixed ARMA-GARCH models of low-level and high-level changes are not hierarchically nested and these tests cannot be applied.

in which γ_k is the kth autocovariance of d_t's, estimated by:

$$\hat{\gamma_k} = \frac{1}{T}\left(\sum_{t=k+1}^{T}\left(d_t - \bar{d}\right)\left(d_{t-k} - \bar{d}\right)\right)$$

The test's statistics, S_{MDM}, has the Student's t-distribution with $(n-1)$ degree of freedom.

In the case that there are few observations (as in our application scenario), the alternative powerful but simpler approach is to use the *sign test* [Diebold and Mariano, 1995]. The null hypothesis of test is that the median of the loss differentials are zero. The test statistics is:

$$S_{SIGN} = \sum_{t=1}^{T} I_+\left(d_t\right)$$

where

$$I_+\left(d_t\right) = \begin{cases} 1 & \text{if } d_t > 0 \\ 0 & \text{otherwise} \end{cases}$$

The test statistics has a binomial distribution with parameters of T as the number of trials, and $1/2$ as the success probability. The significance of the test will be obtained from the cumulative binomial distribution. Considering large samples, the test statistics will be $(S_{SIGN} - T/2)/\sqrt{T/4}$ which asymptotically has a standard normal distribution. It is also informative to mention that other, but less competitive, alternative tests are also available. For more information we refer the interested readers to [Mariano, 2002].

In our application scenario, for all projects except DataVision and Struts which did not show the heteroscedastic effect (see Section 6.2.2), we compared the forecast accuracies of the selected ARMA and mixed ARMA-GARCH models using the sign test. Table 6.9 presents the results of the test. As shown, the null hypothesis of the test is not rejected in all except two cases, at the significance level of 0.05. The null hypothesis is rejected for just low-level changes of CheckStyle and JFreeChart. Since the p-value of the sign test is more than 0.03 for both cases, we do not see any reason to conclude that the forecasting behavior of the ARMA and mixed ARMA-GARCH models are principally not different. Therefore, we conclude that the predictive performance of these models are almost the same and they are performing almost equally well. Although more complex ARMA-GARCH models can better handle the characteristics of the evolution both for low-level and high-level changes, they do not add much to the forecasting accuracies. This result is justified by the fact that the ARCH effect is significant at the first lag of the data and it is not persistent over a long period thus a mixed model will not be more successful (see Sections 6.2.2 and 6.3.1).

6.4 Simulation of Model Evolution

As mentioned earlier, we were interested to analyze and statistically model the evolution of software systems at the abstraction level of design models. In this regard, in

Project	Low-level Changes		High-Level Changes	
	p-value	**Stat.**	**p-value**	**Stat.**
ASM	6.8750×10^{-1}	2.0000×10^{0}	6.8750×10^{-1}	3.0000×10^{0}
CheckStyle	3.1250×10^{-2} $\diamond$	5.0000×10^{0}	6.8750×10^{-1}	2.0000×10^{0}
DataVision	Not Considered		Not Considered	
FreeMarker	2.1875×10^{-1}	4.0000×10^{0}	6.8750×10^{-1}	3.0000×10^{0}
Jameleon	6.8750×10^{-1}	3.0000×10^{0}	2.1875×10^{-1}	1.0000×10^{0}
JFreeChart	3.1250×10^{-2} $\diamond$	5.0000×10^{0}	6.8750×10^{-1}	3.0000×10^{0}
Maven	6.8750×10^{-1}	3.0000×10^{0}	Not Considered	
Struts	Not Considered		Not Considered	

Table 6.9: Comparison of the forecasting performance of ARMA and mixed ARMA-GARCH models ($\diamond$: Significant at 0.05).

Section 6.2, we showed that the models' evolution can be mathematically modeled and captured using ARMA and mixed ARMA-GARCH models. Such models can not only be used to mathematically formulate the models' evolution, but also to infer other useful information about the evolution and their properties. For example, in Section 6.3.2.1 we used the proposed time series models in order to forecast the amount of change in the hold-out sets of our observations with good accuracies.

In many practical applications, as here in this work, it is quite essential that the system under study can be simulated. The simulation principally imitates the operations and properties of the system and regenerates them in order to study and infer different aspects of interest, e.g. doing experiments, develop understanding, diagnose problems, preparation for changes, investment, prediction etc. [Banks, 1999, Shannon, 1998]. It is also known that the simulation and its approach is highly dependent on the scope, application purpose, overall and specific objectives of the study, requirements of the experiments and the needs of the decision makers [Law, 2009].

In this regard, the time series models of ARMA and mixed ARMA-GARCH models proposed to capture and model the evolution of design models can be used in different application scenarios such as understanding the evolution and its behavior over time including short or long time behavior, predicting the evolution, estimating the amount of changes and designing budget and work plans, etc. Here, in the first step, we try to keep ourselves out of any particular application of the proposed time series models and we try to address the question of how valid sample sequences of the proposed time series models can be properly generated. We also discuss general considerations for simulation. In the next step, we turn our attention into our application scenario of generating more realistic model histories for MDE tools which is the main motivation

behind this research.

Considering the above orientation, in Section 6.4.1 we address general considerations about simulation. In Section 6.4.2, we discuss how sample sequences of the proposed time series models can be generated. In Section 6.4.3, we address how the properly generated sample sequences can be used to generate more realistic model histories for MDE tools. In this regard, we address how the generator can be properly configured to generate more realistic model histories.

6.4.1 General Considerations for Simulation

When the outputs of simulations are to be analyzed, simulations are usually categorized into two categories of finite-horizon (terminating) and steady-state (non-terminating) simulations [Alexopoulos and Kim, 2005, Law, 2007b]. In finite-horizon simulations, there is a condition or an event which limits the simulation run or terminates it. Typical conditions or events are: the ones that specify when the system is cleaned out, i.e. finished its processing, the points where there is no useful information afterward and the ones which are set by domain experts or management [Law, 2007b]. However, in steady-state simulations, the long time behavior and properties of a system are of special interests; in such a simulation, the system is in an equilibrium in the long run.

In any simulation, the initial conditions used for the starting the simulation has impacts on the behavior of the simulation. Since in steady-state simulations the long-run behavior of the system is the main concern, often it is very favorable that the effect of initial conditions is removed, bypassed, minimized or even that the system can be simulated in a steady state from the beginning. This issue is usually referred to as the problem of the initial transient in the simulation literature [Gafarian et al., 1976].

Various techniques for detecting when a simulation reaches the steady state, i.e. equilibrium, after its warm up period have been proposed so far. Gafarian et al. [Gafarian et al., 1978] have reviewed different commonly used methods for detecting the steady state in computer simulations and evaluated them on a queue system. Additionally, different start up policies, i.e. methods of selecting the initial conditions, as well as truncation rules have been proposed in order to bypass or minimize the start up period of a simulation for reaching its steady state. For instance different truncation rules in which one discards the first n points in the generated sample or the selection policies for initial conditions in which one chooses the initial conditions close to some values, e.g. mean, are frequently used in practice.

Wilson and Pritsker did a survey on the research related to the simulation start up problem and different selection policies and truncation rules [Wilson and Pritsker, 1978b]. Moreover, they evaluated the start up policies and truncation rules on two Markovian queue systems in [Wilson and Pritsker, 1978a]. In the end, based on simulation requirements and specified research questions and goals, the interested reader can find different techniques and methods for analyzing the simulation outputs in [Alexopoulos and Seila, 1998, Law, 1983, 2007b, Seila, 1992].

6.4.2 Simulating Sequences of the Proposed Time Series Model

In order to properly run a simulation based on the proposed time series methods in Section 6.2.2, it is essential that their sample sequences can be properly generated. Sample sequence generation of the ARMA and the mixed ARMA-GARCH models are done when parameters of equations (6.4) as well as (6.12), (6.13) are respectively estimated[29]. In this regard, two issues should be taken into account. First, principally, the formulations require that sequences of independent and identically distributed (iid) numbers with the normal distribution can be properly generated. Second, the sample sequence generation of ARMA and mixed ARMA-GARCH models require that some initial points are employed in the beginning which their impacts on the generated sequences might be regarded important based on the aims and goals of the simulation.

In this section we first address the question that how sequences of iid numbers of the normal distribution are generated. Then, we discuss different strategies in selecting of initial points in the simulation of the proposed time series models and their implications.

6.4.2.1 Random Variates of the Normal Distribution

As we thoroughly explained in Section 5.3.1, methods for generating iid random numbers of a given non-uniform distribution are usually referred to as random variate generation (RVG) methods while the generation methods of iid numbers of the uniform distribution of $\mathrm{U}[0,1]$ are reffed to as random number generation (RNG) methods. Principally, RNG methods are first employed to generate iid sequences of $\mathrm{U}[0,1]$ and the generated numbers are then used to generate random variates of a given distribution using different techniques such as the acceptance-rejection and the transformation methods [Banks, 1998, Devroye, 1986].

The generation of true iid uniformly distributed random numbers is still actively studied and there have been many RNG methods available. Two methods of Linear Congruential and Combined Linear Congruential [Banks et al., 2010, Knuth, 1998] are popular and are frequently used in practice. We should note that better methods are also available. Assuming that a suitable RNG method is available, the following algorithm, called the Polar method, simultaneously generates two iid random variates of the normal distribution $\mathrm{N}\left(\mu, \sigma^2\right)$.

The formal proof of the algorithm is given in [Knuth, 1998]. Although the Polar method is widely used due to its simplicity and speed, more robust RVG methods for the normal distribution are also available. In this regard, Thomas and Luk have provided a good survey in [Thomas et al., 2007].

6.4.2.2 Initial Conditions in the Simulation of the Time Series Models

As we talked earlier in Section 6.4.1, the initial conditions used in generation of the proposed time series models have impacts on the behavior of the generated sample

[29] The same applies to sample generation of the ARCH and GARCH models given by (6.7) and (6.9) respectively.

Algorithm 14: The Polar method for generating two iid random variates of the normal distribution $\mathrm{N}\left(0, \sigma^2\right)$.

repeat
  Generate U_1 and U_2, two iid random numbers of the uniform distribution $\mathrm{U}\left[0,1\right]$;
  $V_1 \leftarrow 2\,U_1 - 1$; $V_2 \leftarrow 2\,U_2 - 1$;
  $S \leftarrow V_1^2 + V_2^2$;
until $S \leq 1$;
$X_1 \leftarrow \mu + \sigma\, V_1 \sqrt{-2\frac{\ln(S)}{S}}$; $X_2 \leftarrow \mu + \sigma\, V_2 \sqrt{-2\frac{\ln(S)}{S}}$;
return (X_1, X_2)

sequences specially when long term behavior and properties of the simulated system is of special interest. In our application domain of generating more realistic model histories, it is not necessary to have a non-terminating simulation and the impact of the initial conditions are principally desired since we want to replicate the evolution of each project quite closely in the generated test models (see Sections 6.4.3 and 6.5.5). However, since one of our aims is to provide a general framework for analysis, forecasting and simulation of design models' evolution, other researchers might need to eliminate the effect of the initial conditions in the warm up period of their simulations. In this section we address the important issues in this regard and we leave the application of them to the interested readers.

For an $\mathrm{ARMA}\left(p_{_A}, q_{_A}\right)$ model, $p = p_{_A}$ and for an $\mathrm{ARMA}\left(p_{_A}, q_{_A}\right) - \mathrm{GARCH}\left(p_{_G}, q_{_G}\right)$ model, $p = \max\left(p_{_A}, p_{_G}\right)$ initial points should be chosen to start the simulation. As we talked earlier, there are methods for detecting the steady state in a simulation which are covered in [Gafarian et al., 1978]. After detecting the steady state one can discard the warm up period of the simulation and use the remaining for the output analysis. Such methods are generally refereed to as the truncation methods and are covered in detail in [Wilson and Pritsker, 1978b,a]. They are frequently used in practice, e.g. in the R package of "fArma".

As we also noted earlier, another approach is to use the initial conditions which minimize or ideally remove the warm up period. In this regard, McLeod and Hipel [McLeod and Hipel, 1978, Hipel and McLeod, 1994] have developed three procedures. The base algorithm used for non-seasonal ARMA models is called WASIM-1, or Waterloo Simulation Procedure 1. WASIM-1 principally tries to use the pure MA representation of an ARMA model by the Wold's theorem. Let $\mathrm{ARMA}\left(p, q\right)$ model is given by (6.5), we have:

$$\tilde{x}_t = \frac{\Theta_q\left(\mathrm{B}\right)}{\Phi_p\left(\mathrm{B}\right)}\varepsilon_t = \Psi\left(\mathrm{B}\right) = \left(\psi_0 + \psi_1\,\mathrm{B} + \psi_2\,\mathrm{B}^2 + \cdots\right)\varepsilon_t \tag{6.20}$$

where $\psi_0 = 1$. In the case that an AR operator exists, the $\Psi\left(\mathrm{B}\right)$ in (6.20) is an infinite

series which is then approximated by:

$$\Psi(\mathrm{B}) \approx \left(\psi_0 + \psi_1 \mathrm{B} + \psi_2 \mathrm{B}^2 + \cdots + \psi_{q'} \mathrm{B}^{q'}\right) \varepsilon_t \tag{6.21}$$

provided that $\psi_{q'+1}, \psi_{q'+2}, \ldots$ are negligible. The number of participating terms, i.e. q', is chosen large enough in such a way that the difference of the theoretical variance of the ARMA model, γ_0, from the approximation terms in (6.21) are less than the given error rate of err, i.e.

$$\gamma_0 - \sum_{i=0}^{q'} {\psi_i}^2 < err$$

The theoretical variance of γ_0 can be calculated based on the pure MA presentation using the algorithm given in [McLeod, 1975].

Now to simulate a sample sequence of length k, we first generate a sequence of white noise with length of $k + q'$ as $\varepsilon_{-q'+1} + \varepsilon_{-q'+2} + \ldots + \varepsilon_0 + \varepsilon_1 + \ldots + \varepsilon_k$ and then for $t = 1, \ldots, r = \max(p, q)$ we calculate $\tilde{x}_t$ as follows:

$$\tilde{x}_t = \varepsilon_t + \psi_1 \varepsilon_{t-1} + \psi_2 \varepsilon_{t-2} + \ldots + \psi_{q'} \varepsilon_{t-q'} \tag{6.22}$$

For generating the remaining terms of $\tilde{x}_t$; $t = r+1, r+2, \ldots, k$, we use Equation (6.4). It is shown that the effect of approximation by (6.21) can be kept low by using appropriate small values for err. For pure MA (q) models, the algorithm is exact.

Regrading initial points in mixed ARMA-GARCH models (and also GARCH models), to our best knowledge, there is no tailored algorithm to address the issue. Reviewing possible related literature, we found out that it might be be useful that the initial conditions for necessary terms of conditional variances in (6.13) are set to the unconditional variance of the model. Similarly, one can also use unconditional mean of the model as initial values in (6.12). These remedies are also practiced in [MathWorks Company, 2013].

6.4.3 Generating More Realistic Model Histories

Up to this point, we analyzed and mathematically formulated the evolution of software systems at the abstraction level of design models using time series. In the previous section we discussed how one can generate valid sample sequences of the proposed time series models and we discussed different consideration regarding simulation.

Here, we address the question how we can generate more realistic model histories for MDE tools, by simulating the evolution of design models In this regard, as discussed in Chapter 3, the SiDiff Model Generator (SMG) can be used. When the tool is properly configured, it can simulate the evolution in the generated model histories. We show how proper configurations are obtained by valid sample sequences of the proposed time series model and how test model histories are generated in practice.

As discussed in Section 3.3.4, SMG has two interpretation modes in order to control the frequencies or distributions of the created differences. In the Literal Interpretation

Mode the specified frequencies of operations were treated as exact values in which the resulting model should exactly experience the specified number of operation applied to it. This mode was mostly appropriate for exactly controlling the quantitative properties of the generated models.

In the Stochastic Interpretation Mode the specified numbers of operations were treated as probabilities. It was assumed that the provided set of edit operations (see Figure 3.3) has a probability density function and the specified frequency of an operation is then treated as the probability of that operation. Therefore, the true size of the difference, i.e. total number of operations which has to be applied, has to be explicitly specified. In contrast, the size of a difference is implicitly given in the literal interpretation mode, which is the sum of the specified operations in the configuration file.

In the case of generating model histories, the true amount of differences should be specified for each two subsequent models which are revisions of each other. This is done through the knowledge gained in this chapter as we modeled the total amount of changes between subsequent revisions using time series. Since the stochastic interpretation mode is stochastic by nature and is based on the probability of each operation in the configuration file, the generated frequencies of the specified operations approximately exhibit the specified probabilities.

To produce model histories, we are required to know how big is the difference size of each two subsequent revisions. In other words, we had to specify how many edit operations have to be applied from revision r_i to r_{i+1}. In order to answer this question, in Chapter 4, we modeled the evolution of real software systems at the abstraction level of design models by calculating the changes which were applied between them. In Section 6.2.2, we showed that the evolution of real design models can be mathematically modeled using ARMA and mixed ARMA-GARCH time series models. To regenerate the evolution of design models we should first generate appropriate sequences of the proposed time series models and then employ them in the stochastic interpretation mode as discussed in what follows.

Sequences of the Time Series Models As shown in Section 6.2.2, the ARMA and mixed ARMA-GARCH time series models presented in Table 6.7 are the most suitable models for capturing the true evolution of our sample software systems. For each project, the proposed time series model can be used to generate as many sample time series sequences as needed, all exhibiting the same stochastic properties of the real evolution of that project. This is accomplished by the techniques discussed in Section 6.4.2.

However, since we aimed at producing more realistic model histories in which the true evolution is properly replicated, our main concern is to simulate the evolution of each project p in a way that the generated histories resemble the evolution of p. Since in our application scenario, MDE tools are the final recipient of the generated model histories, the lengths of the required model histories are the terminating condition for the simulation of evolution and we are facing a finite-horizon simulation (see Secion 6.4.1). In the case the simulated evolution should resemble the properties of p from

the beginning phase of development, the initial conditions should be taken from the beginning of measured sets of low-level and high-level changes, i.e. observation sets of L_p and H_p (see Section 6.2). Similarly, when the simulation should resemble the evolution beyond the final revision of the project, the initial conditions should be taken from the end of L_p and H_p sets. As we mentioned in Section 6.4.2.2, another practical approach is using the unconditional mean and unconditional variance of the estimated time series model of project p, as the initial conditions for the simulation. As an example, Figures 6.6 and 6.7, each show three simulated sequences of length 259 (the number of revisions in the ASM project), from a time series model for the low-level changes of ASM. The initial conditions are set from the beginning and the end of the measured low-level changes respectively.

As shown in Figure 6.6, the three simulated sequences show many small changes as well as few big changes. The simulated sequences resemble the real observed changes for the ASM project as already shown in Figure 6.1. Similarly to the true evolution, the simulated sequences show quite erratic behavior in the beginning part of the simulation and as the simulation evolves, we see less erratic patterns. This is again consistent with what we observed in real evolution of ASM in Figure 6.1.

The erratic pattern in the beginning of the simulation in Figure 6.6 is due to the fact that we chose our initial points of the simulation from the beginning of the set of real observed changes which they were themselves very erratic. In contrast in Figure 6.7, the initial points are taken from the end of the set of real changes where ASM showed calm period in its changes. Apparently, the magnitude of the simulated sequences are much lower comparing to the case that we chose the beginning points as initial conditions.

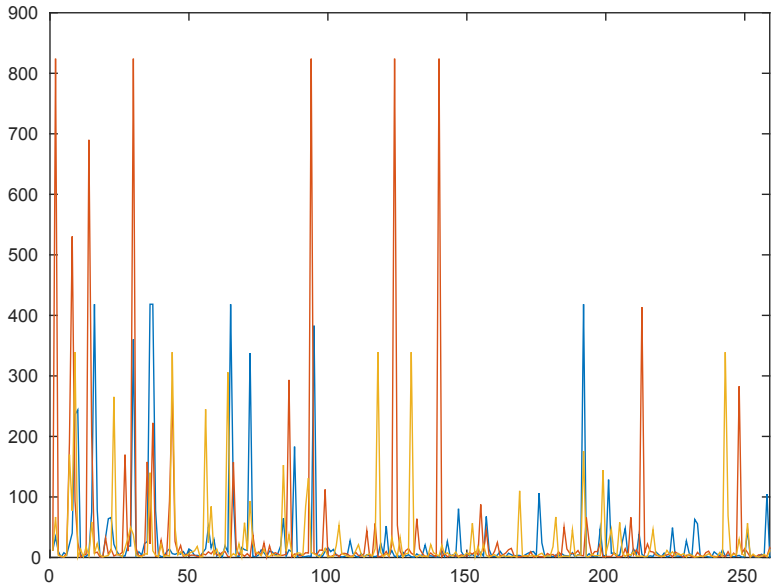


Figure 6.6: Project ASM - Three simulated changes using initial conditions from the beginning of measured low-level changes.

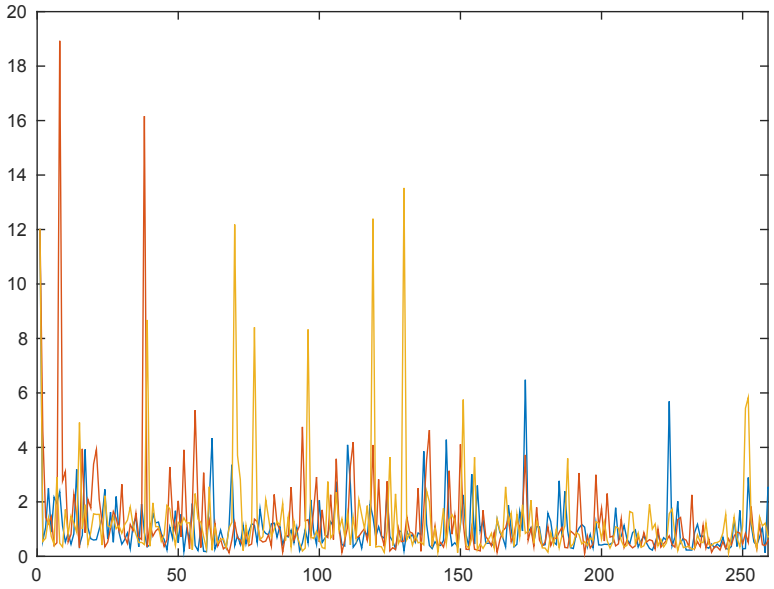


Figure 6.7: Project ASM - Three simulated changes using initial conditions from the end of measured low-level changes.

Applying the Generated Sequences For both of the low-level and high-level changes, the generated sequences of the proposed time series models specifies how many edit operations should be applied between each two subsequent revisions of r_i to r_{i+1} in total. This does not address how many (which portion) of this total number belongs to a specific edit operation that has to be applied between r_i and r_{i+1}. Since we modeled the evolution of the design models as the total number of edit operations applied between each two subsequent revisions, the problem of assigning a reasonable portion of the simulated evolution to each edit operation, can be addressed using two approaches.

The first approach considers the total frequencies of an edit operation which has been detected between each two subsequent revisions as the probability of applying that edit operation. The computed total frequencies are then used in order to assign a reasonable portion of simulated time series sequences using the Roulette Wheel selection technique [Reeves and Rowe, 2002, Michalewicz, 1996]. As explained in Section 3.4.2, in the Roulette Wheel technique, also known as the Fitness Proportionate Selection technique, the chance of selecting a edit operation is proportional to its assigned weight, which is the associated total frequency in this case. The bigger the total frequency, the higher the chance of the corresponding edit operation being selected and then applied between r_i and r_{i+1}.

The second approximate approach is based on the theoretical distributions of edit operations which we studied in Chapter 5. For each of the estimated distributions, the mean of the distribution can be used as the weight in the Roulette Wheel technique. This allows the portion of each edit operation to be proportional to the mean of its

corresponding distribution. This approach has the drawback that for a measured distribution which exhibits heavy skewness, its mean might not be quite a suitable indicator for its proportion.

Before closing this section, it should be reminded (see Sections 3.3.5 and 3.4) that the creation of an "edit step" additionally requires that a Context is selected and suitable parameters of the edit operation are either selected or generated. In other words, it should be decided where the edit operation is going to be applied and which parameters it should take. Although these issues require in-depth analyses on their own and are not considered in this dissertation, various controlling mechanisms and configuration policies which are introduced in Section 3.4.2 can be used in this regard. They allow fine tuning of SMG when such analyses are available or when the domain expert decides to alter the behavior of the generation process to his/her best knowledge.

6.5 Threats to the Validity of Analyses

This section is devoted to the investigation of the threats to the validity of the analysis done in this chapter. Although some of the discussions are principally the same ones[30] that we earlier discussed in Chapter 5, for the sake of self containment of the chapter, we review them again here. In this regard, we consider the accuracy of our measurements, best model selection strategy, forecasting performance of the proposed time series models as well as external validity of this work.

6.5.1 Accuracy in the Measurement of Changes

The first threat to the validity of our work is that how accurate is the change detection pipeline in Figure 4.1 and how accurate low-level and high-level changes are computed.

As we discussed in Chapter 4, low-level changes are computed through model comparison algorithms. These algorithms might deliver sub-optimal or wrong differences when there are no persistent identifiers at model elements. In our analysis due to the absence of persistent identifiers, we employed similarity based model comparison algorithms. For low-level changes, Wenzel [Wenzel, 2010] has computed the error rates in the computed differences of the similarity based algorithms, implemented in the SiDiff model differencing engine, for class diagrams. The total error rates were typically under 2% which would not have any significant effect on the results of our analysis.

Considering high-level changes, the computed differences are guaranteed to be correct in the case that persistent identifiers are used for matching the corresponding elements between models [Kehrer et al., 2011]. Since the similarity comparison technique is used, there might be incorrect matches between elements of two models which leads to false negatives, i.e. the changes which are applied but were not detected. If all correspondences

[30] Principally accuracy in the measurement of changes between design models and the external validity of our research are the same, since the statistical analysis of Chapter 5 and the time series analysis of this chapter used the same measurement techniques and the same sample sets of Java software systems (Chapter 4).

are correctly detected, the semantic lifting engine (see Figure 4.1) will group whole low-level changes into high-level changes and there will be no low-level changes remained ungrouped. We computed the rate of ungrouped low-level changes at less than 0.3%. This shows that both of the difference derivation and the semantic lifting engines have performed quite well and the results are not twisted.

Moreover, the time series approach used in the analysis of the changes is not affected by any possible inaccuracies in the measurement of changes. This is due to the fact that such possible errors will not influence the transformation strategy which was used to make the series weakly stationary. As long as the weakly stationary series is obtained, ARMA and mixed ARMA-GARCH models can be applied to the transformed data.

6.5.2 Best Model Selection Strategy

As discussed in Section 6.1.4.1, in step S5, selection of the best model in the set of the candidate models is done using the Akaike information criterion (AIC). As a competitive approach to AIC, the other frequently used method is the *Bayesian information criterion*[31] (BIC) [Schwarz, 1978] which is is defined by [Wei, 2006, Box et al., 2008]:

$$\mathrm{BIC} = -2\ln(\Lambda) + k\ln(n)$$

in which, similarly to AIC, Λ is the maximum of the likelihood function for the estimated model, k is the number of the parameters in the estimated model and n is the number of observations used in the model estimation.

It is shown that BIC tends to select more parsimonious models in comparison with AIC [Wei, 2006]. One issue of concern might be that why don't we choose more parsimonious models using BIC instead of AIC.

To answer this concern, we should note that it is shown that BIC might select models whose residuals might not be white noise [Enders, 2010]. Due to this, we additionally tried to select the best time series models from the available set of candidates, using BIC. Our experiences revealed that BIC selects the models whose residuals are not white noise. For example, Figure 6.8 shows the residuals of the BIC-selected best ARMA-GARCH model for high-level changes of the ASM project. As seen, there are observable patterns in the residuals of the selected model which causes the model be inappropriate in the diagnostic step (see step S4 in Section 6.1.4.1).

Considering the low-level and high-level changes of all of the sample projects, we checked if the BIC-selected models fulfill the diagnostic requirements or not. The results of our analysis revealed that AIC selects models much more appropriately than BIC in our case. Therefore we conclude that AIC should be used to select time series models which describe the evolution of design models.

6.5.3 Forecasting Performance of the Time Series Models

We studied the forecasting performance of the proposed time series models in Section 6.3.2. Here we discuss the threat to the validity of the forecasting performance of the

[31] Also known as the Schwartz's Bayesian information criterion.

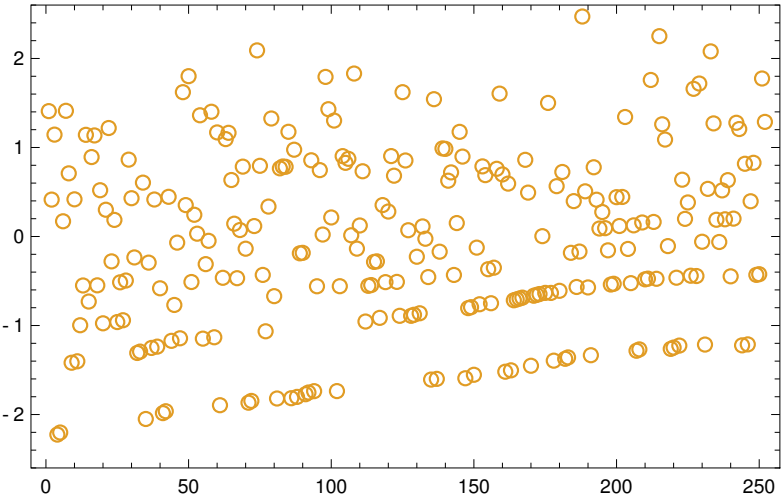


Figure 6.8: Residuals of the best model selected using BIC - High-level changes of ASM.

time series models.

One threat to the validity is that why we chose the hold-out set of length six and not more. In other words, how far can we satisfactorily forecast the future by such time series models. The first answer is that, our main motivation was to simulate the evolution process of the sample projects. In this regard the more observations are considered in the estimation of the time series models, the better fit can be achieved and consequently the more realistic simulation can be done afterward.

Second, it is shown that best optimum[32] forecast of the ARMA models tends to the unconditional mean of the process as the forecasting horizon increases [Kirchgässner and Wolters, 2007, Hamilton, 1994]. In general [Makridakis et al., 1998], in the case that there is no differencing used to make the series weakly stationary, i.e. having an ARIMA $(p, 0, q)$ process, the forecasts tend towards the unconditional mean of the process. If the series is mean-adjusted (see Section 6.1.3.1), the forecasts tend toward zero. When a differencing of degree one is applied to the data, i.e. having an ARIMA $(p, 1, q)$ process, then two cases arises. If there is no constant term in the model (i.e. the series is mean-adjusted), following our previous discussion about the ARIMA $(p, 0, q)$ process, the forecasts will converge to the last observation in the series. If there is a constant in the model, the forecasts will follow a linear trend whose slope is equal to the fitted value for that constant. Moreover, we know that the variance of forecast errors is a monotonically increasing function of the forecast horizon [Kirchgässner and Wolters, 2007, Hamilton, 1994]. Similarly, for the volatility of a GARCH process, it is shown that the forecasts of conditional variance will tend toward the unconditional variance of the series [Engle and Bollerslev, 1986].

[32] The best optimum forecast is the forecast whose expected value of the squared forecasts errors is minimized [Montgomery et al., 2008, Kirchgässner and Wolters, 2007]

From the above discussion, it is clear that we cannot predict far in the future since the forecasts tend toward a specific value and the variance of errors will grow. Since in our data sets there is no cyclic or periodic effect, it is unrealistic to consider hold-out sets with bigger sizes.

Specifically in our data sets, the previous conclusion about the forecasting performance of the time series models can be discussed from another perspective as follows. As we discussed in Sections 6.1.2 and 6.1.3, a GARCH model has an ARMA representation and ARMA models were motivated from the Wold's theorem as a parsimonious representation. The assumptions on the ARMA models allows that an ARMA model can be written as an MA (∞), satisfying the Wold's representation. For our sample projects, the ACF of the data typically showed most significant correlation in the first or first few lags suggesting that a low order moving average model can be considered as a candidate to capture the evolution. This was also motivated when we used BIC instead of AIC in the selection of best time series model, although the BIC-selected models had residuals which were slightly twisted from the assumption of the uncorrelated white noise (see Section 6.5.2). Therefore, the most influencing term(s) in an MA model of our data sets was the last term or were very few latest terms, which causes the behavior of future forecasts be depended on the very few last disturbances and far forecasts are not realistic.

6.5.4 External Validity

The next threat to validity is that how well the results of this analysis are generalizable. This can be discussed from two points of views. First, how well the change extraction techniques discussed in Chapter 4 can be used to compare the models of other types than class diagrams. The pipeline of Figure 4.1 can be applied to other diagram types as long as the model matching mechanism and the difference derivation engine can be properly adapted to handle the model types of interest. In this work we have employed the SiDiff/SiLift model differencing and semantic lifting frameworks. SiDiff allows fine configuration for comparison of different model types and SiLift allows the proper computation of high-level changes.

The second aspect of generalization is that if the time series approach can be employed to model and simulate the evolution for other application domains. As we discussed in Section 6.1, the time series approach of ARMA, GRACH and mixed ARMA-GRACH models concerns the weakly stationary assumption of the observed series. Since it is very frequent that the series is not stationary, a proper transformation should be employed to make the series weakly stationary and afterward the time series models can be applied to the transformed series. As we saw in Section 6.2.1, our sample projects could be successfully transformed into weakly stationary ones using the three step transformation introduced there and we could model the evolution of all projects except HSQLDB.

Although the transformation made the series stationary for HSQLDB, the transformed series were not normally distributed and we showed that the kurtosis of the transformed series were almost 4.7, much exceeding the value of 3 which is the kurtosis of the normal distribution. This caused the residuals of the applied time series models

to be non-normal due to big values for their kurtosis. We conclude that although the BCDM transformation was quite successful in other projects, another possible transformation that can solve the non-normality issue for the HSQLDB project will most likely address the shortcoming.

One issue to consider is that our sample projects are typically medium sized open source Java systems. As preliminary studies show, the results of this chapter will most likely be extendable to big systems. Regarding the closed source software systems, it not quite clear if the computed changes have similar properties to ones that we discussed in this chapter, specially if a particular or a company-specific programming style is practiced. But since the application of time series models is subject to the assumption of stationarity, as we discussed earlier, in the case of a proper transformation the time series models of ARMA and ARMA-GARCH should be readily useful and can be further employed.

Considering other object-oriented programming languages like C++ or C#, since the concepts of object-orientation are quite similar, the approach can be applied without much effort. In this regard, the metamodel for class diagrams (shown in Figure 4.3) should be revised and the SiDiff/SiLift pipeline (Figure 4.1) for computation of differences should be accordingly adjusted.

6.5.5 Validity of the Simulation

The validity of a simulation boils down to the defined research questions, aims and goals of the simulation and how the outcomes of the simulation are going to be used. It is also well established that a simulation is just an approximation to the real system which simplifies its functionality and is a trade-off between reality and simplicity [Maria, 1997]. In this regard [Law, 2009] clearly states: "A simulation model of a complex system can only be an approximation to the actual system, no matter how much time and money is spent on model building. There is no such thing as absolute model validity, nor is it even desired. Indeed, a model is supposed to be an abstraction and simplification of reality.".

Regarding the simulation of the evolution done in Section 6.4, different issues have to be considered. If a steady-state, i.e. non-terminating, simulation of the proposed time series models is needed to analyze the long time behavior of the systems, then the effect of initial conditions should be be taken into account. In our use case of generating more realistic models histories for MDE tools (Section 6.4.3), the simulation would be a terminating simulation since the required length of histories is finite. The length of simulation is depending on different factors such as the features of the tool which should be evaluated, the aim and the purpose of the test models, the generation or the processing time of the test models, the memory considerations etc. More specifically, in the domain of model comparison and model versioning [Kolovos et al., 2009, Stephan and Cordy, 2013, Brosch et al., 2012], the generated model histories can be used to simulate different development phase of software systems. For instance, the development of a software system is usually more erratic in the beginning of the development phase when new features and functions are going to be implemented, and is less erratic and more

calm when the system in maintained afterward [Vasa et al., 2007b]. Therefore, the effect of initial conditions used to generate sequences of the proposed time series models is even desired.

Regarding the application of generated sequences of time series models by the SMG in Section 6.4.3, we observed that a reasonable portion of the total number of edit operations which should be applied between revisions of r_i and r_{i+1}, has to be assigned to each edit operation. In this regard, we assigned the total computed frequency of each edit operation in the life time of the project, as the weight for selecting that edit operation by the Roulette Wheel selection technique. This is justifiable since the total frequency of each edit operation in the life time of the project is a reasonable indicator of how frequent that operation is. More frequent operations have higher frequencies and more portion is assigned to them from revision r_i to r_{i+1}.

The much better approach is that if we could know how frequencies of each edit operation evolves over time with respect to the frequencies of other operations. In other words, it would be ideal if would could analyze the co-evolution of frequencies of all edit operations together over time from a mathematical point of view and replicate them. However, such in-depth analysis is not present at the moment and it can be a topic for further research. Moreover, it is less clear where edit operations have to be applied i.e. how a Context should be mathematically selected for applying an edit operation and what factors influence a Context to be modified in real life. The selection or generation of the required parameters for edit operations also have not been studied so far and can be a topic for further research. The above issues will influence the simulation of evolution from the perspective of generating model histories. However, as discussed earlier, the controlling mechanism of SMG will allow a domain expert to adjust the generation process to his/her best knowledge.

6.6 Related Works

In this chapter we focused on mathematically modeling the evolution of design models through time and in this regard we considered three time series models. The evolution was measured as the total number of difference metrics, done separately on low-level and high-level metrics, which occurred between subsequent revisions. As we saw in Chapter 5, the other aspect of the evolution was the frequencies of low-level and high-level edit operations applied between subsequent revisions of design models. There, we showed that six discrete distributions could be used to model the frequencies.

Principally the related works regarding the statistical analysis of the evolution which we investigated in Section 5.4, are regarded relevant to the results we presented in this chapter. We do not repeat them again here, rather we focus on the particular aspect of the evolution which considers the time. Therefore, the related works in this section focus on time-dependent aspects of evolution or other properties in software systems.

The following papers used time series analysis to answer research questions in the context of software maintenance and evolution. None of these paper addresses the topic of this chapter, i.e. how design models of software systems evolve over a long period of

time, what is the proper way for simulating the evolution and which issues should be considered when one simulates the evolution.

[Antoniol et al., 2001] presented a time series based method for monitoring and predicting the evolution of clones, i.e. duplicated or slightly different code fragments, in software systems. At first, clones were detected within a system based on software metrics, namely layout, size, control flow, function calls and couplings. In the next step the series of the average number of clones per function was modeled as time series. The approach was tested over 27 subsequent versions of mSQL. They were able to predict the number of clones in the next release accurately with this method.

[Fuentetaja and Bagert, 2002] applied time series analysis on growth data of released versions of software systems. The growth was measured based on the number of modules of the systems which were detrended using the first order difference. Their data set consists of two software systems. By showing the presence of long-term correlations in the data they were able to validate the third and eighth laws of software evolution[33]. Caution is advised when considering the presented results though, because the systems had less then 30 released versions, the base data was therefore limited.

[Herraiz et al., 2007b, Herraiz, 2008, 2009] used time series to analyze the growth of open source software systems based on the simple size metric. They analyzed daily size of FreeBSD and NetBSD kernels as well as PostgreSQL. The compared the regression and ARIMA models on the data. They concluded that time series analysis is better for forecasting the evolution of software projects.

[Kenmei et al., 2008] studied the frequencies of changes requests in the frame of software maintenance. They analyzed the number of changes requests per (kilo) lines of codes to identify trends in the patterns of change requests. Increasing trends indicate an increase in the change requests indicating problems in the system, decreasing trends indicate applications stability and maturity. They analyzed the times series of change requests in three large-scale open source software systems, namely Eclipse, Mozilla browser and JBoss. They used two snapshots of each system per month over a period of five years starting from January 2002. They forecast the change requests per (kilo) lines of code.

[Siy et al., 2008] were interested in non-numeric data in software repositories. They used the time series segmentation technique on the changes of three open source software systems in order to identify active developers. In time series segmentation some consecutive data points are considered as a single data point which represents that segment. The optimal segmentation obtained by dynamic programming. The changes or deltas between software revisions were defined in a broad sense such as code changes, check-in entries in CVS log of a source file or an email message about a commit. The deltas were put in item-sets in order to identify patterns of developer activities. In their work they did not work on design models of software systems. Additionally since some of data are combined together in order to form segments, details about the evolution are missing. This makes their approach not suitable for our fine-grained analysis in order to simulate the evolution of design models.

[33] Presented in [Lehman, 1996].

[Raja et al., 2009] used time series approach to predict defects in histories of software systems. They used monthly defect reports for eight open source projects and built up time series models to predict and analyze software defects. The time period considered for the analysis varied between 66 to 90 months. They used the last four months as the hold-out set in order to validate the accuracies of forecasts. The results show that ARIMA models are suitable for predicting the defects.

[Wu et al., 2010] used ARIMA time series and regression models to predict the monthly number of bugs for the Debian distribution over a period of 13 years. They employed the bug tracking system of Debian for this purpose. The results of their prediction show that time series models outperform the regression models when 12 months ahead are predicted. Furthermore they could show in their data, that software bug number series are to a certain extent dependent on seasonal and cyclical factors.

In the domain of networks and web, time series models of ARMA and GARCH are also used in forecasting quality of services (QoS) attributes for web services in [Amin et al., 2012a,b]. The models were used to forecast the future of QoS which had high volatile characteristics. They employed real-world data sets of web services including response time and time between failures. The proposed time series models were able to satisfactorily forecast the QoS and the results were reported to be used in management of the services.

6.7 Summary and Conclusion

In this chapter we studied the evolution of model histories using time series. The evolution was modeled as the sequence of total number of changes between subsequent revisions. The evolution was also measured at two abstraction levels. The first level considered occurrences of low-level edit operations, while the second level considered occurrences of high-level edit operations.

To mathematically model the evolutions, we considered three classes of time series models, namely ARMA, GARCH and mixed ARMA-GARCH models. We provided detailed theoretical fundaments of those models and we deeply discussed the associated methodologies for their estimation, forecasting and simulation.

We observed that evolutions, at both levels, show very erratic behavior. Such erratic patterns were hurdles for time series analysis. Therefore, we proposed a suitable transformation which could make the observed evolution stationary. We found out that most of the projects show heteroscedastic characteristics, suggesting that mixed time series models are more appropriate. Since the heteroscedastic effect was not strong, we checked if simpler ARMA models could model and forecast the evolutions equally good to more complex ARMA-GARCH models. In this regard, we meticulously compared the associated ARMA with mixed ARMA-GARCH models. We found out the mixed model handles the dynamics of evolution more appropriately. Regarding the forecasting capabilities of two models, we found out that their forecasts accuracies are not significantly different.

We also investigated how the proposed time series models can be used to simulate the evolution of design models. We also addressed how simulated sequences of time series models were used to generate more realistic model histories using our model generator.

Project	Model	p-values of Standardized Residuals				p-values of Squared Standardized Residuals			
		5 lags	**10 lags**	**15 lags**	**20 lags**	**5 lags**	**10 lags**	**15 lags**	**20 lags**
ASM	ARMA	9.9434×10^{-1}	9.9974×10^{-1}	9.9925×10^{-1}	9.9999×10^{-1}	7.2483×10^{-1}	5.3556×10^{-1}	8.5174×10^{-1}	9.3800×10^{-1}
	ARMA-GARCH	9.3709×10^{-1}	8.2634×10^{-1}	9.2879×10^{-1}	9.7295×10^{-1}	6.9755×10^{-1}	5.1376×10^{-1}	7.0069×10^{-1}	5.5646×10^{-1}
CheckStyle	ARMA	1.0000×10^{0}	1.0000×10^{0}	9.9993×10^{-1}	9.9986×10^{-1}	5.8322×10^{-1}	5.2531×10^{-1}	3.0460×10^{-1}	1.6750×10^{-1}
	ARMA-GARCH	9.9865×10^{-1}	9.9804×10^{-1}	9.9769×10^{-1}	9.9723×10^{-1}	5.8882×10^{-1}	6.0230×10^{-1}	5.0380×10^{-1}	7.3094×10^{-1}
DataVision	ARMA	1.7009×10^{-1}	1.4027×10^{-1}	1.6918×10^{-1}	4.0803×10^{-1}	1.2335×10^{-1}	3.1978×10^{-1}	6.4850×10^{-1}	8.6452×10^{-1}
	ARMA-GARCH	Not Considered				Not Considered			
FreeMarker	ARMA	9.9916×10^{-1}	9.5335×10^{-1}	9.9113×10^{-1}	9.7406×10^{-1}	7.5992×10^{-1}	7.7695×10^{-1}	9.1324×10^{-1}	9.5954×10^{-1}
	ARMA-GARCH	9.6733×10^{-1}	9.8226×10^{-1}	9.9766×10^{-1}	9.9952×10^{-1}	6.8994×10^{-1}	6.7901×10^{-1}	2.8481×10^{-1}	4.1865×10^{-1}
Jameleon	ARMA	9.9995×10^{-1}	1.0000×10^{0}	1.0000×10^{0}	9.9869×10^{-1}	8.0418×10^{-1}	4.5186×10^{-1}	6.7264×10^{-1}	7.8100×10^{-1}
	ARMA-GARCH	9.6366×10^{-1}	9.6316×10^{-1}	9.9668×10^{-1}	9.9879×10^{-1}	7.7483×10^{-1}	2.4193×10^{-1}	3.9407×10^{-1}	4.7672×10^{-1}
JFreeChart	ARMA	9.6398×10^{-1}	9.9783×10^{-1}	9.9971×10^{-1}	9.9949×10^{-1}	3.1719×10^{-1}	4.6558×10^{-1}	3.4089×10^{-1}	4.5001×10^{-1}
	ARMA-GARCH	9.2308×10^{-1}	9.8403×10^{-1}	9.7476×10^{-1}	9.5203×10^{-1}	6.1368×10^{-1}	6.4415×10^{-1}	7.0465×10^{-1}	7.1189×10^{-1}
Maven	ARMA	8.8442×10^{-3}	3.2245×10^{-2}	1.4539×10^{-1}	3.4948×10^{-1}	5.3626×10^{-1}	8.0391×10^{-1}	7.6968×10^{-1}	8.1692×10^{-1}
	ARMA-GARCH	1.5667×10^{-1}	1.6805×10^{-1}	9.2781×10^{-2}	2.3073×10^{-1}	3.7016×10^{-1}	8.4173×10^{-1}	7.0825×10^{-1}	6.2989×10^{-1}
Struts	ARMA	4.3539×10^{-4}	4.8927×10^{-3}	1.4228×10^{-2}	3.2888×10^{-2}	3.5153×10^{-1}	7.3773×10^{-1}	8.6193×10^{-1}	8.1253×10^{-1}
	ARMA-GARCH	Not Considered				Not Considered			

Table 6.10: Low-Level changes - p-values of the Ljung-Box test of correlation on residuals of the selected models.

Project	Model	p-values of Standardized Residuals				p-values of Squared Standardized Residuals			
		5 lags	**10 lags**	**15 lags**	**20 lags**	**5 lags**	**10 lags**	**15 lags**	**20 lags**
ASM	ARMA	9.9996×10^{-1}	9.9996×10^{-1}	9.9862×10^{-1}	9.9983×10^{-1}	9.1268×10^{-1}	7.1377×10^{-1}	6.9555×10^{-1}	8.1265×10^{-1}
	ARMA-GARCH	8.5304×10^{-1}	7.3143×10^{-1}	9.1976×10^{-1}	9.7747×10^{-1}	7.5689×10^{-1}	7.2393×10^{-1}	9.3471×10^{-1}	7.6970×10^{-1}
CheckStyle	ARMA	9.8770×10^{-1}	9.9844×10^{-1}	9.9390×10^{-1}	9.9833×10^{-1}	1.0569×10^{-1}	7.1706×10^{-4}	1.5841×10^{-3}	6.0110×10^{-3}
	ARMA-GARCH	9.9680×10^{-1}	9.9743×10^{-1}	9.9835×10^{-1}	9.9922×10^{-1}	4.6691×10^{-1}	6.9648×10^{-1}	6.9211×10^{-1}	5.7902×10^{-1}
DataVision	ARMA	3.4028×10^{-1}	2.7955×10^{-1}	2.9532×10^{-1}	5.5756×10^{-1}	5.9083×10^{-1}	7.3132×10^{-1}	8.8416×10^{-1}	9.7598×10^{-1}
	ARMA-GARCH	Not Considered				Not Considered			
FreeMarker	ARMA	9.9476×10^{-1}	9.9956×10^{-1}	9.9972×10^{-1}	9.9997×10^{-1}	1.3559×10^{-1}	2.9807×10^{-1}	3.8119×10^{-1}	4.1704×10^{-1}
	ARMA-GARCH	3.9629×10^{-1}	7.4442×10^{-1}	7.9787×10^{-1}	7.6222×10^{-1}	6.1599×10^{-1}	6.6536×10^{-1}	8.6035×10^{-1}	8.6471×10^{-1}
Jameleon	ARMA	9.9958×10^{-1}	9.9978×10^{-1}	9.9998×10^{-1}	9.9999×10^{-1}	5.8996×10^{-1}	5.0161×10^{-2}	1.3497×10^{-1}	2.8621×10^{-1}
	ARMA-GARCH	9.5435×10^{-1}	7.8191×10^{-1}	3.1622×10^{-1}	4.5863×10^{-1}	6.0655×10^{-1}	3.4806×10^{-1}	3.1688×10^{-1}	1.8832×10^{-1}
JFreeChart	ARMA	9.7192×10^{-1}	9.9973×10^{-1}	9.9997×10^{-1}	9.9999×10^{-1}	4.5106×10^{-1}	4.8238×10^{-1}	4.0736×10^{-1}	4.7795×10^{-1}
	ARMA-GARCH	9.9840×10^{-1}	9.9504×10^{-1}	9.9956×10^{-1}	9.9510×10^{-1}	7.3783×10^{-1}	7.9771×10^{-1}	7.6560×10^{-1}	4.4695×10^{-1}
Maven	ARMA	Not Considered				Not Considered			
	ARMA-GARCH	2.2176×10^{-5}	5.5318×10^{-4}	4.8940×10^{-3}	2.0058×10^{-2}	1.0782×10^{-3}	7.4755×10^{-3}	6.4420×10^{-3}	2.5405×10^{-2}
Struts	ARMA	2.8292×10^{-2}	9.7948×10^{-2}	1.8805×10^{-1}	2.8889×10^{-1}	2.5750×10^{-1}	6.3455×10^{-1}	7.8641×10^{-1}	8.7705×10^{-1}
	ARMA-GARCH	Not Considered				Not Considered			

Table 6.11: High-Level changes - p-values of the Ljung-Box test of correlation on residuals of the selected models.

Part IV

Conclusions and Outlook

CHAPTER 7

Conclusions and Outlook

In this chapter we summarize the results and contributions of this dissertation and outline possible future works.

7.1 Summary and Conclusion

The motivation of this work was the need for test models in order to assess and check algorithms, tools and methods in the domain of model differencing, model versioning, history analysis and associated presentation tools. By studying existing approaches in the domain of MDE, we found out that current approaches are mainly motivated from the domain of model transformation testing and they fail to fulfill the desired properties needed in the domain of model differencing and model versioning, which mostly deal with problems in collaborative and iterative development environments. Particularly, we found out that the existing approaches do not take editing of models into account. The also have no or very little control over the generation process, specified properties of interest cannot be fulfilled in the generated test models and none of them support stochastic or chronological properties in the generated test models.

To address the shortcomings and generate more realistic test models, we proposed a new framework which contributes to generation of stochastically realistic test models for model differencing, model versioning tools, etc. In an overall view, our framework offers a generator which generates models based on the concept of edit operations, addresses the shortcomings in the field and can generate stochastically realistic models. The framework captures the true evolution in design models of real software systems based on the applied low-level and high-level edit operations between subsequent revisions. The pairwise changes, i.e. changes between each two subsequent revisions, are modeled based on many different statistical distributions. The chronological changes (histories) are modeled based on different time series models, namely ARMA, GARCH and mixed ARMA-GARCH models. The framework considers the forecasting and simulation aspects of the mathematical models in order to generate stochastically realistic test models.

In more detail, we proposed an new model generator called SiDiff Model Generator (SMG). SMG generates models based on the concept of edit operations which is more appropriate comparing to the existing appraoches. To generate realistic test models which consider and simulate the properties of real world evolution, one needs to know more about characteristics and properties of evolution in real software systems and regenerate such properties.

Unfortunately, until the contributions of this dissertation, little was known about how models of real software systems evolve over time and which mathematical properties and characteristics they have. The existing approaches for comprehending the evolution of software systems are typically based on software metrics and other static properties of the systems. In such approaches, evolution was typically modeled as differences between measured values of static metrics in revision of software systems. As we showed, such differences can neither fully nor correctly model the evolution, and are not suitable for models of software systems.

In this dissertation we considered the design models of Java software systems. We modeled the evolution of design models using the difference metrics, which were shown to be more useful than the static metrics. The evolution was measured in terms of number of applied edit operations between models, and in this regard two sets of edit operations were used. The first set consist of 75 low-level graph edit operations and the second set consider 188 high-level (developer-friendly) edit operations. Typically a high-level edit operation consists of few to many low-level operations and represents changes at a higher level of abstraction.

In order to study the evolution of design models, we measured the evolution on a set of carefully selected real Java systems as the sample set. For all sample projects, the evolution was measured in terms of both low-level and high-level difference metrics. In the first step, we didn't consider chronological properties of changes, i.e. we just considered the changes without explicitly counting time-dependent properties of evolution. To model the evolution (pairwise changes) we considered sixty statistical models, i.e. distributions, which could be promising. The typical characteristics of data were that, there were big changes between revisions and the histograms of the changes were skewed with heavy tails. We showed that six distributions were quite successful in modeling the evolution and can be used to mathematically formulate the evolution.

To study the chronological properties of evolution, we used time series analysis. In this regard, we used three categorizes of time series models, namely ARMA, GARCH and mixed ARMA-GARCH models. We showed that the evolution could be modeled using ARMA and mixed ARMA-GARCH models. In order to decide which one serves better to properly handle the dynamics of evolutions, we deeply studied and compared these two categories. We found out that although ARMA models are satisfactorily good to handle underlying properties of evolution, mixed ARMA-GARCH models serve better in this regard.

To simulate the evolution and generate more realistic test models, we deeply studied how our proposed statistical models can be simulated and how their valid samples be generated. In this regards, we presented the associated random variate generation (RVG)

algorithms of our successful distributions. Few distributions had complex probability distribution functions, for which there were no tailored RVG algorithms. Therefore, we indirectly generated their random variates. That means, instead of directly handling the problem we generated their random variates by appropriately combining simpler RVG algorithms. Regarding simulation of the proposed time series models, we deeply studied different aspects and strategies for such simulations. To highlight, in the case of test model generation, we are facing the finite-horizon simulations and we showed that initial conditions have strong effects on the simulated sequences. We also addressed how our simulations could be used in our model generator to generate more realistic test models.

We can shortly list our findings, contributions and results as follows:

1. In the domain of model differencing and model versioning, the need for test models could not be fulfilled with the existing approaches which are mostly motivated from the domain of model transformation testing. In the existing approaches, the generation process is not finely under control and the stochastic or other properties of interest cannot be created in the generated models.

2. Our proposed generator, SiDiff Model Generator, addresses the shortcomings in the field. The generation process is finely under control by devising different controlling mechanisms, e.g. selection policies, fitness values, etc. Supporting and using low-level and high-level edit operations, simple and complex structures of interest can be created within the generated models. The generator supports stochastic and other properties of interest within models. Moreover, it can create realistic pairs or sequences of models.

3. Evolution of software systems were mostly studied using static metrics and at code level. Very little was known about the evolution of software design models, their characteristics or mathematical properties. Static metrics are not quite suitable when we consider models of software systems. Therefore, in this dissertation we used difference metrics which are shown to be more appropriate. The evolution was not only measured in terms of low-level, but also high-level metrics. To finely capture the evolution, we used 75 low-level and 188 high-level metrics defined on our design model representation of Java systems.

4. The evolution of design models (pairwise changes) was studied using the low-level and high-level metrics separately. The histogram of changes are typically skewed with heavy tails, therefore suitable statistical models should be able to handle such properties. We investigated sixty promising distributions. Just six of them were able to model the evolution with very good rates of success. The successful models were the discrete Pareto, negative binomial, Yule, Waring, beta-negative binomial and generalized Poisson distributions. For the discrete Pareto and generalized Poisson distributions, we presented existing approaches to generate their random variates. For the rest, we proposed indirect methods to generate their random variates. The random variate generation algorithms were used to generate more realistic test models.

5. The chronological evolution was studied using three types of time series models, i.e. ARMA, GARCH and mixed ARMA-GARCH models. The evolution showed very erratic behaviors and the variance of changes very not stable, thus we proposed a transformation to stabilize the variance. For most of our sample projects, the changes showed significant ARCH effect in the very first few lags, suggesting that the heteroscedastic effect was present, but had no long effect. Due to this fact, we investigated whether simpler ARMA models are satisfactorily good as mixed ARMA-GARCH models. To this aim, we studied whether they handle dynamics of changes or forecast future changes equally good. Although mixed models were superior in capturing the dynamics of changes, their forecasting performance were equally good as simpler ARMA models. To simulate the chronological evolution, we deeply studied how valid sequences of the proposed time series models can be properly simulated.

7.2 Outlook and Future Research Directions

In this work, we captured the evolution of Java software systems based on low-level and high-level metrics which were defined on our design model representation of Java systems. Furthermore, we studied the pairwise and chronological properties of evolution using different statistical distributions and time series models. Now we sketch possible research directions which can be further investigated.

1. Our sample sets of software systems were open source Java systems. A possible research direction is to check if our results are extendable to closed source systems, particularly when business-specific development styles and disciplines are practiced. The same can be investigated on other object-oriented systems instead of Java. The suitability of our Java metamodel and associated sets of edit operations should be further investigated.

2. In this work we focused on design model diagrams, one possible direction for further research is to consider other types of diagrams such as BPMN (Business Process Model Notation), state machine, sequence or activity diagrams. The possible metamodel and the associated sets of edit operations should be accordingly considered and suitability of the proposed statistical models be investigated.

3. In the direction of software evolution analysis, typical approaches consider software metrics and other static properties of systems. In that domain, we did not find research works which consider our proposed statistical distributions to model the changes. One possible research direction is to employ our proposed statistical models and to evaluate their appropriateness against existing approaches. In the case of success, it might help to establish a wider application domain for these statistical models in software systems. They can then capture and model the evolution at code and higher abstraction levels, resulting a unified way of mathematically modeling the evolution.

4. In the domain of software maintenance and cost estimation, the proposed statistical or time series models can be further investigated. The key connection would be to establish a suitable and meaningful relationship between the evolution of design models and the cost of software development or maintenance activities. The proposed statistical models might be then used to infer useful information about future coarse of a system with respect to maintenance and cost estimations.

5. Regarding the pairwise evolution of design models, we used 75 low-level and 188 high-level difference metrics. Since our set of sample Java systems consists of just nine projects, it was not possible to investigate the shape parameters of our proposed distributions. A possible research direction is to investigate whether shape parameters of our distributions have meaningful properties or relationships. In such a case, we will know more details about the evolution, its limiting conditions and its most or least probable outcomes.

6. In this dissertation we modeled the chronological evolution as differences between sum of all metrics for each revision. This let us study the evolution from a coarser granularity. A more delicate approach is to study the co-evolution of metrics all together and try to mathematically formulate it. In this regard, multivariate time series methods might be useful. In the case of such a mathematical model can achieved, the evolution can be finely modeled. Not only this allows us to know how much will be the changes in the next coming revisions, but also the types of such changes as well as the mutual effects. This provide finer flexibility and application e.g. in estimation of cost and maintenance activities.

7. In this work, we considered the occurrences of applied edit operation between revisions of design models and we mathematically modeled the changes. We did not consider where the edit operation were applied and which properties of a model element affect its destiny for being modified. If such properties are investigated, we might be able to mathematically formulate the probability of each model element for being modified. Such information let us find the most probable model elements for modification. In addition to applications in prediction and maintenance, it will let us to more finely control the generation process of test models using our proposed generator, resulting even more realistic test models.

8. The modification of model elements by our generator requires that suitable edit operations are equipped with suitable parameters. In this dissertation, we did not considered the parameters of the applied edit operation between models. A possible future work will be to consider such parameters and try to investigate and formulate them. In conjunction to our other research directions, such information not only has application in maintenance, but also in model differencing and model versioning. Moreover, it will contribute to our test model generation and more realistic test models will be achieved.

References

Lada A. Adamic and Bernardo A. Huberman. Zipf's law and the internet. *Glottometrics*, 3:143–150, 2002. URL `http://www.hpl.hp.com/research/idl/papers/ranking/adamicglottometrics.pdf`.

Joachim H Ahrens and Ulrich Dieter. Computer methods for sampling from gamma, beta, Poisson and bionomial distributions. *Computing*, 12(3):223–246, 1974.

Hirotugu Akaike. A new look at the statistical model identification. *Automatic Control, IEEE Transactions on*, 19(6):716–723, 1974.

Marcus Alanen and Ivan Porres. Model interchange using OMG standards. In *Software Engineering and Advanced Applications, 2005. 31st EUROMICRO Conference on*, pages 450–458. IEEE, 2005.

Marcus Alanen and Ivan Porres. Model interchange using OMG standards. Technical report, TUCS Turku Centre for Computer Science, Abo Akademi University, 2014.

Christos Alexopoulos and Seong-Hee Kim. Review of advanced methods for simulation output analysis. In *Proceedings of the 37th conference on Winter simulation*, pages 188–201. Winter Simulation Conference, 2005.

Christos Alexopoulos and Andrew F Seila. Output data analysis. *Handbook of Simulation*, pages 225–272, 1998.

Kerstin Altmanninger, Martina Seidl, and Manuel Wimmer. A survey on model versioning approaches. *International Journal of Web Information Systems*, 5(3):271–304, 2009.

David Ameller. Considering non-functional requirements in model-driven engineering. Master's thesis, Llenguatges i Sistemes Informátics (LSI), June 2009.

Ayman Amin, Alan Colman, and Lars Grunske. An approach to forecasting QoS attributes of web services based on ARIMA and GARCH models. In *IEEE 19th International Conference on Web Services (ICWS)*, pages 74–81. IEEE, 2012a.

Ayman Amin, Lars Grunske, and Alan Colman. An automated approach to forecasting qos attributes based on linear and non-linear time series modeling. In *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*, pages 130–139. IEEE, 2012b.

G. Antoniol, G. Casazza, M. Di Penta, and E. Merlo. Modeling clones evolution through time series. In *Proc. IEEE Inter. Conf. Software Maintenance*, pages 273–280, 2001.

Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: Advanced concepts and tools for in-place EMF model transformations. In *Model Driven Engineering Languages and Systems*, pages 121–135. Springer, 2010.

J. Scott Armstrong. Evaluating forecasting methods. In *Principles of forecasting*, pages 443–472. Springer, 2001.

J Scott Armstrong and Fred Collopy. Error measures for generalizing about forecasting methods: Empirical comparisons. *International journal of forecasting*, 8(1):69–80, 1992.

Colin Atkinson and Thomas Kuhne. Model-driven development: A metamodeling foundation. *Software, IEEE*, 20(5):36–41, 2003.

Jerry Banks, editor. *Handbook of Simulation: Principles, Methodology, Advances, Application and Practice*. Wiley, 1998.

Jerry Banks. Introduction to simulation. In *Proceedings of the 31st conference on Winter simulation: Simulation - A bridge to the future*, volume 1, pages 7–13. ACM, 1999.

Jerry Banks, John S. Carson, Barry L. Nelson, and David M. Nicol. *Discrete-Event Systems Simulation*. Pearson, 5th edition, 2010.

Clark W Barrett, Roberto Sebastiani, Sanjit A Seshia, and Cesare Tinelli. Satisfiability modulo theories. *Handbook of satisfiability*, 185:825–885, 2009.

Stephen Barrett, Patrice Chalin, and Greg Butler. Model merging falls short of software engineering needs. In *Proc. of the 2nd Workshop on Model-Driven Software Evolution*. Citeseer, 2008.

Benoit Baudry, Trung Dinh-Trong, Jean-Marie Mottu, Devon Simmonds, Robert France, Sudipto Ghosh, Franck Fleurey, Yves Le Traon, et al. Model transformation testing challenges. In *ECMDA workshop on Integration of Model Driven Development and Model Driven Testing.*, 2006.

Gareth Baxter, Marcus Frean, James Noble, Mark Rickerby, Hayden Smith, Matt Visser, Hayden Melton, and Ewan Tempero. Understanding the shape of Java software. In *ACM Sigplan Notices*, volume 41, pages 397–412. ACM, 2006.

B. S. Bennett. *Simulation Fundamentals*. Prentice Hall, 1st edition, 1995.

Daniel Bilar. Probability theory for networks (part 2), 2008a. URL `http://cs.wellesley.edu/~cs249B/lecture/`. CS 249B: Science of Networks, Week 03, Lecture 5.

Daniel Bilar. Scale free distributions: Pareto and Zipf, 2008b. URL `http://cs.wellesley.edu/~cs249B/lecture/`. CS 249B: Science of Networks, Week 03, Lecture 5.

Baki Billah, Maxwell L King, Ralph D Snyder, and Anne B Koehler. Exponential smoothing model selection for forecasting. *International journal of forecasting*, 22(2): 239–247, 2006.

Soren Bisgaard and Murat Kulachi. *Time series analysis and forecasting by example*. Wiely, 2011.

Gerhard Bohm and Günter Zech. *Introduction to statistics and data analysis for physicists*. DESY, 2010.

Tim Bollerslev. Generalized autoregressive conditional heteroskedasticity. *Journal of econometrics*, 31(3):307–327, 1986.

Tim Bollerslev. On the correlation structure for the generalized autoregressive conditional heteroskedastic process. *Journal of Time Series Analysis*, 9(2):121–131, 1988.

Tim Bollerslev, Robert F Engle, and Daniel B Nelson. ARCH models. *Handbook of econometrics*, 4:2959–3038, 1994.

G. E. P. Box and D. R. Cox. An analysis of transformations. *Journal of the Royal Statistical Society. Series B (Methodological)*, 26(2):pp. 211–252, 1964. ISSN 00359246.

George E. P. Box, Gwilym M. Jenkins, and Gregory C. Reinsel. *Time series analysis, forecasting and control*. Wiley, 4th edition, 2008.

Paul Bratley, Bennett L Fox, and Linus E Schrage. *A guide to simulation*. Springer-Verlag New York, 2nd edition, 1983.

Peter J. Brockwell and Richard A. Davis. *Introduction to time series and forecasting*. Springer, 2nd edition, 2002.

Peter J. Brockwell and Richard A. Davis. *Time series: theory and methods*. Springer, 2006.

Petra Brosch. *Conflict Resolution in Model Versioning*. PhD thesis, Faculty of Informatics, Vienna University of Technology, 2012.

Petra Brosch, Gerti Kappel, Martina Seidl, Konrad Wieland, Manuel Wimmer, Horst Kargl, and Philip Langer. Adaptable model versioning in action. *Lecture Notes in Informatics, Gesellschaft für Informatik*, 161, 2010.

Petra Brosch, Gerti Kappel, Philip Langer, Martina Seidl, Konrad Wieland, and Manuel Wimmer. An introduction to model versioning. In Marco Bernardo, Vittorio Cortellessa, and Alfonso Pierantonio, editors, *Formal Methods for Model-Driven Engineering*, volume 7320 of *Lecture Notes in Computer Science*, pages 336–398. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-30981-6.

E. Brottier, F. Fleurey, J. Steel, B. Baudry, and Y. Le Traon. Metamodel-based test generation for model transformations: An algorithm and a tool. In *17th International Symposium on Software Reliability Engineering, 2006. ISSRE '06.*, pages 85 –94, nov. 2006.

Alan W. Brown. Model Driven Architecture: Principles and practice. *Software and Systems Modeling*, 3(4):314–327, 2004.

Alan W Brown, Jim Conallen, and Dave Tropeano. Introduction: Models, Modeling, and Model Driven Architecture (MDA). In *Model-Driven Software Development*, pages 1–16. Springer, 2005.

Richard L. Burden and J. Douglas Faires. *Numerical analysis*. Brooks Cole, 7th edition, 2000.

Jordi Cabot, Robert Clarisó, and Daniel Riera. Verification of UML/OCL class diagrams using constraint programming. In *IEEE International Conference on Software Testing Verification and Validation Workshop, 2008 (ICSTW'08)*., pages 73–80. IEEE, 2008.

John Y. Campbell, Andrew W. Lo, and A. Craig MacKinlay. *The Econometrics of Financial Markets*. Princeton University Press, 1997.

Ngai Hang Chan. *Time Series, Applications to Finance with R and S-Plus*. Wiely, 2nd edition, 2010.

R. C. H. Cheng. The generation of gamma variables with non-integral shape parameter. *Applied Statistics*, pages 71–75, 1977.

R. C. H. Cheng. Generating beta variates with nonintegral shape parameters. *Commun. ACM*, 21(4):317–322, April 1978. ISSN 0001-0782.

John R Cogdell. *Modeling Random Systems*. Prentice Hall, 2004.

Christian Collberg, Ginger Myles, and Michael Stepp. An empirical study of Java bytecode programs. *Software: Practice and Experience*, 37(6):581–641, 2007.

Giulio Concas, Michele Marchesi, Sandro Pinna, and Nicola Serra. On the suitability of Yule process to stochastically model some properties of object-oriented systems. *Physica A: Statistical Mechanics and its Applications*, 370(2):817–831, 2006.

Giulio Concas, Michele Marchesi, Sandro Pinna, and Nicola Serra. Power-laws in a large object-oriented software system. *IEEE Tran. Software Engineering*, 33, 2007. ISSN 0098-5589.

Prem C Consul and Gaurav C Jain. A generalization of the Poisson distribution. *Technometrics*, 15(4):791–799, 1973.

Rama Cont. Empirical properties of asset returns: stylized facts and statistical issues. *Quantitative Finance*, 1(2):223–236, 2001.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009. ISBN 0262033844, 9780262033848.

Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, volume 45, pages 1–17. Citeseer, 2003.

Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. *Communications of the ACM*, 54(9):69–77, 2011.

Luc Devroye. *Non-Uniform Random Variate Generation*. Springer-Verlag, 1986.

Luc Devroye. Random variate generators for the Poisson and related distributions. *Computational Statistics & Data Analysis*, 8(3):247–278, 1989.

Francis X Diebold and Roberto S Mariano. Comparing predictive accuracy. *Journal of Business and Economic Statistics*, 13(3):253–63, July 1995.

Karsten Ehrig, Jochen M Küster, Gabriele Taentzer, and Jessica Winkelmann. Generating instance models from meta models. In *Formal Methods for Open Object-Based Distributed Systems*, pages 156–170. Springer, 2006.

Karsten Ehrig, Jochen Malte Küster, and Gabriele Taentzer. Generating instance models from meta models. *Software & Systems Modeling*, 8(4):479–500, 2009.

Paul Embrechts and Marius Hofert. A note on generalized inverses. *Preprint, ETH Zurich*, 2010.

Walter Enders. *Applied Econometric Time Series*. John Wiley & Sons, Inc., 3rd edition, 2010.

Robert F Engle. Autoregressive conditional heteroscedasticity with estimates of the variance of united kingdom inflation. *Econometrica: Journal of the Econometric Society*, pages 987–1007, 1982.

Robert F Engle and Tim Bollerslev. Modelling the persistence of conditional variances. *Econometric reviews*, 5(1):1–50, 1986.

Arthur Erdelyi, Wilhelm Magnus, Fritz Oberhettinger, and Francesco G. Tricomi. *Higher transcendental functions*, volume 1. McGraw-Hill, 1955.

Norman Fenton and James Bieman. *Software metrics: a rigorous and practical approach*. CRC Press, 2014.

Juan Fernandez-Ramil, Angela Lozano, Michel Wermelinger, and Andrea Capiluppi. Empirical studies of open source evolution. In *Software evolution*, pages 263–288. Springer, 2008.

George S. Fishman. *Discrete-Event Simulation, Modeling, Programming and Analysis*. Springer, 2001.

Franck Fleurey, Jim Steel, and Benoit Baudry. Validation in model-driven engineering: testing model transformations. In *Model, Design and Validation, 2004. Proceedings. 2004 First International Workshop on*, pages 29–40. IEEE, 2004.

Catherine Forbes, Merran Evans, Nicholas Hastings, and Brian Peacock. *Statistical Distributions*. Wiley, 4th edition, 2011.

Martin Fowler. *UML Distilled: A brief guide to the standard object modeling language*. Addison-Wesley Professional, 2nd edition, 2004.

Martin Fowler and Rebecca Parsons. *Domain-specific languages*. Addison-Wesley, 2010.

Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999. ISBN 0201485672.

Christian Francq and Jean-Michel Zakoian. Maximum likelihood estimation of pure GARCH and ARMA-GARCH processes. *Bernoulli*, 10(4):605–637, 2004.

Christian Francq and Jean-Michel Zakoian. *GARCH Models Structure, Statistical Inference and Financial Applications*. Wiely, 2010.

David S Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. John Wiley & Sons, 2003.

Eduardo Fuentetaja and Donald J. Bagert. Software evolution from a time series perspective. In *ICSM*, pages 226–229, 2002.

Xavier Gabaix. Power laws in economics and finance. *Annual Review of Economics*, Vol. 1:255–294, 2009.

AV Gafarian, CJ Ancker Jr, and T Morisaku. The problem of the initial transient in digital computer simulation. In *Proceedings of the 76 Bicentennial conference on Winter simulation*, pages 49–51. Winter Simulation Conference, 1976.

AV Gafarian, CJ Ancker, and T Morisaku. Evaluation of commonly used rules for detecting "steady state" in computer simulation. *Naval Research Logistics Quarterly*, 25(3):511–529, 1978.

Guojun Gan, Chaoqun Ma, and Jianhong Wu. *Data clustering: theory, algorithms, and applications*, volume 20. Siam, 2007.

I. S. Gradshteyn and I. M. Ryzhik. *Table of integrals, series and products*. Academic Press, 7th edition, 2007.

William H. Greene. *Econometric Analysis*. Prentice Hall, 5th edition, 2002.

Massimo Guidolin. Univariate volatility models: ARCH and GARCH, Accessed: July, 2014. URL `http://didattica.unibocconi.eu/myigier/doc.php?idDoc=22285&IdUte=135242&idr=14063&Tipo=m&lingua=eng`.

Terry Halpin. Object-role modeling (ORM/NIAM). In *Handbook on architectures of information systems*, pages 81–103. Springer, 2006.

James Douglas Hamilton. *Time series analysis*. Princeton University Press, 1994.

David Harvey, Stephen Leybourne, and Paul Newbold. Testing the equality of prediction mean squared errors. *International Journal of forecasting*, 13(2):281–291, 1997.

Andreas Henrich, Hans-Werner Six, FemUniversit& Hagen, and Peter Widmayer. The LSD tree: spatial access to multidimensional point and non-saint objects. 1989.

Israel Herraiz. *A statistical examination of the evolution and properties of libre software*. PhD thesis, Universidad Rey Juan Carlos, 2008.

Israel Herraiz. A statistical examination of the evolution and properties of libre software. In *ICSM*, pages 439–442, 2009.

Israel Herraiz, Jesus M Gonzalez-Barahona, and Gregorio Robles. Towards a theoretical model for software growth. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 21. IEEE Computer Society, 2007a.

Israel Herraiz, Jesus M Gonzalez-Barahona, Gregorio Robles, and Daniel M German. On the prediction of the evolution of libre software projects. In *IEEE International Conference on Software Maintenance, 2007 (ICSM 2007)*, pages 405–414. IEEE, 2007b.

Israel Herraiz, Daniel M Germán, and Ahmed E Hassan. On the distribution of source code file sizes. In *ICSOFT (2)*, pages 5–14, 2011.

Israel Herraiz, Daniel Rodriguez, Gregorio Robles, and Jesus M Gonzalez-Barahona. The evolution of the laws of software evolution: a discussion based on a systematic literature review. *ACM Computing Surveys (CSUR)*, 46(2):28, 2013.

Israel Herraiz Tabernero, Daniel Rodriguez, and Rachel Harrison. On the statistical distribution of object-oriented system properties. *2012 3rd International Workshop on Emerging Trends in Software Metrics (WETSoM)*, 2012.

Keith W Hipel and A Ian McLeod. *Time series modelling of water resources and environmental systems*. Elsevier Science Publishing Co, 1994.

Rob J. Hyndman and George Athanasopoulos. *Forecasting: principles and practice*. OTexts, 2013.

Rob J. Hyndman and Anne B. Koehler. Another look at measures of forecast accuracy. *International journal of forecasting*, 22(4):679–688, 2006.

M. Ichii, M. Matsushita, and K. Inoue. An exploration of power-law in use-relation of Java software systems. In *19th Australian Conference on Software Engineering ASWEC*, 2008.

Lovro Ilijašic and Lorenza Saitta. Long-tailed distributions in grid complex network. In *Proceedings of 2nd Workshop Grids Meets Autonomic Computing GMAC*, USA, 2010. ACM. ISBN 978-1-4503-0100-8.

Joseph Oscar Irwin. The generalized Waring distribution. Part I. *Journal of the Royal Statistical Society. Series A (General)*, 138(1):pp. 18–31, 1975. ISSN 00359238.

Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

Daniel Jackson, Aleksandar Milicevic, Joe Near, Eunsuk Kang, and Emina Torlak. Alloy: A language and a tool for relational models. Website, 2014. URL `http://alloy.mit.edu/alloy/`.

Joann Jasiak. *Course in Financial Econometric*. York University, Department of Economics. URL `http://dept.econ.yorku.ca/jasiakj/4140/lecture6.pdf`.

Norman L. Johnson, Samuel Kotz, and Adrienne W. Kemp. *Univariate discrete distributions*. Wiley Interscience, 2nd edition edition, 1992.

Norman L. Johnson, Samuel Kotz, and N. Balakrishnan. *Continuous Univariate Distributions, Volume 1 and Volume 2*. Wiley, 2nd edition, 1994.

Norman L. Johnson, Samuel Kotz, and N. Balakrishnan. *Discrete Multivariate Distributions*. A Wiley-interscience publication. Wiley, 1997.

Norman L. Johnson, Samuel Kotz, and Adrienne W. Kemp. *Univariate discrete distributions*. Wiley Interscience, 3rd editon edition, 2005.

Dieter Jungnickel. *Graphs, networks and algorithms*, volume 5. Springer, 3rd edition, 2006.

Voratas Kachitvichyanukul and Bruce W Schmeiser. Binomial random variate generation. *Communications of the ACM*, 31(2):216–222, 1988.

Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(2):77–131, 2007. ISSN 1532-0618.

Autar Kaw and Egwu Eric Kalu. *Numerical methods with applications*. 1st edition, 2008.

Timo Kehrer. The SiLift Project, semantic lifting of model differences. `http://pi.informatik.uni-siegen.de/Projekte/SiLift/`, 2015.

Timo Kehrer, Udo Kelter, and Gabriele Taentzer. A rule-based approach to the semantic lifting of model differences in the context of model versioning. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 163–172. IEEE Computer Society, 2011.

Timo Kehrer, Udo Kelter, Manuel Ohrndorf, and Tim Sollbach. Understanding model evolution through semantically lifting model differences with SiLift. In *28th IEEE International Conference on Software Maintenance (ICSM 2012)*, pages 638–641. IEEE, 2012a.

Timo Kehrer, Udo Kelter, Pit Pietsch, and Mike Schmidt. Adaptability of model comparison tools. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012)*, ASE 2012, pages 306–309, New York, NY, USA, 2012b. ACM.

Timo Kehrer, Udo Kelter, and Gabriele Taentzer. Integrating the specification and recognition of changes in models. *Softwaretechnik-Trends*, 32(2):41–42, 2012c.

Timo Kehrer, Udo Kelter, and Gabriele Taentzer. Consistency-preserving edit scripts in model versioning. In *IEEE/ACM 28th International Conference on Automated Software Engineering (ASE 2013)*, pages 191–201. IEEE, 2013a.

Timo Kehrer, Udo Kelter, and Gabriele Taentzer. Consistency-preserving edit scripts in model versioning - technical report. Technical report, Department Elektrotechnik und Informatik, Universität Siegen, Germany and Fachbereich Mathematik und Informatik, Philipps-Universität Marburg, Germany, 2013b.

Timo Kehrer, Pit Pietsch, Hamed Shariat Yazdi, and Udo Kelter. Detection of high-level changes in evolving Java software. *Softwaretechnik-Trends*, 33(2), 2013c.

Timo Kehrer, Michaela Rindt, Pit Pietsch, and Udo Kelter. Generating edit operations for profiled UML models. In *ME @ MoDELS*, pages 30–39. Citeseer, 2013d.

Udo Kelter and Maik Schmidt. Comparing state machines. In *Proceedings of the 2008 International Workshop on Comparison and Versioning of Software Models (CVSM)*, pages 1–6. ACM, 2008.

Udo Kelter, Jürgen Wehren, and Jörg Niere. A generic difference algorithm for UML models. In *Software Engineering*, pages 105–116, 2005.

Udo Kelter, Gabriele Taentzer, Timo Kehrer, and Kristopher Born. The MOCA Project, specifying and recognizing model changes based on edit operations. `http://www.dfg-spp1593.de/index.php?id=46`, 2015.

Bénédicte Kenmei, Giuliano Antoniol, and Massimiliano Di Penta. Trend analysis and issue prediction in large-scale open source systems. In *CSMR*, pages 73–82, 2008.

Samir Khuller and Balaji Raghavachari. Graph and network algorithms. *ACM Computing Surveys*, 28(1):43–45, March 1996. ISSN 0360-0300.

Gebhard Kirchgässner and Jürgen Wolters. *Introduction to modern time series analysis*. Springer, 2007.

Barbara Kitchenham. What's up with software metrics? a preliminary mapping study. *Journal of Systems and Software*, 83(1):37–51, 2010.

Anneke G Kleppe, Jos B Warmer, and Wim Bast. *MDA explained, the Model Driven Architecture: Practice and promise*. Addison-Wesley Professional, 2003.

Donald E Knuth. *The Art of Computer Programing – Volume 2 / Seminumerical Algorithms*, volume 2. Addison-Wesley, 2nd edition, 1998.

Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. Model comparison: A foundation for model composition and model transformation testing. In *Proceedings of the 2006 International Workshop on Global Integrated Model Management*, pages 13–20. ACM, 2006.

Dimitrios S Kolovos, Davide Di Ruscio, Alfonso Pierantonio, and Richard F Paige. Different models for model matching: An analysis of approaches to support model differencing. In *ICSE Workshop on Comparison and Versioning of Software Models (CVSM'09)*, pages 1–6. IEEE, 2009.

K. Krishnamoorthy. *Handbook of Statistical Distributions with Applications*. Chapman & Hall / CRC, 2006.

Maher Lamari. Towards an automated test generation for the verification of model transformations. In *Proceedings of the 2007 ACM symposium on Applied computing*, pages 998–1005. ACM, 2007.

Philip Langer, Manuel Wimmer, Petra Brosch, Markus Herrmannsdörfer, Martina Seidl, Konrad Wieland, and Gerti Kappel. A posteriori operation detection in evolving software models. *Journal of Systems and Software*, 86(2):551–566, 2013.

Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice - Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer, 2006. ISBN 978-3-540-24429-5.

Yoann Laurent, Reda Bendraou, and Marie-Pierre Gervais. Generation of process using multi-objective genetic algorithm. In *Proceedings of the 2013 International Conference on Software and System Process*, pages 161–165. ACM, 2013.

Averill M. Law. Statistical analysis of simulation output data. *Operations Research*, 31 (6):983–1029, 1983.

Averill M. Law. *Simulation Modeling and Analysis*. McGraw-Hill, 4th edition, 2007a.

Averill M. Law. Statistical analysis of simulation output data: the practical state of the art. In *Simulation Conference, 2007 Winter*, pages 77–83. IEEE, 2007b.

Averill M. Law. How to build valid and credible simulation models. In *Simulation Conference (WSC), Proceedings of the 2009 Winter*, pages 24–33. IEEE, 2009.

Pierre L'Ecuyer. Uniform random number generation. *Annals of Operations Research*, 53(1):77–120, 1994.

Manny M Lehman. Laws of software evolution revisited. In *Software process technology*, pages 108–124. Springer, 1996.

Meir M Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

Meir M Lehman, Juan F Ramil, Paul D Wernick, Dewayne E Perry, and Wladyslaw M Turski. Metrics and laws of software evolution-the nineties view. In *Proceedings of 4th International Symposium on Software Metrics*, pages 20–32. IEEE, 1997.

B Yu Lemeshko, SB Lemeshko, and SN Postovalov. The power of goodness of fit tests for close alternatives. *Measurement techniques*, 50(2):132–141, 2007.

Yuehua Lin, Jing Zhang, and Jeff Gray. Model comparison: A key challenge for transformation testing and version control in model driven software development. In *OOPSLA Workshop on Best Practices for Model-Driven Software Development*, volume 108, page 6, 2004.

Alexander M Lindner. Stationarity, mixing, distributional properties and moments of GARCH(p,q) processes. In *Handbook of financial time series*, pages 43–69. Springer, 2009.

Shiqing Ling. Self-weighted and local quasi-maximum likelihood estimators for ARMA-GARCH/IGARCH models. *Journal of Econometrics*, 140(2):849–873, 2007.

Greta M Ljung and George EP Box. On a measure of lack of fit in time series models. *Biometrika*, 65(2):297–303, 1978.

Jon Loeliger and Matthew McCullough. *Version Control with Git: Powerful tools and techniques for collaborative software development.* O'Reilly Media, Inc., 2012.

Panagiotis Louridas, Diomidis Spinellis, and Vasileios Vlachos. Power laws in software. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 18(1): 2, 2008.

Edwin Lughofer. *Evolving Fuzzy Systems - Methodologies, Advanced Concepts and Applications.* Springer-Verlag Berlin Heidelberg, 2011.

David MacKenzie, Paul Eggert, and Richard Stallman. *Comparing and Merging Files with GNU diff and patch.* Network Theory Ltd., 2002.

Spyros Makridakis and Michele Hibon. Evaluating accuracy (or error) measures, 1995.

Spyros Makridakis, Steven C. Wheelwright, and Rob J. Hyndman. *Forecasting methods and applications.* Wiley, 3rd edition, 1998.

Michele Marchesi, Sandro Pinna, Nicola Serra, and Stefano Tuveri. Power laws in Smalltalk. In *Proceedings of the ESUG Conference*, pages 27–44. Citeseer, 2004.

Anu Maria. Introduction to modeling and simulation. In *Proceedings of the 29th conference on Winter simulation*, pages 7–13. IEEE Computer Society, 1997.

Roberto S Mariano. Testing forecast accuracy. *A companion to economic forecasting*, pages 284–298, 2002.

MathWorks Company. *Matlab R2013b Documentation Center.* The MathWorks Incorporation, R2013b edition, 2013. URL `http://www.mathworks.com/help/documentation-center.html`.

David S Matteson and David Ruppert. Time-series models of dynamic volatility and correlation. *Signal Processing Magazine, IEEE*, 28(5):72–82, 2011.

Matthew J McGill, RE Kurt Stirewalt, and Laura K Dillon. Automated test input generation for software that consumes ORM models. In *On the Move to Meaningful Internet Systems: OTM 2009 Workshops*, pages 704–713. Springer, 2009.

Allan I McLeod and William K Li. Diagnostic checking ARMA time series models using squared-residual autocorrelations. *Journal of Time Series Analysis*, 4(4):269–273, 1983.

Angus Ian McLeod and Keith William Hipel. Simulation procedures for Box-Jenkins models. *Water Resources Research*, 14(5):969–975, 1978.

Ian McLeod. Derivation of the theoretical autocovariance function of autoregressive-moving average time series. *Applied Statistics*, 24(2):255–256, 1975.

Jacqueline A McQuillan and James F Power. A metamodel for the measurement of object-oriented systems: An analysis using Alloy. In *1st International Conference on Software Testing, Verification, and Validation*, pages 288–297. IEEE, 2008.

Stephen J. Mellor, Anthony N. Clark, and Takao Futagami. Model-driven development. *IEEE software*, pages 14–18, 2003.

Stephen J Mellor, Kendall Scott, Axel Uhl, and Dirk Weise. *MDA Distilled: Principles of Model-Driven Architecture*. Addison-Wesley Professional, 2004.

Tom Mens. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, 2002.

Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.

Tom Mens, Michel Wermelinger, Stéphane Ducasse, Serge Demeyer, Robert Hirschfeld, and Mehdi Jazayeri. Challenges in software evolution. In *8th International Workshop on Principles of Software Evolution*, pages 13–22. IEEE, 2005.

Marjan Mernik, Jan Heering, and Anthony M Sloane. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4):316–344, 2005.

Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, 3 edition, 1996.

Aleksandar Milicevic, Joseph P Near, Eunsuk Kang, and Daniel Jackson. Alloy*: A higher-order relational constraint solver. Technical report, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology (MIT), 2014.

Michael Mitzenmacher. A brief history of generative models for power law and lognormal distributions. *Internet Mathematics*, 1:226–251, 2004.

Douglas C. Montgomery, Cheryl L. Jennings, and Murat Kulachi. *Introduction to Time Series Analysis and Forecasting*. Wiley, 2008.

Alix Mougenot, Alexis Darrasse, Xavier Blanc, and Michále Soria. Uniform random generation of huge metamodel instances. In Richard Paige, Alan Hartman, and Arend Rensink, editors, *Model Driven Architecture - Foundations and Applications*, volume 5562 of *Lecture Notes in Computer Science*, pages 130–145. Springer Berlin / Heidelberg, 2009.

Shiva Nejati, Mehrdad Sabetzadeh, Marsha Chechik, Steve Easterbrook, and Pamela Zave. Matching and merging of statecharts specifications. In *Proceedings of the 29th International Conference on Software Engineering*, pages 54–64. IEEE Computer Society, 2007.

M. E. J. Newman. Power laws, Pareto distributions and Zipf's law. *Contemporary Physics*, 46:323–351, December 2005.

NIST. *NIST/SEMATECH e-Handbook of Statistical Methods*. NIST (National Institute of Standards and Technology) and SEMATECH, 2013. URL `http://www.itl.nist.gov/div898/handbook/`.

Frank W. J. Olver, Daniel W. Lozier, Ronald F. Boisvert, and Charles W. Clark. *NIST handbook of mathematical functions*. NIST (National Institute of Standards and Technology) and Cambridge University Press, 2010.

Object Management Group OMG. Model Driven Architecture (MDA), Document number ormsc/2001-07-01. *Object Management Group*, 2001. URL `http://www.omg.org/mda/`.

Object Management Group OMG. Meta Object Facility (MOF) Specification, Version 1.4, formal/05-05-05. *Object Management Group*, 2005a.

Object Management Group OMG. Unified Modeling Language Specification Version 1.4.2, formal/05-04-01. 2005b.

Object Management Group OMG. Meta Object Facility (MOF) 2.0 Query / View / Transformation Specification, Version 1.1. *OMG White Paper*, 2011.

Object Management Group OMG. Information technology - Object Management Group Object Constraint Language (OCL) formal/2012-05-09. *Object Management Group*, 2012a.

Object Management Group OMG. Information technology - Object Management Group Unified Modeling Language (OMG UML), Superstructure, formal/2012-05-07. 2012b.

Object Management Group OMG. Model-Driven Architecture (MDA) Guide, Revision 2.0, Document ormsc/2014-06-01. *Object Management Group*, 2014. URL `http://www.omg.org/mda/`.

Stefan Otte. Version control systems. *Computer Systems and Telematics Institute of Computer Science Freie Universität Berlin, Germany*, 2009.

Filippo E Pani and Giulio Concas. Stochastic models of software development activities. In *Proceedings of International WSEAS Conference*, number 7 in Recent Advances in Computer Engineering Series. WSEAS, 2012.

Pit Pietsch. The SiDiff framework, technical report. Technical report, Universität Siegen, Praktische Informatik, 2009. URL `http://pi.informatik.uni-siegen.de/Mitarbeiter/pietsch/publications/TechRep.pdf`.

Pit Pietsch and Hamed Shariat Yazdi. The QuDiMo project. `http://pi.informatik.uni-siegen.de/qudimo/`, 2011.

Pit Pietsch, Hamed Shariat Yazdi, and Udo Kelter. Generating realistic test models for model processing tools. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 620 –623, nov. 2011.

Pit Pietsch, Hamed Shariat Yazdi, and Udo Kelter. Controlled generation of models with defined properties. In *Software Engineering*, pages 95–106, 2012a.

Pit Pietsch, Hamed Shariat Yazdi, Udo Kelter, and Timo Kehrer. Assessing the quality of model differencing engines. In *Comparison and Versioning of Software Models (CVSM 2012)*, 2012b.

C Michael Pilato, Ben Collins-Sussman, and Brian W Fitzpatrick. *Version control with subversion*. O'Reilly Media, Inc., 2008.

Hartmut Pohlheim. *GEATbx: Genetic and evolutionary algorithm toolbox for use with Matlab documentation*. `http://www.geatbx.com/docu/docutoc.html`, version 3.80 edition, December 2006.

A. N. Porunov. Box-Cox transforamtion and the illusion of the Normality of macroeconomics series. *Business Informatics*, 2, 2010.

Alex Potanin, James Noble, Marcus Frean, and Robert Biddle. Scale-free geometry in oo programs. *Communications of the ACM*, 48(5):99–103, 2005.

Distributions Core Team R-forge. A guide on probability distributions, 2008-2009.

Svetlozar T. Rachev, Stefan Mittnik, Frank J. Fabozzi, Sergio Focardi, and Teo Jasic. *Financial Econometrics: From Basics to Advanced Modeling Techniques*. John Wiley and Sons, Inc., 2007.

Uzma Raja, David P. Hale, and Joanne E. Hale. Modeling software evolution defects: a time series approach. *Journal of Software Maintenance and Evolution: Research and Practice*, 21(1):49–71, 2009.

Nornadiah Mohd Razali and Yap Bee Wah. Power comparisons of Shapiro-Wilk, Kolmogorov-Smirnov, Lilliefors and Anderson-Darling tests. *Journal of Statistical Modeling and Analytics*, 2(1):21–33, 2011.

Ronald C Read and Derek G Corneil. The graph isomorphism disease. *Journal of Graph Theory*, 1(4):339–363, 1977.

Colin R. Reeves and Jonathan E. Rowe. *Genetic Algorithms Principles and Presentation, A Guide to GA Theory*. Kluwer Academic Publisher, 2002.

Michaela Rindt, Timo Kehrer, and Udo Kelter. Automatic generation of consistency-preserving edit operations for mde tools. In *Proceedings of the Demonstrations Track of the ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems (MODELS 2014) Valencia, Spain.*, 2014.

Brian D. Ripley. *Stochastic Simulation.* Wiley, 1987.

Sheldon M. Ross. *Simulation.* Academic Press, 4th edition, 2006.

Eduardo Rossi. Lecture notes on GARCH models, March 2004. URL `http://economia.unipv.it/pagp/pagine_personali/erossi-old/note32004a.pdf`.

David Ruppert. *Statistics and Data Analysis for Financial Engineering.* Springer, 2011.

R. M. Sakia. The Box-Cox transformation technique: A review. *Journal of the Royal Statistical Society. Series D (The Statistician)*, 41(2):pp. 169–178, 1992. ISSN 00390526.

Joao Paulo Pedro Mendes de Sousa Saraiva. *Development of CMS-based web applications with a multi-language model-driven approach.* PhD thesis, Instituto Superior Tecnico, Universidade Tecnica de Lisboa, 2013.

Richard Saucier. *Computer Generation of Statistical Distributions.* Army Research Laboratory, 2000.

Douglas C Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–31, 2006.

Gideon Schwarz. Estimating the dimension of a model. *The Annals of Statistics*, 6(2): 461–464, 1978.

Edwin Seidewitz. What models mean. *IEEE Software*, 20(5):26–32, 2003.

Andrew F Seila. Advanced output analysis for simulation. In *Proceedings of the 24th Conference on Winter Simulation*, pages 190–197. ACM, 1992.

Bran Selic. The pragmatics of model-driven development. *Software, IEEE*, 20(5):19–25, 2003.

Bran Selic. Personal reflections on automation, programming culture, and model-based software engineering. *Automated Software Engineering*, 15(3-4):379–391, 2008.

Petri Selonen. A review of UML model comparison approaches. In *Nordic Workshop on Model Driven Engineering*, page 37, 2007.

Sagar Sen, Benoit Baudry, and Jean-Marie Mottu. Automatic model generation strategies for model transformation testing. In *Theory and Practice of Model Transformations*, pages 148–164. Springer, 2009.

Shane Sendall and Wojtek Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45, 2003.

Robert E Shannon. Introduction to the art and science of simulation. In *Proceedings of the 30th Conference on Winter Simulation*, pages 7–14. IEEE Computer Society Press, 1998.

Hamed Shariat Yazdi and Pit Pietsch. The SiDiff Model Generator. `http://pi.informatik.uni-siegen.de/qudimo/smg/`, 2011.

Hamed Shariat Yazdi, Pit Pietsch, Timo Kehrer, and Udo Kelter. Statistical analysis of changes for synthesizing realistic test models. In *Multi-conference Software Engineering 2013 (SE2013)*, pages 225–238. Gesellschaft für Informatik (GI), 2013.

Hamed Shariat Yazdi, Mahnaz Mirbolouki, Pit Pietsch, Timo Kehrer, and Udo Kelter. Analysis and prediction of design model evolution using time series. In *Advanced Information Systems Engineering Workshops*, volume 178 of *Lecture Notes in Business Information Processing*, pages 1–15. Springer International Publishing, 2014a. ISBN 978-3-319-07868-7.

Hamed Shariat Yazdi, Pit Pietsch, Timo Kehrer, and Udo Kelter. Synthesizing realistic test models. *Computer Science - Research and Development*, pages 1–23, 2014b.

Raed Shatnawi and Qutaibah Althebyan. An empirical study of the effect of power law distribution on the interpretation of oo metrics. *ISRN Software Engineering*, 2013, 2013.

Robert H. Shumway and David S. Stoffer. *Time Series Analysis and Its Applications: With R Examples*. Springer, 3rd edition, 2011.

S. N. Sivanandam and S. N. Deepa. *Introduction to genetic algorithms*. Springer, 2008.

Harvey Siy, Parvathi Chundi, Daniel J. Rosenkrantz, and Mahadevan Subramaniam. A segmentation-based approach for temporal analysis of software version repositories. *Journal of Software Maintenance and Evolution*, 20(3):199–222, 2008. ISSN 1532-060X.

Thomas Stahl, Markus Voelter, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006. ISBN 0470025700.

Michael Steele and Janet Chaseling. Powers of discrete goodness-of-fit test statistics for a uniform null against a selection of alternative distributions. *Communications in Statistics - Simulation and Computation*, 35(4):1067–1075, 2006.

Mike Steele, Janet Chaseling, and Cameron Hurst. Comparing the simulated power of discrete goodness-of-fit tests for small sample sizes. In *2nd International Conference on Asian Simulation and Modeling, Towards Sustainable LIvelihood and Environment.*, 2007.

Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, Boston, MA, 2nd edition, 2009. ISBN 978-0-321-33188-5.

Matthew Stephan and James R Cordy. A survey of methods and applications of model comparison. Technical report, Queen's University, Ontario, Canada, 2012.

Matthew Stephan and James R Cordy. A survey of model comparison approaches and applications. *International Conference on Model-Driven Engineering and Software Development MODELSWARD (To Appear)*, 2013.

Andreas Svendsen, Øystein Haugen, and Birger Møller-Pedersen. Synthesizing software models: generating train station models automatically. In *SDL 2011: Integrating System and Software Modeling*, pages 38–53. Springer, 2012.

Gabriele Taentzer. Instance generation from type graphs with arbitrary multiplicities. *Electronic Communications of the EASST*, 47, 2012.

Tetsuo Tamai. Process of software evolution. In *2013 International Conference on Cyberworlds*. IEEE Computer Society, 2002.

Tetsuo Tamai and Takako Nakatani. An empirical study of object evolution processes. In *International Workshop on Principles of Software Evolution (IWPSE'98)*, pages 33–37, 1998.

Tetsuo Tamai and Takako Nakatani. Analysis of software evolution processes using statistical distribution models. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 120–123. ACM, 2002.

David B Thomas, Wayne Luk, Philip HW Leong, and John D Villasenor. Gaussian random number generators. *ACM Computing Surveys (CSUR)*, 39(4):11, 2007.

Christoph Treude, Stefan Berlik, Sven Wenzel, and Udo Kelter. Difference computation of large models. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ESEC-FSE '07, pages 295–304, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-811-4.

Ruey S. Tsay. *Analysis of financial time series*. Wiley, 2nd edition, 2005.

Ruey S. Tsay. *An introduction to analysis of financial data with R*. Wiley, 2013.

Ivana Turnu, Giulio Concas, Michele Marchesi, Sandro Pinna, and Roberto Tonelli. A modified Yule process to model the evolution of some object-oriented system properties. *Information Sciences*, 181(4):883–902, 2011.

University of Siegen. Bibliography on comparison and versioning of software models. Online, 2014. URL `http://pi.informatik.uni-siegen.de/CVSM/`.

Rajesh Vasa. *Growth and change dynamics in open source software systems*. PhD thesis, Swinburne University of Technology, 2010.

Rajesh Vasa, Jean-Guy Schneider, and Oscar Nierstrasz. The inevitable stability of software change. In *IEEE Inter. Conf. Software Maintenance, ICSM*, 2007a.

Rajesh Vasa, Jean-Guy Schneider, Oscar Nierstrasz, and Clinton Woodward. On the resilience of classes to change. *ECEASST*, 8, 2007b.

Rajesh Vasa, Markus Lumpe, and Allan Jones. Helix - Software Evolution Data Set, 2010. URL `http://www.ict.swin.edu.au/research/projects/helix`.

Viktor. The Box-Cox transforamtion. Online, March 2010. URL `http://www.mql5.com/en/articles/363`.

Markus Voelter and et al. *DSL Engineering Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013.

Dennis Wackerly, William Mendenhall, and Richard Scheaffer. *Mathematical statistics with applications*. Cengage Learning, 7th edition, 2007.

Christian Walck. Handbook on statistical distributions for exprimentalists. Technical report, Particle Physics Group, Fysikum, University of Stockholm, September 2007.

Junhua Wang, Soon-Kyeong Kim, and David Carrington. Automatic generation of test models for model transformations. In *19th Australian Conference on Software Engineering (ASWEC 2008)*, pages 432–440. IEEE, 2008.

Jos B Warmer and Anneke G Kleppe. *The Object Constraint Language: getting your models ready for MDA*. Addison-Wesley Professional, 2003.

Website. Accompanied material and data of the thesis. `http://pi.informatik.uni-siegen.de/qudimo`, 2015.

William W. S. Wei. *Time series analysis, univariate and multivariate methods*. Pearson, 2nd edition, 2006.

Sven Wenzel. *Unique identification of elements in evolving models, towards fine-grained tracibility in model-driven engineering*. PhD thesis, Universität Siegen, 2010.

Sven Wenzel and Udo Kelter. Analyzing model evolution. In *ACM/IEEE 30th International Conference on Software Engineering (ICSE '08)*, pages 831 –834, may 2008.

Sven Wenzel, Hermann Hutter, and Udo Kelter. Tracing model elements. In *IEEE International Conference on Software Maintenance (ICSM 2007)*, pages 104–113. IEEE, 2007.

Richard Wheeldon and Steve Counsell. Power law distributions in class relationships. In *Proceedings of 3rd IEEE International Workshop on Source Code Analysis and Manipulation*, pages 45–54. IEEE, 2003.

James R Williams and Simon Poulding. Generating models using metaheuristic search. *Sponsoring Institutions*, page 53, 2011.

James R Wilson and A Alan B Pritsker. Evaluation of startup policies in simulation experiments. *Simulation*, 31(3):79–89, 1978a.

James R Wilson and A Alan B Pritsker. A survey of research on the simulation startup problem. *Simulation*, 31(2):55–58, 1978b.

Gejza Wimmer and Gabriel Altmann. *Thesaurus of univariate discrete probability distributions*. Stamm, 1st edition edition, 1999.

Wolfram Research Inc. *Matematica 10 Language and System Documentation Center*, version 10 edition, 2014. URL `http://reference.wolfram.com/language/`.

Hao Wu, Rosemary Monahan, and James F Power. Metamodel instance generation: A systematic literature review. *arXiv preprint arXiv:1211.6322*, 2012.

Wenjin Wu, Wen Zhang, Ye Yang, and Qing Wang. Time series analysis for bug number prediction. In *Proceedings of 2nd International Conference on Software Engineering and Data Mining (SEDM)*, pages 589–596, 2010.

Diethelm Wurtz, Yohan Chalabi, and Ladislav Luksan. Parameter estimation of ARMA models with GARCH/APARCH errors. an R and S-Plus software implementation. *Journal of Statistical Software, forthcoming*, 2006.

He Xiao, Zhang Tian, Ma Zhiyi, and Shao Weizhong. Randomized model generation for performance testing of model transformations. In *Computer Software and Applications Conference (COMPSAC), 2014 IEEE 38th Annual*, pages 11–20. IEEE, 2014.

Zhenchang Xing and Eleni Stroulia. UMLDiff: An algorithm for object-oriented design differencing. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 54–65. ACM, 2005.

Bo Zhou, Dan He, and Zhili Sun. Traffic predictability based on ARIMA/GARCH model. In *2nd Conference on Next Generation Internet Design and Engineering (NGI'06)*, pages 8–pp. IEEE, 2006.

Eric Zivot and Jiahui Wang. *Modelling Financial Time Series with S-Plus*. Springer, second edition, 2006.

Peter Zörnig and Gabriel Altmann. Unified representation of Zipf distributions. *Computational Statistics and Data Analysis*, 19(4):461 – 473, 1995. ISSN 0167-9473.