Controlled Generation of Models with Defined Properties

Pit Pietsch, Hamed Shariat Yazdi, Udo Kelter

Software Engineering Group
University of Siegen
pietsch, shariatyazdi, kelter@informatik.uni-siegen.de

Abstract: Test models are required to evaluate and benchmark algorithms and tools which support model driven development. In many cases, test models are not readily available from real projects and they must be generated. Using existing model generators leads to test models of poor quality because they randomly apply graph operations on graph representations of models. Some approaches do not even guarantee the basic syntactic correctness of the created models. This paper presents the SiDiff Model Generator, which can generate models and model histories and which can modify existing models. The resulting models are syntactically correct, contain complex structures, and have specified statistical properties, e.g. the frequencies of model element types.

1 Introduction

Model-Driven Development (MDD) has become a central development paradigm in many application domains. As a consequence, new tools and methods specifically tailored for MDD are being developed and need to be evaluated with regard to aspects like quality, efficiency and scalability. Examples of such tools are model transformation tools, testing, validation and verification tools, model repositories, and difference and evolution analysis tools.

Finding real test models for evaluation purposes is difficult because in many domains real models are only scarcely available. Even when models from real projects are available their properties are often unknown, or they are not adequate for a particular testing purpose. Different types of tools and MDD methods require significantly different test models. Model comparison and history analysis algorithms need pairs, or sequences, of models, where the evolution is precisely documented. Other algorithms just need very large "realistic" models for scalability tests. As a solution to this challenge we present the *SiDiff Model Generator* (SMG) [PSYK11].

The SMG can create or modify arbitrary model instances complying to EMF Ecore based meta models. It has been developed in the context of the ***SiDiff*** system (s. [KWN05, TBWK07]), a highly configurable framework for model comparison. While originally intended for creating synthetic benchmarks for model comparison algorithms, the SMG evolved into a highly configurable tool for test model generation.

The paper is structured as follows. Section 2 discusses in greater detail the requirements

on a highly versatile model generator. Section 3 presents an overview of the SiDiff Model Generator and explains the main steps in the process of generating synthetic models. The Stochastic Controller, which is responsible for implementing statistical properties of models, is explained in greater detail in Section 4. In Section 5 the runtime of the SMG and the quality of the produced models are discussed. Section 6 discusses other approaches to generate test models and the paper closes in Section 7 with a summary and an outlook.

2 Background and Requirements

Before discussing the requirements on a model generator some definitions that are used throughout the paper are introduced.

We assume that models are typed and that each model type has a set of *edit operations* which can be used to alter a model of this type. For example, state machines should have an edit operation `createState(name)` which creates a new state with a given name. An *Invocation of an Edit Operation* is an edit operation with concrete arguments.

An *asymmetric difference*[1] (or *Patch*) is a sequence of invocations of edit operations. It can be regarded as a linear sequential program, each statement being an edit operation. A difference can be *applied* to, or *executed* on, a model which is called the *base model* in this context. By executing the sequence of invocations on the base model, a new model is created which can be regarded as a successor version.

Invocations of edit operations can refer to existing model elements and/or positions of elements in a given model; this is one reason why an error can occur when an invocation is executed on a model. A difference is *applicable* on a model if no error occurs when the sequence of invocations is executed on this model.

Main Usage Scenarios. A model generators should be able to handle three main use cases: (a) *creating new models* from scratch, (b) *modifying existing models*, i.e. create a difference which is applicable to a given model, and apply this difference, and (c) *creating model histories*. One might think that (c) is just the repeated application of (b) where the last created model is the base model for the next iteration. However, this is not the case: A model history in this context has properties that are connected to multiple revisions, e.g. the life time of model elements.

Correctness of Generated Models. The generated models should be correct according to their complete meta model, i.e. not only conform to the basic abstract syntax, but also to additional constraints which are typically specified as OCL expressions.

Properties of Generated Differences. One should be able to specify properties of the generated models and differences, for example the *size* of the difference(s), i.e. the number of contained operation invocations. Another important property is the *frequency* of edit operations which occur in the generated differences.

[1] There are also symmetric differences, but they are not relevant in this paper.

A model generator should support two interpretations of frequencies: literal and stochastic. With *literal interpretation*, the generated differences will contain exactly the specified number of edit operations. The literal interpretation imposes restrictions on the specified values for the size of the difference and the frequencies: they must be nonnegative integers.

With *stochastic interpretation*, the specified frequencies of edit operations are interpreted as the probabilities of each operation. The generated differences will contain only approximately the specified frequencies. Stochastic interpretation is useful when large models are required for random testing and when the set of edit operations is large and the distribution of frequencies is skewed.

Complex Edit Operations. A model generator should be able to create all correct models, including those containing complicated structures which are not accidentally created by elementary operations. As a consequence, a model generator should offer means to define such complex non-atomic edit operations.

All existing approaches to generate models do not meet several of the above requirements, s. Section 6.

3 The SiDiff Model Generator

Figure 1 gives an overview of the four different main components of the SiDiff Model Generator (SMG), as well as the input and output documents. The meta model and the operation set are dependent on the model type. Further OCL constraints can be specified to disallow unwanted properties in the output model.

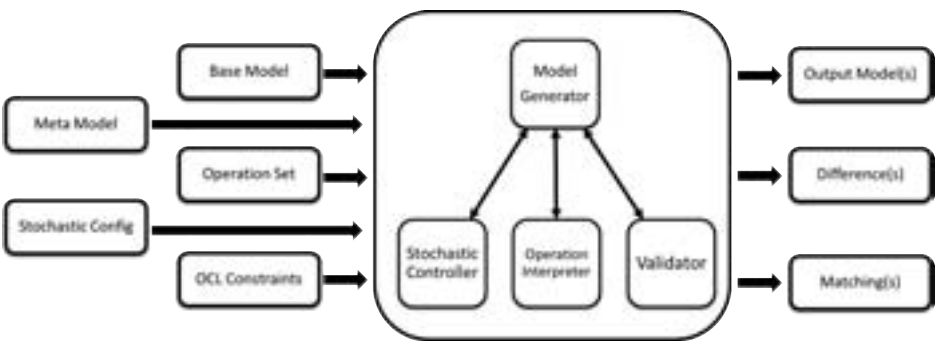

Figure 1: The SiDiff Model Generator - Overview

The *Operation Interpreter* applies selected edit operations to the base model which is provided as input. The *Stochastic Controller* (SC) is configured by an XML-file and ensures that operations are generated in a controlled manner. The *Validator* checks the correctness of the modified model. The *Model Generator* is the central communication interface between the different components.

For each pair of a base and an output model two additional documents are generated: the difference, i.e. the applied operations, and a matching, i.e. a table with corresponding elements of the two models.

The SMG generates only differences which are applicable to the base model without errors. If such a difference cannot be created the whole creation process fails and no output (other than error messages) is created. Any generated differences are, of course, not guaranteed to a applicable to other models which differ from the specified base model.

The SMG covers the all three main usage scenarios mentioned above. Use case (b) is implemented by repeatedly generating an operation invocation which is applicable to the current state of the model being constructed. Use case (a), creating a new model, is implemented by generating a difference which is applicable to an empty model. Use case (c), creating a history, is implemented by creating one large difference which is partitioned into smaller, consecutive parts. In all three cases, the primary product of the SMG is a difference; the new or modified model is a by-product, although this may be the most important result from a user's point of view. The two different interpretation modes for frequencies of operations always refer to this difference; they are further refined as follows.

Literal Interpretation Mode: Here the user specifies exactly the number of invocations for each edit operation. For example, if the edit operation *createClass* is specified to be executed five times, the difference will contain five invocations of *createClass* and the modified model will have five classes more than the base model. The configuration is regarded as a literal specification of both the number and the kind of invoked edit operations, thus allowing the user to define quantitative properties of the generated models in great detail. All other aspects, e.g. where an operation is applied, are still selected with the help of the Stochastic Controller. This mode is, for instance, suited to recreate an observed modification process, e.g. based on data gathered from real model repositories.

Stochastic Interpretation Mode: In the stochastic interpretation mode *Probability Mass Functions* (PMF) of edit operations are used to configure the SMG. In this mode, the frequencies of edit operations are interpreted as probabilities. In case the specified probabilities do not sum up to 100%, the SC internally normalizes them by dividing them to the sum. The size of the difference has to be explicitly specified (whereas in literal interpretation mode, the size is implicitly defined by the sum of the number of edit operation invocations). The behavior of this mode is highly dependent on the size of the difference and on the shape of the PMFs: the generated difference will only approximately exhibit these frequencies. For skewed distributions with heavy tails a higher number of invocations will give better approximations. The stochastic interpretation mode can be used, for example, to create differences of varying size, but similar properties, with very low specification effort.

3.1 Model Modification Process and Index Maps

The model modification process has several consecutive steps: (1) A ContextType, i.e. a meta class which represents a model element type from the given meta model, is selected.

(2) A concrete instance of this meta class and the edit operation are selected. (3) The parameters required for the execution of the operation are selected or created. (4) The created operation is executed.

In the described model modification process situations may arise where it is not possible to complete the creation of an operation: For example, after the selection of ContextType, Context and edit operation, a parameter value which is a reference to a model element might be required, but no suitable model element exists in the model. To avoid expensive backtracking algorithms in these cases, we devised the concept of index maps: Based on the meta model, the operation set and the base model a set of maps is initialized. These maps essentially preserve the knowledge which operations can be successfully created at the current state of the model modification process. In each of the four selection steps mentioned above, the Stochastic Controller filters choices for which it would be impossible to create an applicable operation. The maps are automatically updated after each successful creation of an operation invocation.

The SMG comes with a custom tailored operation model (for more details see [PSYK11]) that allows users to specify simple operations, i.e. creating, deleting, moving and editing attribute values of single model elements, as well as to define complex operations that consist of two or more simple ones.

The SMG is open to the integration of other model transformation engines integrated with EMF Refactor[2]. However, the index maps mentioned above are not supported when EMF Refactor is used since they are tightly coupled to the operation model of the SMG.

4 Stochastic Controller

One very important requirement for a model generator is that the generated differences resemble real, user driven modification processes as much as possible. Common examples of such modification processes may be adding new model elements in close proximity to each other to resemble the implementation of a new subsystem or applying few scattered changes mimicking a debug process. As briefly discussed in Section 3, the *Stochastic Controller* (SC) is responsible for the controlled stochastic selection and application of edit operations to a given base model. In this section we first introduce qualitative properties of models. After this other concepts used in the SC, such as Decision Tables and Selection Policies, are discussed in detail.

4.1 Qualitative Properties of Differences and Fitness Values

In addition to the measurable properties of differences, i.e. size and frequency of edit operations (s. Section 2), qualitative properties affecting the resulting models exist, too. As mentioned in Section 3.1, the invocation of an edit operation consists of four selec-

[2] http://www.eclipse.org/modeling/emft/refactor/

tion steps. The properties of an individual that are used in selection processes are called *effective properties*. These properties mainly depend on the model type and the application domain. For instance, effective properties for the selection of UML classes could be the metric *Number of Methods* or the count of applied modifications. Because effective properties are often not precisely known and complex correlations between them exist, it is necessary to discuss two cases: When enough knowledge of effective properties is available and when they are mostly unknown. For both cases solutions are outlined after the concept of Fitness Values is introduced.

A *Fitness Value* (FV) is a value that describes the aptness of an individual for a definite goal, i.e. selection in our case. For instance, the number of modifications already applied to a model element can be used as a FV to decide whether it is more or less likely that the element in question is selected for further modification. Generally any function that maps all (or some) of the effective properties of individuals to their selection likelihood can be considered as a generator of FVs and we may obtain or construct a cumulative distribution function (CDF) based on them. The case that the range of FVs is finite is a special case in which we may obtain the PMF of individuals by calculating their frequencies. All model elements are annotated with their FVs. Additionally, FVs of individuals can be kept fixed during the model modification process or they can be updated after each successful execution of an edit operation (s. Section 4.2). Since the selection process is altered by this decision, it has a direct effect on the quality of the produced models.

For example, when benchmarking model comparison algorithms, FVs are used to explicitly prevent the application of redundant operations, e.g. renaming an element twice, or operations that cancel each other out, e.g. deleting a newly created element. Since such differences cannot be traced by model comparison algorithms, they would skew the evaluation results.

When the effective properties for selection and their corresponding PMFs are known the SC can simply be adjusted accordingly. In cases where such data is not available, a domain expert still has various options to configure the SC to her best knowledge. One solution is to perform the selection process randomly based on an uniform distribution. Because this is an oversimplification in most cases, more complex methods based on FVs, called *Selection Policies*, were devised. These policies give users the ability to control the selection process quite precisely (s. Section 4.2).

In the case of creating a model history additional effective properties connected to the life cycle of model elements exist, which cannot be considered when simply applying n consecutive differences to a base model. One example of effective properties connected to a history of models are the number of modifications a model element undergoes through its life cycle and the distributions of these modifications over different revisions of the history. Another example are different life time expectations of model elements based on their type. The SC is also capable of handling such complex behaviors.

4.2 Decision Tables and Selection Policies

Suppose that we are going to model the evolution of a given model M to M'. A *Decision Table* (DT) is, roughly speaking, a mapping that tells the system how to modify M in order to eventuate in M'. This concept can be extended to handle model histories: Let m be the number of revisions and $1 \leq i < m$, DT_i is a map between the model in revision i and its successor revision $i + 1$. DT_is are highly configurable and are defined in a configuration file for the system.

Selection Policies (SP) are statistical tools that can be used in the DTs. They select individuals in a controlled stochastic manner based on FVs.

Roughly each DT_i contains the followings:

(C-I) A list of ContextTypes each accompanied with predefined probabilities of selection as well as a SP for selecting one of them.

(C-II) SPs that choose a Context (instance) for each ContextType.

(C-III) A list of defined Operations on each ContextType that are also accompanied with selection probabilities as well as a SP for selecting one operation out of the list. These probabilities can be given from the observed frequencies of edit operations or from PMFs of edit operations. They are only effective in the stochastic interpretation mode.

(C-IV) Each Operation is additionally accompanied with the Number of Frequency which states how often each one will be applied to the model. This configuration data is only relevant when the literal interpretation mode is used.

(C-V) SPs for selecting Parameter values.

Currently five different types of SPs known from Genetic Algorithms [Mic96, Poh06, RR02, SD08] are implemented in the SMG. From now on we suppose that the set of FVs is bounded and their values are nonnegative. This will be a consistent assumption throughout our application scenarios.

As its name implies, the *Random Selection Policy* (RSP) is the obvious case of an uniform random selection method. In *Roulette Wheel Selection Policy* (RWSP) the selection probability of an element is proportional to its FV. This causes better fitted elements to have a higher chance of being selected. If the FVs of selected elements are increased after each operation execution there is a risk that the most fitted elements eventually overwhelm the selection procedure. To avoid this situation, elements can be sorted according to their FVs in a non-decreasing order and the indexes can be used as new FVs. This selection policy is called *Simple Ranking Selection Policy* (SRSP). When there are many individuals, the ones with the lowest FVs will only have a very low chance of being selected. To address this shortcoming either the *Linear Ranking Selection Policy* (LRSP) or the *Non-Linear Ranking Selection Policy* (NLRSP) can be used. Both work based on the extended concept of the SRSP and the rankings are done in a configurable way. Let n be the number of

individuals participating in the selection process, and also let the individuals be already sorted according to their FVs in non-decreasing form, then according to LRSP the new FV for the i th individual is defined as:

$$r_l(i,n) = \frac{1}{n}\big(\alpha + 2\,(1-\alpha)\frac{i-1}{n-1}\big)$$

where $1 \leq i \leq n$ and $\alpha \in [0,2], \alpha \in \mathbb{R}$ is the selection pressure. When α is less than 1 the slope is positive, if $\alpha = 1$ then the line will be horizontal causing all individuals to have equal probability for being selected and when α is greater than 1 we have a negative slope in which most fitted individuals have a lower chance of selection (s. Figure 2).

With previous assumptions, the NLRSP can be obtained by:

$$r_{nl}(i,n) = \frac{n\,x^{i-1}}{\sum_{j=1}^{n} x^{j-1}}$$

where x is the positive root of the following equation which can be solved using Bisection or Newton-Raphson methods [BF00], [KK08]:

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

In the above equation, $\beta \in [1, n-2]$ is the selection pressure and we suppose that $n \geq 3$. By the linear transform $\beta = (n-3)\,\alpha + 1$ where $\alpha \in [0,1], \alpha \in \mathbb{R}$, then we have a suitable handy representation of NLRSP in which α will be the selection pressure. When α tends to zero the slope of the line will tend to 0 as well so we will have a uniform random selection. When α tends to 1 then the selection pressure increases in a non-linear manner (s. Figure 2). Additionally, due to non-linear characteristic of the NLRSP, when the number of participating individuals are big, even moderate selection pressures have strong impacts, i.e. the very few of individuals at the end of the sorted list will get most of

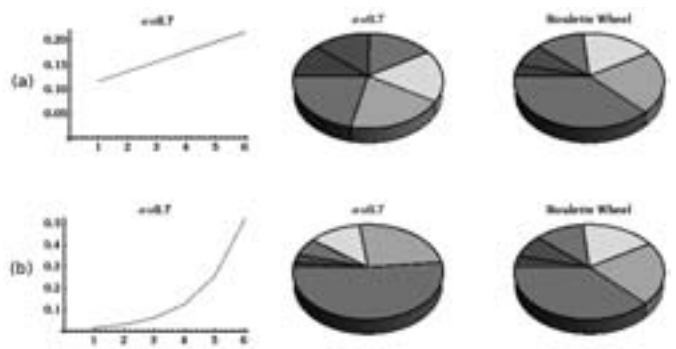

Figure 2: (a) LRSP vs RWSP (b) NLRSP vs RWSP. For a sample of $n = 6$ individuals and selection pressure $\alpha = 0.7$, X-Axis shows the individuals and Y-Axis shows the selection probabilities. Pie charts show the corresponding chance of selection in these two methods.

selection probabilities. Therefore in such situations smaller selection pressures might be more favorable.

Since during the modification process the size and structure of a model is changed dynamically, the SPs can be used to control this process in a subtle way and they have qualitative effects on the results (s. Section 4.1). A small part of a simplified conceptual decision table[3] in the XML format is shown here:

```
<DT>
    <SPCT Name="RWSP" UpdateMode="Dynamic" />
    <CT Name="UMLPackage" p̂="20">
        <SPC Name="NLRSP" α ="0.7" />
        <SPO Name="RWSP" UpdateMode="Fix" />
        <O Name="createClass" p̂="20" f̃="3" />
        <O Name="createInterface" p̂="1" f̃="2" />
    ⋮
</DT>
```

Considering the DTs specifications (s. 4.2), the above XML file is interpreted as: C-I corresponds to the <**SPCT**> tag as well as <**CT**> tags. Each <**CT**> tag contains a <**SPC**> (C-II) and a <**SPO**> (C-III) as well as at least one <**O**> (C-III).

Selection probabilities ($\hat{p}$ values), which are defined in C-I and C-III, can also be used as FVs for SPs since they have the required characteristics. To make things more organized, when SP=RWSP then the values of $\hat{p}$ are used as they are for selecting the individuals (they are automatically normalized), when SP=SRSP, LRSP, NLRSP these values are used as FVs and the selection is done based on the specified SP (in this case normalization is not necessary).

The parameter *UpdateMode* decides whether or not the effective FVs in the selection process are dynamically updated after each successful edit operation execution or stay fixed through the whole the modification process. In the end, the $\tilde{f}$ values are frequencies that are used in the literal interpretation mode (C-IV).

5 Evaluation

To evaluate the SMG the runtime for creating and modifying models was measured. The results of five executions were averaged and show a good performance of the tool. Using a configuration file containing just create edit operations, 4 models (class diagrams) with the sizes of 100, 1000, 5000 and 10000 were first created and then modified. The execution times on an MacBook Pro 2010 with an 2.66 GHz Intel Core i7 processor and 4 GB RAM are shown in Table 1.

In order to evaluate the runtime for modifications of models, a configuration that randomly

[3] In this DT we use the following abbreviation: SP=Selection Policy, CT=ContextType, SPCT=SP for ContextType, SPC=SP for Contexts, SPO=SP for Operations, O=Operation, $\hat{p}$=Selection Probabilities (PMFs), $\tilde{f}$=Frequency of an Edit Operation, α=Selection Pressure for a SP.

No. Elements in the base model (NEBM)		100	1000	5000	10000
Creation times		0.03s	0.52s	10.77s	55.47s
Modification times, with the No. Edit Opr. in % of NEBM	25%	0.02s	0.36s	9.43s	50.31s
	50%	0.03s	0.70s	19.95s	117.21s
	75%	0.02s	1.04s	35.76s	249.59s

Table 1: Runtime for creation and modification of models. The times do not include the times for loading and serialization of the models.

selects and executes edit operations was used. The size of the difference was chosen as 25%, 50%, and 75% of the number of elements in the base model. For example, for the model with 1000 elements and the case of 50%, 500 edit operations were created and executed. Obviously, the runtimes required by the SMG are good and acceptable in practice.

Edit Operation	Spec.freq.	100	1000	5000	10000
create Package	0.64%	3.33%	0.42%	0.84%	0.50%
create Class	4.20%	5.56%	3.85%	4.27%	4.19%
create Interface	0.60%	0.00%	0.73%	0.49%	0.59%
create Attribute	11.50%	12.22%	12.50%	11.64%	11.25%
create Method	28.83%	11.11%	25.46%	27.96%	29.54%
create Parameter	51.83%	62.22%	55.00%	51.97%	51.68%
create Association	2.40%	5.56 %	2.08%	2.83%	2.25%

Table 2: Observed frequencies of elements in the created models vs the specified frequencies.

We also checked how well specified frequencies of operations are actually implemented in the newly generated models for one of the five executions. The frequencies used in our evaluation are shown in column "Spec.freq." in Table 2; they were observed on the software repository of a real project[4].

Not surprisingly, actual frequencies in small differences can diverge substantially from the specified frequencies. Generally such behaviors are intrinsic to probabilities and statistics due to their stochastic nature. Specially in this case, it is caused by the uneven skewed shape of the specified distribution. The actual frequencies of larger difference tend to show better approximations. In our example with the size of 1000 and above our observed frequencies are reasonably close to the specified ones.

6 Related Work

Brottier et al. [BFS+06] present a generator for models, which is used in the context of model transformation testing. The input of the tool consists of an arbitrary meta model and a set of fragments, which are manually or automatically created object structures of inter-

[4] the ASM project, s. http://asm.ow2.org/.

est. [BFS+06] does not discuss how correct fragments are created. The algorithm creates a model by randomly choosing and connecting fragments. None of our requirements is fully met by this approach.

Mougenot et al. [MDBS09] propose a generator specifically aimed at creating large model instances. The algorithm works on a tree representation of models. Therefore the meta model of the model type must be transformed into a tree specification; however, this transformation does not preserve all information. The generation process randomly performs tree edit operations with a uniform distribution. An adaption of the algorithm to realistic statistical distributions is labeled as work in progress. None of our requirements is met by this approach.

Ehrig et al. [EKTW06] use graph grammars to systematically generate instances of arbitrary meta models. The set of rules is organized in three layers: Layer 1 rules create instances of meta model classes, Layer 2 and 3 rules establish relationships between created elements. The rules are automatically deduced from the meta model and applied randomly. This approach meets our correctness requirement, but cannot control properties of the generated models.

The Ecore Mutator developed in the AMOR[5] project is a tool which can modify Ecore-based models. The generator offers a set basic of operations, called mutations, to modify a given base model; additional operations can be implemented if necessary. The operations are performed randomly; the created output model is not checked for validity. None of our requirements are fully met by this approach.

To sum up, the main weakness of existing tools for test model generation is that the resulting models might be syntactically incorrect, might not contain typical constructs, and might have characteristics which differ substantially from realistic models or sets of models.

7 Summary

The SiDiff Model Generator presented in this paper is a versatile tool for creating test models for various purposes and in various usage scenarios. It overcomes several limitations of other current state-of-the-art approaches for test model generation. Most notably, it creates only correct models, it can modify existing models, and it enables users to control the size and many other properties of the created models and differences.

The SMG is generic in the sense that it can create models of arbitrary type. The primary requirement to support a new model type is a meta model and an associated set of edit operation, most of which can be generated from the meta model. Complex operations such as refactorings can be added manually with limited effort.

The SMG is being used in our own projects, a large set of generated models is available from our web site[6].

[5] http://www.modelversioning.org/

[6] http://pi.informatik.uni-siegen.de/qudimo/smg

Acknowledgments

This work was supported by Deutsche Forschungsgemeinschaft under grant KE499/5-1. The authors would like to thank Tim Sollbach and Michaela Rindt as student members of the SMG development team for their help.

References

[BF00] Richard L. Burden and J. Douglas Faires. Numerical Analysis. Brooks Cole, 7th edition, 2000.

[BFS+06] Erwan Brottier, Franck Fleurey, Jim Steel, Benoit Baudry, and Yves Le Traon. Metamodel-based Test Generation for Model Transformations: an Algorithm and a Tool. In Proceedings of ISSRE'06, Raleigh, NC, USA, 2006.

[EKTW06] Karsten Ehrig, Jochen Malte Küster, Gabriele Taentzer, and Jessica Winkelmann. Generating Instance Models from Meta Models. In FMOODS, pages 156–170, 2006.

[KK08] Autar Kaw and Egwu Eric Kalu. Numerical Methods with Applications. 1st edition, 2008.

[KWN05] Udo Kelter, Juergen Wehren, and Joerg Niere. A Generic Difference Algorithm for UML Models. In Software Engineering 2005. Fachtagung des GI-Fachbereichs Softwaretechnik, 2005.

[MDBS09] Alix Mougenot, Alexis Darrasse, Xavier Blanc, and Michèle Soria. Uniform Random Generation of Huge Metamodel Instances. In Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications, ECMDA-FA '09, pages 130–145, Berlin, Heidelberg, 2009. Springer-Verlag.

[Mic96] Zbigniew Michalewicz. Genetic Algorithms + Data Structures = Evolution Programs. Springer-Verlag, 3 edition, 1996.

[Poh06] Hartmut Pohlheim. Geatbx: Genetic and evolutionary algorithm toolbox for use with matlab documentation, version 3.80 edition, December 2006.

[PSYK11] 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 (ASE 2011), Lawrence, KA, USA, Nov 2011.

[RR02] Colin R. Reeves and Jonathan E. Rowe. Genetic Algorithms Principles and Presentation, A Guide to GA Theory. Kluwer Academic Publisher, 2002.

[SD08] S. N. Sivanandam and S. N. Deepa. Introduction to genetic algorithms. Springer, 2008.

[TBWK07] Christoph Treude, Stefan Berlik, Sven Wenzel, and Udo Kelter. Difference Computation of Large Models. In ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, pages 295–304, New York, NY, USA, 2007. ACM.