Uniform random generation of huge metamodel instances

Alix Mougenot*, Alexis Darrasse, Xavier Blanc, and Michèle Soria**

UPMC Paris Universitas, LIP6, France

Abstract. The size and the number of models is drastically increasing, preventing organizations from fully exploiting Model Driven Engineering benefits. Regarding this problem of scalability, some approaches claim to provide mechanisms that are adapted to numerous and huge models. The problem is that those approaches cannot be validated as it is not possible to obtain numerous and huge models and then to stress test them.
In this paper, we face this problem by proposing a uniform generator of huge models. Our approach is based on the Boltzmann method, whose two main advantages are its linear complexity which makes it possible to generate huge models, and its uniformity, which guarantees that the generation has no bias.

1 Introduction

The size and the number of models is drastically increasing, preventing organizations from fully exploiting MDE (Model Driven Engineering) benefits [9]. Today systems are already composed of hundreds of models whose size is quite often close to the thousand of model elements [13]. Regarding the evolution of system complexity [4], one can easily observe that scalability is the most critical of today's (and tomorrow's) challenges.

Approaches that address scalability issues claim to propose adapted mechanisms that deal with numerous and huge models. However, they lack of a complete large-scale validation. Indeed, as it is very difficult to obtain numerous and huge models, it is not possible to really stress test them.

To face this problem, one possible approach is to gather numerous and huge models into open repositories [8]. The idea is to ask large organizations to populate the repositories by providing their larger models. This approach is however not really convincing because, as said by the authors themselves, "one of the main challenges was to find a good quantity of models". Indeed, only 150 models have been stored in the Moogle repository [8], which correspond to 80 thousands model elements, and is therefore not sufficient to realize large-scale stress test.

In this paper, we propose another approach that consists in a uniform generator of huge models. Indeed, we argue that (1) the generator should be uniform[1]

* This work was partly funded by the french DGA.
** work partially supported by ANR contract GAMMA, n°BLAN07-2_195422

[1] By uniform we mean that for a finite class of objects $\mathcal{C}$, any object of $\mathcal{C}$ is produced with equal probability $1/\text{card}(\mathcal{C})$.

in order to be used to validate existing approach without introducing any bias and that (2) the generated models should be huge (millions of model elements) in order to measure the scalability of the approaches.

As we will detail in section 5, most of existing approaches that provide generators of models aim at generating constrained models. Their objective is to find, if possible, models that are consistent regarding a set of constraints. Those approaches are based on constraint solvers and hence have difficulties in generating huge models.

Our approach is based on the Boltzmann method [2] whose two main advantages are its linear complexity which makes it possible to generate huge models, and its uniformity, which guarantees that the generation has no bias.

This article is structured as follows. Section 2 presents the Boltzmann method. Section 3 presents our contribution that is to exploit the Boltzmann method to generate huge models. Section 4 then presents our realization, then section 5 presents works related to the problem of models generation and section 6 presents our conclusion.

2 Boltzmann random generation of trees

Our approach is based on the random sampling of combinatorial structures, within the frame of Boltzmann method, as introduced in [2] (see [11] for an up-to-date review of the method's developments and applications). The main feature of this method is uniform generation with linear complexity, thus allowing for generation of much larger objects than was possible before.

Most random generation methods deal with finite classes of objects, usually objects with a given size, for example binary trees of size one thousand. In the case of Boltzmann method, the notion of uniformity is extended to classes of objects for which the cardinality is infinite, like binary trees of any size. The Boltzmann method only guarantees uniformity for structures of the same size, with the constraint that there is a finite number of elements having the same size. For instance, the number of possible binary trees is infinite, but there is a finite number of binary trees for a given size.

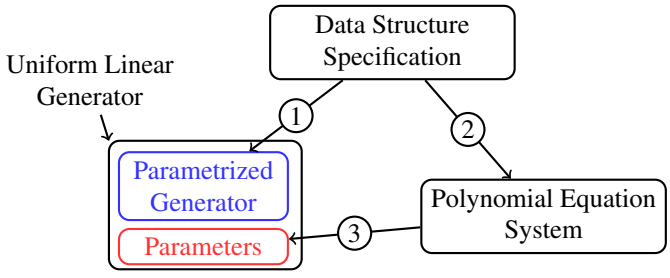


Fig. 1. Boltzmann Method Process Overview

Boltzmann method is generic and can be applied to data whose structure specifications are based on a rich set of constructors, such as disjoint union, Cartesian product, sequences, sets, cycles, etc. It relies on three steps. The first one is the transformation of a `data structure specification` (1) into a `parameterized generator`. The second and third step aim at computing the right parameters for this generator. Step two is the production of a `polynomial equations system` (2) from the `data structure specification` and step three consists in working on the `polynomial equations system` with analytical techniques (these are the domain of analytical combinatorics, described in [5]) in order to compute the actual parameters of the `parameterized generator`, which will make the generator uniform with a linear complexity. By these means, a generator can be automatically compiled from a data structure specification (see figure 1).

This section is continued with a presentation of Boltzmann generation of trees. Section 2.1 presents the notion of *tree specification* which will be used for specifying the structure of the trees to be generated. Section 2.2 describes how to automatically derive the corresponding parametrized generator. Section 2.3 then presents how to compute the actual parameter for making the generator uniform with a linear complexity. Finally, section 2.4 explains why the uniform generator has a linear complexity.

2.1 Tree specifications

In this paper we use the Boltzmann method to generate trees. A *tree specification* in this context will be a context-free grammar with two terminals (Z and ϵ) and three operators (Seq, $|$ and $*$). Z represents one instancible element (either a leaf or any node) whereas ϵ is the empty element. The unary operator (Seq) is used to specify sequences of an arbitrary size $k \geq 0$. The binary operators ($|$) and ($*$) are used to specify respectively union and product.

The size of a tree T, denoted by $|T|$, will be the number of Z it contains. Remember that for a grammar to be admissible by the Boltzmann method, it must only allow for a finite number of trees of a given size, therefore constructions such as Seq(ϵ), which creates an infinity of zero-sized objects, are not allowed. Figure 2 shows three classical examples of tree specifications. First, a binary tree ($\mathcal{T}_1$) which is either a leaf ($\mathcal{L}_1$) or a node ($\mathcal{N}$), with leaves being of one size unit (Z) and nodes being of one size unit that aggregate two binary trees ($Z * \mathcal{T}_1 * \mathcal{T}_1$). Then, one-two tree and a general tree specifications are also given as example.

2.2 General generator automatic construction

In this section we present the transformation that inputs a tree structure specification and returns a corresponding parameterized generator. A parameterized generator is a set of procedures that correspond to each non terminal of the tree structure specification (by convention, we name genT() the procedure that

Tree type	A corresponding grammar
Binary trees	$\mathcal{T}_1 = \mathcal{L}_1 \mid \mathcal{N}$
	$\mathcal{L}_1 = Z$
	$\mathcal{N} = Z * \mathcal{T}_1 * \mathcal{T}_1$
One-two trees	$\mathcal{T}_2 = \mathcal{L}_2 \mid \mathcal{U} \mid \mathcal{B}$
	$\mathcal{L}_2 = Z$
	$\mathcal{U} = Z * \mathcal{T}_2$
	$\mathcal{B} = Z * \mathcal{T}_2 * \mathcal{T}_2$
General trees	$\mathcal{T}_3 = Z * \mathrm{Seq}(\mathcal{T}_3)$

Fig. 2. Classical examples of tree specifications.

corresponds to the generation of the non terminal $\mathcal{T}$). The parameterized generator can be used to generate any tree that conforms to the input structure specification.

When there is a choice point, for union or sequence in the case of tree specifications, the generator uses its parameters to determine either which element of the union should be generated, and or the length of the sequence to generate. Each choice point must respect a particular *choice probability* in order for the global generator to be uniform. These probabilities are driven by a weight operator, noted w, that will ensure that the generation is uniform. The actual parameters of the generator are the weights of each non-terminal in the specification that, if set correctly, will guarantee the uniformity of the generation and the weight of the terminal Z that, if set correctly, will ensure the linear complexity.

The following rules are then used to build the generation procedures, in addition to each rule we give the corresponding weight operator equation for each construction:

- $A = B$: This construct means that the element A is in fact specified by B. Any generation of A is substituted by the generation of B. $w(A)$ is a generator parameter, and $w(A) = w(B)$.

- $B \mid C$: with this construction, either B or C will be generated. The weights of the elements are used here to control the probability to generate either one or the other. The probability of generating B is $w(B)/(w(B) + w(C))$, and the probability of generating C is symmetric. A pseudo-random number can be used to determine which element should be generated with respect to the given probability. The corresponding weight is $w(B \mid C) = w(B) + w(C)$.

- $\mathrm{Seq}(B)$: this construction independently generates a sequence of B. First the number k of components in the sequence is drawn, following a geometric law ($k = \mathrm{geom}(w(B)) = \lfloor \frac{\ln(random([0,1[))}{\ln(w(B))} \rfloor$), and then k elements of type B are independently generated and returned as a sequence. The weight of such a construction is $w(\mathrm{Seq}(B)) = \frac{1}{1-w(B)}$.

– $B * C$: this construction independently generates both an element B and an element C, and the weight is $w(B * C) = w(B) \cdot w(C)$.

– Z: the Z element in the tree specification corresponds to one tree size unit, which very often corresponds to one node. Wherever there is a Z in the specification, a terminal element is generated. The generation of terminals is not handled by the Boltzmann method, it therefore needs to be provided. The weight $w(Z)$ is a special parameter of the generator given by the solver (see details below).

– ϵ: ϵ is the empty element, thus nothing will be generated, and $w(\epsilon) = 1$.

Figure 3 shows one generation algorithm for each specification given in figure 2. In this example we name Ext() the constructor of terminals which must be provided by an external source. The explicit values of the weight will be given in section 2.3.

For better understanding of the implementation of such generator, we detail here the construction of the binary tree's generator, the two other ones are given as illustrations.

The specification of binary trees is $\mathcal{T}_1 = \mathcal{L}_1 \mid \mathcal{N}, \mathcal{L}_1 = Z, \mathcal{N} = Z * \mathcal{T}_1 * \mathcal{T}_1$. This recursive specification states that a binary tree is either a leaf ($\mathcal{L}_1$) or a node ($\mathcal{N}$) aggregating two binary trees. The binary tree random generator (noted genT_1) will have to respect the $\mid$ specification; the probability to generate a leaf must be $w(Z)/(w(\mathcal{L}_1) + w(\mathcal{N}))$ (note that as $w(\mathcal{L}_1) + w(\mathcal{N}) = w(\mathcal{T}_1)$, the probability to generate a leaf is simplified as $w(Z)/w(\mathcal{T}_1)$). To generate elements with the right probability we use a pseudo-random generator for real numbers in $]0, 1]$, if the value produced by the pseudo-random generator is smaller than the leaf probability ($w(Z)/w(\mathcal{T}_1)$) a leaf will be produced, otherwise a node is produced using the external binary tree node constructor that inputs two binary trees generated using recursive calls.

Remark 1. If the tree specification has more than one equation, we obtain one generator for each non-terminal, with possible calls to the other non-terminals generators. We thus need to specify one non-terminal as the "root" of the grammar, in order to provide an entry point for the generation.

The goal of the second step of the Boltzmann method is then to compute the actual parameters in order to make the generator uniform with a linear complexity.

2.3 Boltzmann method

In this section we present how to calculate the weights used by the generator.

Boltzmann method applies to the generation of structured objects, using the powerful tool of *generating functions*. Given a class $\mathcal{C}$ of objects, each object γ having a size denoted by $|\gamma|$, we denote by $C(z)$ its generating function, which is the series $C(z) = \sum_{\gamma \in \mathcal{C}} z^{|\gamma|} = \sum_n c_n z^n$, where c_n is the number of objects of

Binary trees: gen$T_1()$ = **if** random() $< w(Z)/w(\mathcal{T}_1)$
then return gen$L_1()$ **else return** gen$N()$
gen$L_1()$ = **return** Ext()
gen$N()$ = **return** Ext(Gen$T_1()$,Gen$T_1()$)

One-two trees: gen$T_2()$ = r := random**;**
if $r < w(Z)/w(\mathcal{T}_2)$ **then return** gen$L_2()$
elsif $r < w(\mathcal{U})/w(\mathcal{T}_2)$ **then return** gen$U()$
else return gen$B()$
gen$L_2()$ = **return** Ext()
gen$U()$ = **return** Ext(gen$T_2()$)
gen$B()$ = **return** Ext(gen$T_2()$,gen$T_2()$)

General trees: gen$T_3()$ = k := geom($w(\mathcal{T}_3)$)**;** res := []**;**
for i **from** 1 **to** k **do** res := gen$T_3()$::res **done;**
return Ext(res)

Fig. 3. The generation algorithms for the example grammars.

size n in $\mathcal{C}$. In Boltzmann method each object γ is generated with probability $z^{|\gamma|}/C(z)$.

The symbolic method [5] provides a dictionary for translating structural constructions into operators on generating functions: concerning tree constructions, the dictionary reduces to:

$$\begin{aligned} \mathcal{Z} &\to z, & \epsilon &\to 1, \\ \mathcal{C} = \mathcal{A} \mid \mathcal{B} &\to C(z) = A(z) + B(z), \\ \mathcal{C} = \mathcal{A} * \mathcal{B} &\to C(z) = A(z) \cdot B(z), \\ \mathcal{C} = \mathrm{Seq}(\mathcal{A}) &\to C(z) = \tfrac{1}{1-A(z)}. \end{aligned}$$

Thus in a tree specification, each line transforms into a corresponding *generating function* equation (which also corresponds to the weight relations from section 2.2), and a system of specifications transforms into a polynomial system of equations.

For computing weights as described in section 2.2, we need to solve such systems of equations for a given value x of variable z: the weight of element Z is set to x, and the weight of a non-terminal C is the value of series $C(z)$ evaluated at $z = x$. The resolution is analytically coherent for $0 \leq x \leq \rho$, where ρ is a special value, called the *singularity* of the system.

Solving polynomial systems of equations is a very complex problem in general, but systems corresponding to specifications do have a structure that can be exploited in the computations. In our implementation we use a combinatorial newton method that gives a very efficient solver [12], that can also be used to calculate an approximation of the singularity ρ.

In figure 4, we show the generating functions for the previously introduced tree specifications and the calculated weights for each of these systems for $z = \rho$.

In each case, using the values of these functions at $z = \rho$, the Boltzmann algorithms of 2.2 derive a linear time generator with the property of *uniformity*:

Tree type	Corresponding generating functions	Weights
Binary trees	$T_1(z) = L_1(z) + N(z)$ $L_1(z) = z$ $N(z) = z \cdot T_1(z)^2$	$w(Z) = \rho = 1/2$ $w(T_1) = 1$ $w(L_1) = 1/2$ $w(N) = 1/2$
One-two trees	$T_2(z) = L_2(z) + U(z) + B(z)$ $L_2(z) = z$ $U(z) = z \cdot T_2(z)$ $B(z) = z \cdot T_2(z)^2$	$w(Z) = \rho = 1/3$ $w(T_2) = 1$ $w(L_2) = 1/3$ $w(U) = 1/3$ $w(B) = 1/3$
General trees	$T_3(z) = z \cdot \frac{1}{1-T_3(z)}$	$w(Z) = \rho = 1/4$ $w(T_3) = 1/2$

Fig. 4. The generating functions and calculated weights of the example grammars.

given a size n, two trees of that size have exactly the same probability of being generated. These generators however have the particularity that the generated trees are not all of size n, but have a random size, with a mean value depending on parameter z. We show in section 2.4 how to deal with this aspect, using ρ as the value for z.

2.4 Complexity and generation of huge trees

With Boltzmann method, the size of the generated trees is random, with a distribution that depends on the specification and a mean value that goes from 0 to infinity when parameter x goes from 0 to ρ. More precisely the probability for the result to be of size n depends on parameter x and on the singularity ρ, which is attached to the system of equations corresponding to the specification: for large n, this probability is proportional to $n^{-\frac{3}{2}} x^n \rho^{-n}$. Thus the closest is x to the value of ρ, the biggest is the probability of generating large size trees.

As an illustration, in figure 5 we plot the probability of producing a tree of size n in function of n, with different values of x: there are five different curves, corresponding to $x = 0.9\rho, x = 0.999\rho, x = 0.99999\rho, x = 0.9999999999\rho$ and $x = \rho$. In the right part, the curves are plotted with both axes in logarithmic scale, in order to show up the differences. It is quasi-impossible to obtain a tree of size one hundred with a precision of $1/10$ for x/ρ, whereas it is likely to produce a tree of size ten million when ρ is approximated with a precision of $1/10^{10}$.

Boltzmann samplers are particularly efficient if we accept some variability in the size of the generated structures: fixing a target size n and a margin of error δ, generating a structure of size belonging to $[(1-\delta)n, (1+\delta)n]$ can be completed in mean time $O(n)$ (whereas exact size average complexity can be up to quadratic).

In [2], it is showed that in the case of tree sampling, linear time complexity can be achieved by Boltzmann method by using either *pointing* or *singular sampling*. For our implementation, we chose the second approach, consisting in taking ρ as the value of x, and this leads to both issues of computing ρ and rejecting trees

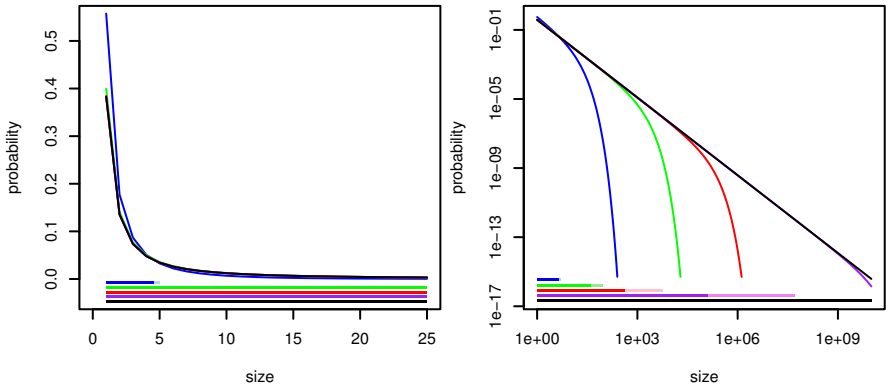


Fig. 5. Probability distribution of sizes for trees generated with Boltzmann method, with a parameter $x = 0.9\rho, 0.999\rho, 0.99999\rho, 0.9999999999\rho$, and ρ. The solid color bars show the range inside which the generators have a guaranteed linear complexity, which in practice extends to the whole colored range. In the second plot, both axes are in logarithmic scale.

of non admissible size. Indeed in the case of singular sampling, the mean size of the generated structures is infinite. We will never generate an infinite object, but there is however a non-trivial probability of generating objects of sizes that we cannot handle. The solution to this problem is simple and consists in aborting the generating process as soon as we pass the upper bound of our target size. As for the second point, the evaluation of ρ is non trivial and will not be detailed in this paper; it uses a dichotomy heuristic and the Newton algorithm of [12].

3 Model generation based on meta model specification

We present in this section a scalable, uniform, random model generation process. This process can be used to generate very big models based on their metamodel specification. It makes use of the uniform random trees generation previously presented. We present here how to transform a metamodel specification into a tree specification and how to complete the generated trees to obtain the final instance.

3.1 Running example

In figure 6 is presented a simple MOF/ECORE [10] metamodel inspired from the classic ECLIPSE EMF[6] Library example. This metamodel will be used throughout this paper to illustrate our approach. It contains four meta-classes: Library, Book, Volume and Compilation where Volume and Compilation inherit the Book abstract meta-class. This metamodel shows three containment relations, from Library to Book, fro Library to Writer and from Compilation to Book, and one relation from book to writer.

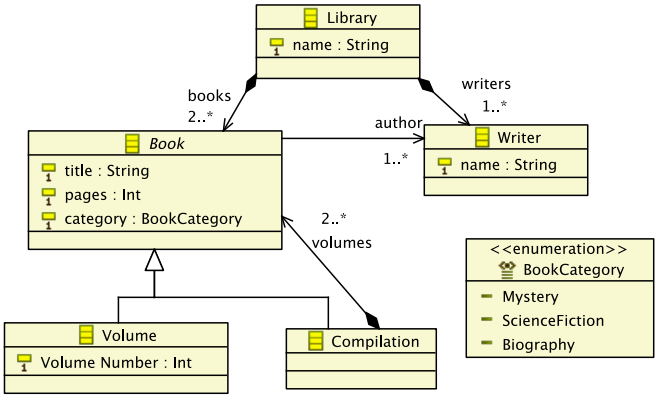


Fig. 6. Running example metamodel diagram

3.2 From metamodels to tree specification

Metamodels and tree specifications are not equivalent. The metamodel language is far more expressive than tree specifications. We present here how to interpret metamodel constructions into tree specification constructions. The transformation we propose is done in three steps where each step refines the output specification tree. Note that the model transformation we define here is not total, some elements in the metamodel will not be translated into the corresponding tree specification. Our approach only generates the core structure of the model.

Identification of base trees The first step to build the random generator is to identify parts of the metamodel that will correspond to the randomly generated tree. A metamodel is a directed graph that is used to specify other graphs. We need to identify trees in the metamodel graph to be able to generate trees that respect the metamodel specifications. The trees used by the random generator are identified thanks to the containment relationships in the metamodel. The containment relationship offers two advantages, they allow to hierarchically generate the model and are acyclic. For each containment relationship found in the metamodel both source and target meta-classes are created in the tree specification and are equal to Z, i.e. an element with one size unit. Abstract meta-classes are created but are not equal to anything at this stage as they should not be instantiated. Finally, each of the containment relationships adds that source equals its value times the target. In the running example we identified three containment relationships. The result of the transformation on the

running example is this tree specification:

$$\begin{aligned} Library &= Z \ * \ Book \ * \ Writer \\ Book &= void \\ Writer &= Z \\ Compilation &= Z \ * \ Book \end{aligned}$$

Inheritance relations The second step to obtain the tree specification is to handle inheritance relations. The inheritance relation is interpreted as a logical or. If meta-class B inherits A, the generation of an instance of A can be replaced by the generation of an instance of B. Therefore, are added to the specification rules for each meta-class A that has daughter meta-class B the fact that A is $A \vee B$. If a new meta-class is encountered, it is equals Z. At the end of this stage all tree specification entries must have a value, all abstract meta-classes that are not inherited must be removed from the tree specification system as they can not be instantiated.

In the running example, *Compilation* and *Volume* inherit *Book*. The resulting tree specification is :

$$\begin{aligned} Library &= Z \ * \ Book \ * \ Writer \\ Book &= Volume \mid Compilation \\ Writer &= Z \\ Volume &= Z \\ Compilation &= Z \ * \ Book \end{aligned}$$

Cardinalities The third step makes sure the cardinalities constraints are respected. To respect lower and upper bound cardinalities the target is multiplied as many times as requested in the tree specification, if the cardinality of a relation A to B is $x..y$ then A is $\vee_{i=x}^{y} B^i$. If the upper-bound is $*$ we use the sequence concept, where $Seq(B)$ denotes an arbitrary long B sequence, note that a sequence may be empty. If the lower bound is 0, the empty element ϵ is used.

If we apply the cardinality constraints to our tree specification, we obtain :

$$\begin{aligned} Library &= Z \ * \ Book^2 \ * \ Seq(Book) * \ Writer \ * \ Seq(Writer) \\ Book &= Volume \mid Compilation \\ Writer &= Z \\ Volume &= Z \\ Compilation &= Z \ * \ Book^2 \ * \ Seq(Book) \end{aligned}$$

Unadapted metamodels At the end of the transformation, all meta-classes in the metamodel should have a value in the tree specification. If it is not the case,

this meta-classes are not accessible, meaning that they can not be randomly generated in a global uniform random generation process, i.e. the metamodel is not suited for our random generation process.

At the end of the transformation, all meta-classes should be linked directly or indirectly with the root of the metamodel. If it is not the case, the given metamodel has more than one root, our random generation process can be used with any of this roots, but only a subpart of the metamodel will be generated.

3.3 Model final structure generation

The tree specification corresponding to the metamodel is used to generate the skeleton of the instance generation as described in section 2, however the Boltzmann tree random generator only generates a model core. It needs a mechanism to generate basic relations that are not containments in order to generate instances of the metamodel. To our knowledge, there is no methodology to uniformly generate such structures and keep the overall generation process random (Boltzmann model does not apply well to graphs). Therefore, any generation process for the basic relations that respects the metamodel specification can be used to complete the skeleton. For instance, in [3] is described as stage two and three such a process. In our implementation on UML Class models we implemented a generator that strictly satisfies the lower bound constraints.

4 Validation

In this section we present our implementation of the generator, as well as particularities of the random sampling that were verified in particle uses when generating UML 2.2 Class Models.

4.1 Implementation

We present in this section the application of the generation process to UML class models. From the official UML 2.2 specification we extracted a simplified class metamodel. The figure 7 presents the tree specification corresponding to this metamodel. Note that the generation processed has been tweaked, the probability to generate packages was reduced and the probability to generate operations, properties, literal integers and parameters augmented. This manipulation is later detailed in this section.

We implemented the generator as an Eclipse Plugin which is available online[2]. The plugin can generate UML files containing the class model from a graphical interface shown in figure 8.

We implemented a value generator for the names, literal values, visibility kinds and direction kind in order to produce a valid model. The properties value generation we implemented is constrained in order to only produce valid

[2] see http://meta.lip6.fr for more details.

Fig. 7. UML 2.2 based tree specification for class models

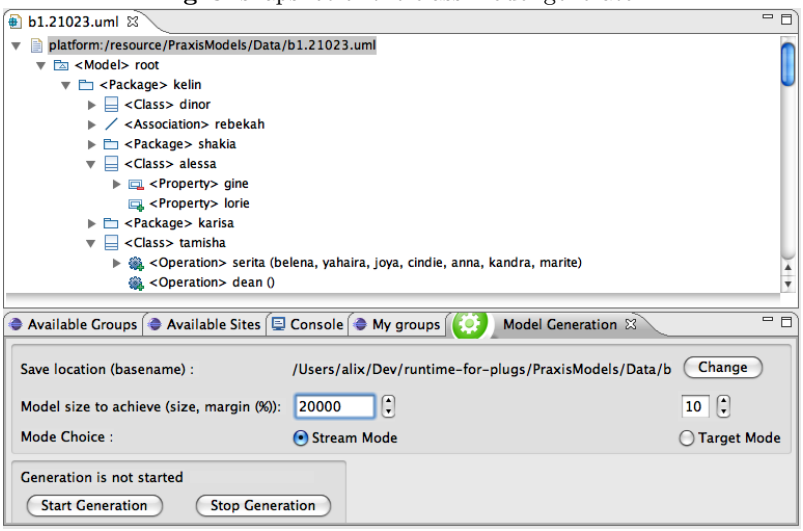
$$
\begin{aligned}
model &= package \\
package &= 0,01Z * Seq(packageableElement) \\
packageableElement &= package \mid class \mid association \\
class &= Z * Seq(property) * Seq(operation) * Seq(generalization) \\
generalization &= Z \\
property &= 3Z * (valueSpecification \mid \epsilon) \\
association &= Z \\
valueSpecification &= literalBoolean \mid literalNull \mid literalInteger \mid literalString \\
literalBoolean &= Z \\
literalNull &= Z \\
literalInteger &= 2Z \\
literalString &= Z \\
operation &= 2Z * Seq(parameter) \\
parameter &= 3Z * (valueSpecification \mid \epsilon)
\end{aligned}
$$

models. We also implemented a generator for generalization and references that randomly chose a valid target in the generated elements. However, the generation of constrained values is not in the scope of this paper.

Even with a linear complexity in random calls, the actual generation process may be long for big models. Indeed, the meta-classes instantiations and properties value generation is time consuming, in order to avoid the generation of unused elements we propose to use a simulation. When a model of a particular size is to be generated, the current random seed is saved, the algorithm to generate a random instance is run without any instantiation, and if the simulation is successful (the size of the model is correct), the seed is reused to generate the actual model, otherwise the simulation is run again. This optimization does not change the theoretical complexity of the sampling but allows to gain a huge amount of time. In our implementation on UML Class models, there is a factor higher than 1000 between the simulation of a valid generation and the same calculation with all meta-classes instantiations.

We present in figure 4.1 the performances of our prototype. This chart shows that the time to obtain a seed that will produce a valid model (valid size) is very short and its average is linear. However our implementation based on EMF [6] is quite slow and can not reasonably produce models above a size of 250 000 model elements due to a huge memory consumption. Therefore we can not provide valid building times for the size 500 000 and one million. It is important to note that the time we provide for obtaining a valid seed is an average for one hundred runs. As the complexity is an mean time complexity, the actual time spent to find a valid seed can significantly vary.

Fig. 8. snapshot of the class model generator



4.2 Generating instances of a particular size

The presented theory states that the mean time to generate a model of a particular size, within a reasonable margin, is linear. The probability to obtain a model of the right size allows us to randomly generate models and only keep the ones with a valid size. As the complexity to generate one model of the wanted size is linear, the complexity to generate a fixed size sampling of uniform models with a size in $[n(1-\delta), n(1+\delta)]$ has a linear complexity too.

It has to be noted, however, that the Boltzmann sampler is very sensible to the value of its parameter, particularly as it approaches its maximal value ρ. Our method depends on taking a parameter equal to ρ, but as ρ can be any real

Model size (10% margin)	Average simulation time	Building time
100	6.50 ms	44.6 ms
1 000	11.2 ms	154 ms
10 000	91.1 ms	1.61 s
50 000	0.501 s	9.87 s
100 000	0.934 s	26.0 s
200 000	1.79 s	52.8 s
250 000	2.48 s	63.2 s
500 000	4.32 s	not applicable
1 000 000	8.86 s	not applicable

Fig. 9. Prototype's performance chart ran on a MacBook Air

number between 0 and 1, it is not possible to calculate it exactly in most cases and we will use an approximation. The effect of this, illustrated in figure 10, is a "ceiling" in the maximal size attainable by the generator. It is therefore important to make a very precise approximation of ρ to be able to generate very large objects. In figure 10 is represented the distribution of class model sizes generated using different approximations of ρ, it appears that the proportion of big models is affected by the accuracy of ρ's calculation. For instance no model of a size greater than five hundred thousand model elements could be generated with a value of ρ correct up to the seventh digit, but with a precision of fifteen digits, ten million model element is possible to reach in linear time.

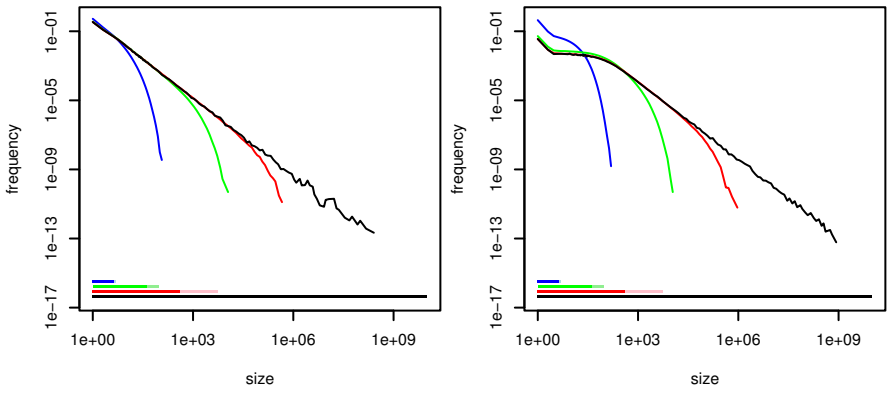


Fig. 10. Distribution of sizes of generated class models by a Boltzmann sampler with a parameter of 0.9ρ, 0.999ρ, 0.99999ρ and ρ (ρ calculated with a 10^{-15} precision). The left plot corresponds to a grammar without coefficients, while in the right the exact grammar of figure 7 is used. Note that both axes are in logarithmic scale.

4.3 Influencing generation output

It is in our opinion very important to be able to characterize the probability distribution of the generated structures, as it allows us to control a possible bias. However, the uniform distribution provided by the Boltzmann samplers may not be the best fit to this needs. We might find, for example, that the number of operations in classes in the random class models is not sufficient. We thus need to add ponderations in our specification, in order to influence the frequency of appearance of the different elements, in a way that allows us to calculate the resulting bias.

The extension of the theory of Boltzmann sampling in this direction is work in progress, but there are some elements that we can already use. We allow the definition, for each non-terminal, of a coefficient with a default value of 1 that

will influence the frequency of the corresponding element. The influence of the coefficient values to the frequencies is not trivial, as the different frequencies depend on each other, so the good choice of coefficients is done for the moment via trial and error. It is possible to calculate the frequencies given the values of the coefficients, and the complexity of the generation process is not affected. In figure 7 where the tree specification of simplified UML class diagrams is given, the probabilities have been modified, the probability to generate packages was reduced and the probability to generate operations, properties, literal integers and parameters augmented in order to obtain more *realistic* class models.

5 Related works

Alloy which is a lightweight specification language based on first-order relational logic [7] can be used to generated models. Indeed, its main principle is to compute all models of a fixed size and that correspond to a particular specification. Then Alloy is able of extracting form this set of models the ones that are consistent regarding to a set of specified constraints. Alloy is based on a SAT solver (the SAT problem belongs to the NP-Hard class) and therefore is not able to produce huge models.

In [3], is presented an algorithm that can generate instances of metamodels. It is based on a transformation of the metamodel structure into a set of graph specification rules. This set of rules is able to generate any skeleton of metamodel instances, and can be coupled with constraint rules in order to respect specific needs. The random process resides in the random election of generation rules. The outputted models can be biased as the choosing of the rule is constrained by the graph specification rule application formalism. Plus, this approach may not scale, the applying of each graph rule has an exponential complexity as it needs to find the existence of a subgraph in the already generated graph which limits the efficiency of this tool (the general problem is NP-Hard).

In [1], a formalism is presented to generate random constrained models. The approach consists in using mutations to derive, from a given instance, random other alike instances. The approach is effective and can handle very huge models since the mutation process is very effective. However, this approach is biased by definition, it needs to input one instance of the model to generate others, therefore the outputted models will have a lot of similarities.

6 Conclusion

In this paper we presented an adaptation of the Boltzmann random sampling theory to metamodel instance generation. The resulting generator has three interesting particularities.
First it is scalable, the complexity of the generating process is linear with the size of the generated structures. And this size is controllable.
Then it outputs uniform samplings for a given size, the probability for any structure of size n to be generated is the same.

And finally, it allows to experimentally change the form of outputted models to meet with specific requirements.

However, metamodels usually come with a set of constraints to precise the specification of its instances, in this paper we did not describe how to generate values for the properties of these instances, however our implementation on class models successfully took this challenge in consideration. In the particular purpose of generating models that satisfy important model constraints, the property generation must be carefully controlled, and possibly a random generation process may not be adapted. Further research in this direction must be done in order to exploit the high performances of the random generation of metamodel instances to constrained models.

References

1. E. Brottier, F. Fleurey, J. Steel, B. Baudry, and Y. L. Traon. Metamodel-based test generation for model transformations: an algorithm and a tool. In Software Reliability Engineering, 2006. ISSRE '06. 17th International Symposium on, pages 85–94, 2006.
2. P. Duchon, P. Flajolet, G. Louchard, and G. Schaeffer. Boltzmann samplers for the random generation of combinatorial structures. Combinatorics, Probability and Computing, 13:577–625, 2004.
3. K. Ehrig, J. Kuster, G. Taentzer, and J. Winkelmann. Generating instance models from meta models. In Formal Methods for Open Object-Based Distributed Systems, pages 156–170, 2006.
4. P. Feiler, R. Gabriel, J. Goodenough, R. Linger, T. Longstaff, R. Kazman, M. Klein, L. Northrop, D. Schmidt, K. Sullivan, et al. Ultra-large-scale systems: The software challenge of the future. Technical report, Software Engineering Institute, Carnegie Mellon University, ISBN 0-9786956-0-7, 2006.
5. P. Flajolet and R. Sedgewick. Analytic Combinatorics. Cambridge University Press, 2009.
6. T. E. Fondation. EMF (Eclipse Modeling Framework). http://www.eclipse.org/modeling/emf/.
7. D. Jackson. Software Abstractions: Logic, Language, and Analysis. The MIT Press, April 2006.
8. D. Lucrédio, R. P. de Mattos Fortes, and J. Whittle. Moogle: A model search engine. In Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3, 2008. Proceedings, pages 296–310, 2008.
9. S. J. Mellor, A. N. Clark, and T. Futagami. Guest editors' introduction: Model-driven development. IEEE Software, 20(5):14–18, 2003.
10. OMG. Meta Object Facility (MOF) 2.0 Core Specification, Jan. 2006.
11. C. Pivoteau. Génération aléatoire de structures combinatoires : méthode de Boltzmann effective. PhD thesis, UPMC, 2008.
12. C. Pivoteau, B. Salvy, and M. Soria. Boltzmann oracle for combinatorial systems. In Fifth Colloquium on Mathematics and Computer Science Algorithms, Trees, Combinatorics and Probabilities, DMTCS Proceedings, pages 475–488, 2008.
13. B. Selic. The pragmatics of model-driven development. IEEE Software, 20(5):19–25, 2003.