Generation of Large Random Models for Benchmarking

Markus Scheidgen[1]

Humboldt Universität zu Berlin, Department of Computer Science,
Unter den Linden 6, 10099 Berlin, Germany
{scheidge}@informatik.hu-berlin.de

Abstract. Since model driven engineering (MDE) is applied to larger and more complex system, the memory and execution time performance of model processing tools and frameworks has become important. Benchmarks are a valuable tool to evaluate performance and hence assess scalability. But, benchmarks rely on reasonably large models that are unbiased, can be shaped to distinct use-case scenarios, and are "real" enough (e.g. non-uniform) to cause real-world behavior (especially when mechanisms that exploit repetitive patterns like caching, compression, JIT-compilation, etc. are involved). Creating large models is expensive and erroneous, and neither existing models nor uniform synthetic models cover all three of the wanted properties.
In this paper, we use randomness to generate unbiased, non-uniform models. Furthermore, we use distributions and parametrization to shape these models to simulate different use-case scenarios. We present a meta-model-based framework that allows us to describe and create randomly generated models based on a meta-model and a description written in a specifically developed generator DSL. We use a random code generator for an object-oriented programming language as case study and compare our result to non-randomly and synthetically created code, as well as to existing Java-code.

1 Introduction

In traditional model driven software engineering, we are not concerned about how much memory our editors consume or how long it takes to transform a model; models are small and execution is instantaneous. But when models become bigger, their processing requires substantial resources. Up to the point, where we began to conceive technology that is specifically designed to deal with large models. In this context, memory consumption and execution times are of central concern and *BigMDE* technology is valued by its performance. Consequently, *benchmarks* that enable sound comparison of a method's, framework's, or tool's performance are valuable and necessary tools in evaluating our work.

In computer science, we define a *software benchmark* as the measurement of a certain performance property taken in a well defined process under a well defined workload in a well defined environment. Where the benchmark mimics certain application scenarios. In MDSE scenarios, workloads comprise input models and tasks that are performed on these models.

Input models characteristics have an influence on the quality of the benchmark. First, input models need to be unbiased, i.e. they must not deliberately or accidentally ease the processing by one technology and burden another. Secondly, they need to be *real* enough to invoke behavior that is similar to behavior caused by actual input models. Non random synthetic models for example are often overly uniform and can therefore fool compression, caching, or runtime optimization components (often contained in BigMDE technology) into unrealistic behavior. Thirdly, benchmarks mimic different application scenarios. Different scenarios require input models with different *shapes*. Here, the concrete form of possible shapes depends on the meta-model. E.g. shapes can be expressed as sets of model metrics. Fourthly and finally, input models need to scale, i.e. we need to find input models of arbitrary size. Only with scalable input, we can use benchmarks to access the scalability of MDSE technology.

It is common practice to either use existing models (e.g. the infamous Grabats 09 models) or non random synthetically generated models. Where the former yields in realistic, somewhat unbiased, but only given shapes and scale, the later results in arbitrary large, but utterly unrealistic models. Our approach is to use randomness as a tool to achieve both scalability and configurability as well as a certain degree of realism and reduced bias. We present a generator description language and corresponding tools that allows us to describe and perform the generation of random models with parameterizable generator rules.

The paper is structured as follows. We start with a discussion of related work in the next section. After that, we introduce our generator language. The next section, demonstrates our approach with a generator for object-oriented program code models. The paper closes with conclusions and suggestions for further work.

2 Related Work

Benchmarking with Existing Models

The idea to generate large models for benchmarking large model processing technologies is new and to our knowledge there is no work specifically targeting this domain. However, benchmarking of large model processing technologies has been done before; mostly by authors of such technologies. Common practice is the use of a specific set of large existing models. If we look at benchmarking model persistence technology for NoSQL databases, which is our original motivation for this work, the set of the Grabats 09 graph transformation contest example models are exclusivly used by all work known to us [7,1,8,2].

This benchmarking practice has itself established as a quasi standard to evaluate and compare model persistence technology, even though it exhibits all of the previously stated flaws of using existing models for benchmarks. First, the Grabats models aren't exactly large ($<10^7$ objects), at least when compared to the target scale of the benchmarked technology. Secondly, there is no meaningful mathematical relationship between size metrics of the models in the set, e.g. there is no usable ramp-up in model-size. Even though the models show an increasing size, the models have internally different structure, which makes them non-comparable. Some models represent Java

code in full detail, others only cover declarations. This makes it impossible establish a meaningful formal relationship between size-metric and performance measurements. Thirdly, we have a very small set of 4 models and all models are models of the same meta-model. This makes the example set biased (towards reverse-engineered Java code models) and again makes it difficult to establish relationships between metrics and performance. Forthly, the internal structure of the models makes it impossible to import all the models into CDO. At least no publication presented any measurements for CDO and the biggest two of the four Grabats models. This is especially bad, since CDO as most popular SQL-based persistence technology, presents the most reasonable (and only used) baseline.

More concrete benchmarks including the precise definition of tasks exist for model queries and transformations [11]. The Grabats 09 contents actually formulates such benchmarks. In [11,5] the authors define frameworks to precisely define tasks for model transformation and query benchmarks and provide the means to evaluate the created benchmarks in that domain.

Model Generation for Test

Before benchmarking became an issue for MDSE, models were generated to provide test subjects for MDSE technology. We can identify three distinct approaches.

SAT-solver For most test scenarios, not only syntactically but also static semantically correct models are needed. Brottier et al. [3] and Sen et al. [9] propose the use of SAT-solvers to derive meta-model instances from the meta-model and it's static semantic constraint. Meta-model and constraints are translated into logical formula, solutions that satisfy these are translated back to corresponding meta-model instances. Since each solution "solves" the meta-model and it's constraints, the solution represents a semantically correct instance of the given meta-model. The non-polynomial complexity of SAT problems and the consequently involved heuristics do not scale well.

Existing graph generators Mougenot et al. [6] use existing graph generators and map nodes and edges of generated graphs to meta-model classes and associations. While this approach scales well, it does not allow to shape the generated model. The used algorithms provide uniform looking random graphs and result in uniform models. In reality, most models cover different aspects of a system with completely different structural properties. E.g. the graph that represents package,class,method declarations has different properties/metrics as a graph that describes the internals of a method implementation.

Constructive formalisms Models can be generated with constructive formalisms like formal grammars or graph grammars. Ehrig et al. [4] propose to use graph grammars and random application of appropriate graph grammar rules to generate models. Our own approach (which is very similar to context-free grammars) fits this category as well. In comparison it lacks the formal properties of the graph grammar approach and is limited to context-free constructs (e.g. no static semantic constraints), but scales better due to the simpler formalism. In practice graph-grammars introduce

further restrictions, since graph grammars become inherently complex, especially if one tries to map static semantic constraints into the rules.

3 Generator Language

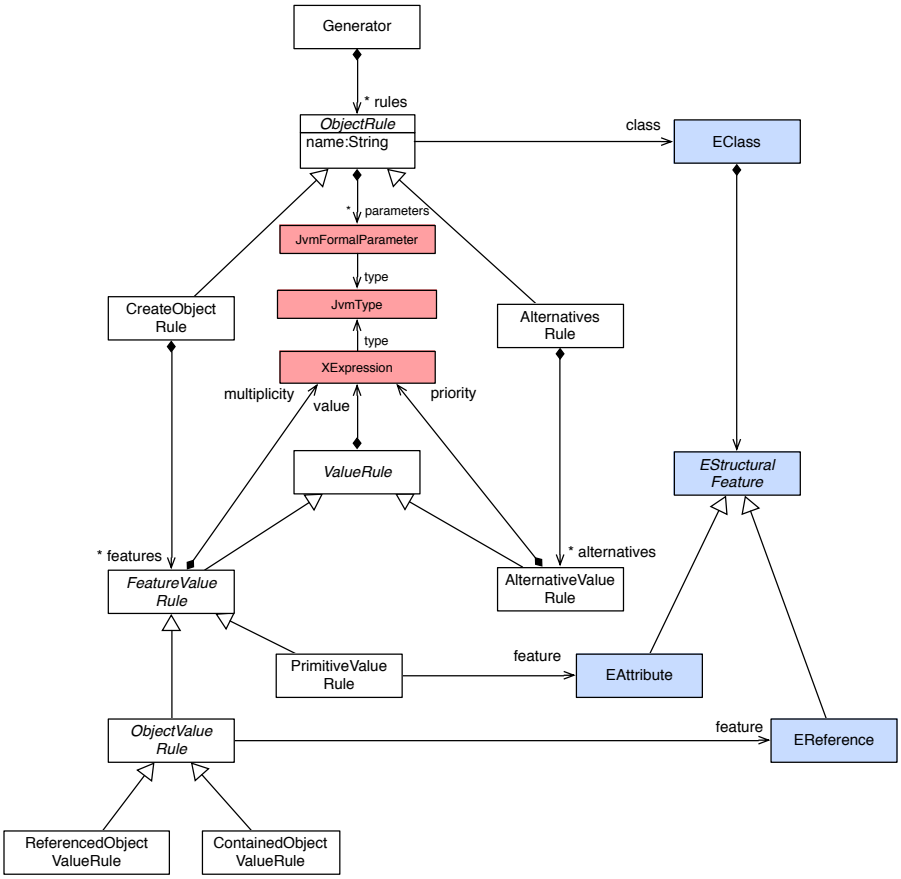

Fig. 1: Meta-model of the proposed model generators description language.

We developed the model generator description language *rcore*[1] as an external domain specific language based on EMF, xText, and xBase with the consequent eclipse-based tool support. Fig. 1 depicts the underlying meta-model of our generator language. *RCore* follows a declarative approach and uses production rules to describe

[1] The source code and examples can be obtained here:
http://github.com/markus1978/RandomEMF

possible models. Similar to a formal grammar that can be used to generate strings over an alphabet, we generate models over a meta-model. In contrast to grammars, *rcore* rules govern rule application via expressions that present concrete choices (i.e. concrete multiplicities or chosen alternatives). These expressions can either use fix-values (e.g. to generate synthetic models) or call random number generators following different distribution functions (i.e. to generate random models). Build in variables (e.g. the generated model in progress, or depth of rule application) and custom rule parameters can also be used within expressions.

Generally, we distinguish between `ObjectRule`s that can generate model elements (i.e. EMF objects) and `FeatureValueRule`s that assign values to an object's structural features (i.e. EMF attributes and references). Cascading application of `ObjectRule-FeatureValueRule-ObjectRule-Fea...` allows clients to generate containment hierarchies, i.e. the spine of each EMF model.

Each *rcore* description comprises an instance of `Generator` and is associated with an ecore-package that represents the meta-model that this generator is written for. Each `Generator` consist of a set of `ObjectRule`s, where the first `ObjectRule` is the start rule. Each of these `ObjectRule`s is associated with a `EClass` that determines the meta-type of the objects generated by this rule. There are two concrete types of `ObjectRule`s: `CreateObjectRule`s that describe the direct creation of objects (i.e. meta-class instances, a.k.a model-elements) and `AternativesRule`s that can be used to randomly refer object creation to several alternative rules. Further, `ObjectRule`s can have parameters.

While `ObjectRule`s are used to describe object generation, `ValueRule`s are used to determine values that can be used within `ObjectRule`s. `ValueRule`s determine concrete values via an xBase `XExpression` that can evaluate to a primitive value (e.g. to be assigned to an attribute, `PrimitiveValueRule`), or that calls an `ObjectRule` (e.g. to create a value for a containment reference, `ContainedObjectValueRule`), or that queries the generated model for a reference target (e.g. to be assigned to a non-containment reference, `ReferencedObjectValue`), or that refers object creation to an `ObjectRule` (e.g. to create an alternative for an `AlternativesRule`, `AlternativeValueRule`).

Concrete `FeatureValueRule`s are associated with a `EStructuralFeature` and are used to assign values to the according feature of the object created by the containing `CreateObjectRule`. Each `FeatureValueRule` also has an expression that determines the multiplicity of the feature, i.e. that determines how many values are assigned to the feature, i.e. how many times the value expression is evaluated. `ObjectValueRule`s are associated with `EReference` and are used to assign values to references, and `PrimitiveValueRules` are associated with `EAttribute` and are used to assign value to attributes.

`AlternativeValueRule`s have an additional expression that determines the priority of the alternative within the containing `AlternativesRule`. When applied the `AlternativesRule` will uniformly choose an alternative with priorities as weights. Only the chosen alternative is evaluated to provide an object.

Further static semantic constraints have to be fulfilled for a correct *rcore* description. (1) the value expressions of `AlternativeValueRules` must have compatible

type with the `EClass` associated with the containing `AlternativesRule`. The types of all value expressions in `FeatureValueRule`s must be compatible with the associated structural feature's type. The associated features of `FeatureValueRule`'s must be features of the `EClass` associated with the containing `CreateObjectRule`. All used `EClass`es must be contained in the meta-model that is associated with the `Generator`.

```
1.  package de.hub.rcore.example

3.  import org.eclipse.emf.ecore.EDataType
4.  import org.eclipse.emf.ecore.EcorePackage
5.  import static de.hub.randomemf.runtime.Random.*

7.  generator RandomEcore for ecore in "platform:/resource/org.eclipse.emf.ecore/model/Ecore.ecore" {
8.    Package: EPackage ->
9.      name := LatinCamel(Normal(3,2)).toLowerCase
10.     nsPrefix := RandomID(Normal(2.5,1))
11.     nsURI := "http://hub.de/rcore/examples/" + self.name
12.     eClassifiers += Class#NegBinomial(5,0.5);
13.
14.   Class: EClass ->
15.     name := LatinCamel(Normal(4,2))
16.     abstract := UniformBool(0.2)
17.     eStructuralFeatures += Feature#NegBinomial(2,0.5);
18.
19.   alter Feature: EStructuralFeature ->
20.     Reference(true) | Reference(false) | Attribute#2;
21.
22.   Reference(boolean composite):EReference ->
23.     name := LatinCamel(Normal(3,1)).toFirstLower
24.     upperBound := if (UniformBool(0.5)) -1 else 1
25.     ordered := UniformBool(0.2)
26.     containment := composite
27.     eType:EClass := Uniform(model.EClassifiers.filter[it instanceof org.eclipse.emf.ecore.EClass]);
28.
29.   Attribute:EAttribute ->
30.     name := LatinCamel(Normal(3,1)).toFirstLower
31.     upperBound := if (UniformBool(0.1)) -1 else 1
32.     eType:EDataType := Uniform(EcorePackage.eINSTANCE.EClassifiers.filter[it instanceof DataType]);
33. }
```

Fig. 2: Example generator description for Ecore models.

Fig. 2 presents an example *rcore* description. The described generator produces randomly generated Ecore models. Some of the expressions use predefined random number generator based method to create random ids, strings, numbers (based on different distributions such as normal, neg. binomial, etc.)). The method `Uniform(List<EObject>)` for example, is used to uniformly draw reference targets from a collection of given possible target objects. Another example is `LatinCamel(int)` that generates a camel case identifier with the given number of syllables. The two methods `Normal(double,double)` and `NegBinomial(double,double)` are exam-

ples for the use of random number generators to create normal and negative binomial distributed random numbers.

4 Example Results

```
1.  package dabobobues;
3.  class Dues {
4.
5.    DuBoBuTus begubicus;
6.    ELius brauguslus;
7.
8.    void Dues(Alius donus, FanulAudaCio aubetin) {
9.    }
10.
11.   void baGusFritus() {
12.     eudaguslius = "";
13.     bigusdaGubolius();
14.     if ("") {
15.       annulAugusaugusfrigustin("");
16.       albucio = Dues()<=++12;
17.       bi();
18.       eBoTor();
19.     } else {
20.       brauguslus = 9;
21.       baGusFritus();
22.       duLus = ""=="";
23.     }
24.   }
25.
26.   void aufribonulAubufrinus(Dues e) {
27.     dobubogutor();
28.     aubiguTus = 9;
29.   }
30. }
```

Fig. 3: Example of randomly generated code for an object-oriented programming language.

In this section, we want to demonstrate the use *rcore* for meta-models that are more complex than the *Ecore* example in the previous section. Based on their popularity, overall complexity, and well understood properties, we chose an object-oriented Java-like programming language with packages, classes, fields, methods, statements, expressions (incl. operators and literals). We omitted details that are either repetitive (template parameters, interfaces, anonymous classes) or not interesting from a randomized generation point of view (most flags and enums like abstract, overrides, visibility, etc.). We developed the model generator based on a ecore meta-model for the described example language, but we use a simple code generator to pretty print the generated models in a Java-like syntax for better human comprehension. Fig. 3 depicts a sample of the generated models in the serialized form.

Similar to our *Ecore* generator in Fig. 2, we use probability distributions to produce randomized models that exhibit certain characteristics such as average number

of methods per class and similar metrics. We compare the generated results to non synthetic instances of the same meta-model and to program code models of a real-life Java project. We look at two aspects: first the overall containment hierarchy and secondly the use of non-containment references exemplified by method calls (i.e. references between method call and respective method declarations).

Containment Hierarchies

Fig. 4 shows a sunburst chart representation of three packages of object-oriented code. One is synthetically generated, one is actual Java code taken from our EMF-fragments project, and one is randomly generated. The synthetically generated program comprises a fixed number of classes, each of which contains a fixed number of inner classes, fixed number of methods, etc. Multiplicity and choice of contained elements are determined by constants or values in a repetitive pattern. The result is a homogeneous repetitive non random structure. The actual Java code shows that containment in real code is varying: there are classes with more or less methods, methods can be short or longer. Tamai et al [10] suggest that multiplicities of containment references can be modeled with negative binomial distributed random variables. Our generator for random code uses such negative binomial distributed random variables. We chose parameters that yield in expected values that represent reasonable metrics for classes per package, methods per class, statements per methods, depth of expressions, etc. We chose metrics that resemble the empirically determined corresponding metric in the reference Java code of the prior example.

References and Dependencies

Fig. 5 show chord chart representations of method calls in three different object-oriented programs. In this chart each line represents a method call and the line end points represent calling and called method. Methods are clustered in blocks that represent the classes that contain them. The first program was randomly generated. We chose the called method for each method call uniformly from the set of all methods within the program. Hence, each class has a similar relative number of expected dependencies to each other class, including the calling class. The code taken from an actual program (again, the same EMF-fragments code as before) shows a different distribution. Calls of methods within the containing class are more likely and furthermore calls of methods within in certain other depending classes are more likely than calls of methods of other less depending classes. The last chart shows generated code gain. Now, we changed our generator to emulate the reference distribution of actual Java code. We do not chose methods uniformly from all methods, but put higher probability on methods of the calling class and of depending classes. We pseudo randomly chose pairs of depending classes based on the similarity of the hash code of their corresponding EMF objects.

5 Conclusions

We presented the domain specific language *rcore* and a corresponding compiler that aids clients in the development of generators that automatically create instances of

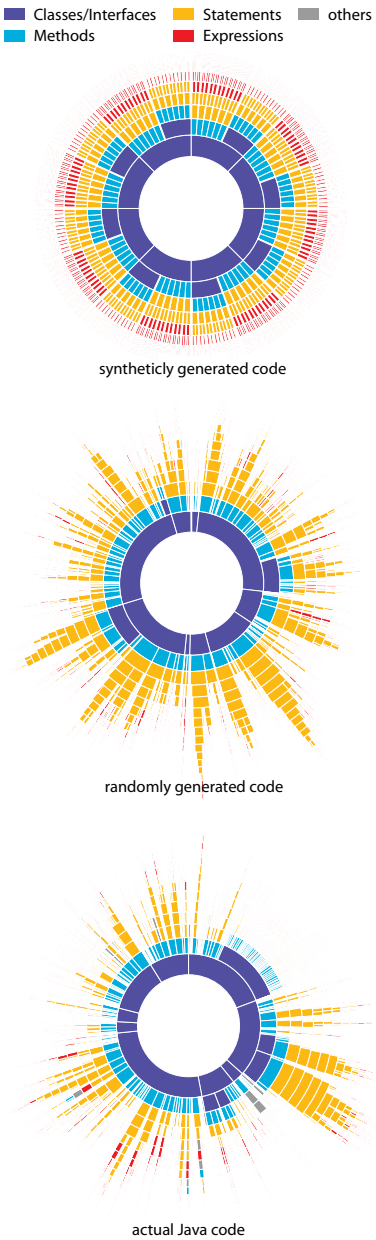

Fig. 4: Sunburst representations of containment hierarchies of generated and actual object-oriented code.

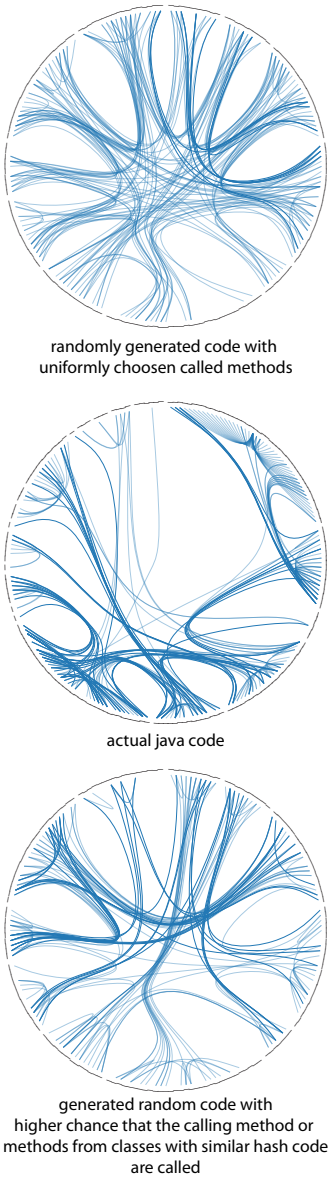

Fig. 5: Chord chart visualization of calling-called-method dependencies in generated and actual code.

given Ecore meta-models based on a set of generator rules. These parameterizable rules can be used to control the generation of model elements with random variables of certain probabilistic distributions. This gives clients randomness and configurability as means in their efforts to create instances of their ecore meta-models that are potentially unbiased, have a certain *shape*, and yet mimic real world models. Furthermore, *rcore* allows to describe arbitrary large models with a small set of generator rules. *Rcore*'s simple operational semantics of one executed element generating rule calling other element generating rule should lead to linear time complexity in the number of generated elements, unless the user defined functions used to determine the concrete feature values for the generated elements introduce higher complexities. Therefore, the actual amount of achieved bias, real world mimicry, configurability, and scalability depends on concrete generator descriptions. As future work, we need to create a larger set of case study generators and evaluate these generators for the proposed characteristics.

References

1. Barmpis, K., Kolovos, D.S.: Comparative analysis of data persistence technologies for large-scale models. In: Proceedings of the 2012 Extreme Modeling Workshop. pp. 33–38. ACM (2012)
2. Benelallam, A., Gómez, A., Sunyé, G., Tisi, M., Launay, D.: Neo4EMF, a scalable persistence layer for EMF models. In: Modelling Foundations and Applications, pp. 230–241. Springer (2014)
3. Brottier, E., Fleurey, F., Steel, J., Baudry, B., Le Traon, Y.: Metamodel-based test generation for model transformations: an algorithm and a tool. In: Software Reliability Engineering, 2006. ISSRE'06. 17th International Symposium on. pp. 85–94. IEEE (2006)
4. Ehrig, K., Küster, J.M., Taentzer, G.: Generating instance models from meta models. Software & Systems Modeling 8(4), 479–500 (2009)
5. Izso, B., Szatmari, Z., Bergmann, G., Horvath, A., Rath, I.: Towards precise metrics for predicting graph query performance. 2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013 - Proceedings pp. 421–431 (2013)
6. Mougenot, A., Darrasse, A., Blanc, X., Soria, M.: Uniform random generation of huge metamodel instances. In: Model Driven Architecture-Foundations and Applications. pp. 130–145. Springer (2009)
7. Pagán, J.E., Cuadrado, J.S., Molina, J.G.: Morsa: a scalable approach for persisting and accessing large models. In: Proceedings of the 14th international conference on Model driven engineering languages and systems. pp. 77–92. Springer-Verlag (2011)
8. Scheidgen, M., Zubow, A., Fischer, J., Kolbe, T.H.: Automated and transparent model fragmentation for persisting large models. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). vol. 7590 LNCS, pp. 102–118 (2012)
9. Sen, S., Baudry, B., Mottu, J.M.: Automatic model generation strategies for model transformation testing. In: Theory and Practice of Model Transformations, pp. 148–164. Springer (2009)
10. Tamai, T., Nakatani, T.: Analysis of software evolution processes using statistical distribution Models. Proceedings of the international workshop on Principles of software evolution - IWPSE '02 p. 120 (2002), http://portal.acm.org/citation.cfm?doid=512035.512063
11. Varró, G., Schürr, A., Varró, D.: Benchmarking for graph transformation. Proceedings - 2005 IEEE Symposium on Visual Languages and Human-Centric Computing 2005, 79–88 (2005)