Metric Driven Approach for Automatic Creation of Model Benchmarks

Victor Makarenkov and Mira Balaban

Computer Science Department
Ben-Gurion University of the Negev, Beer-Sheva 84105, ISRAEL
vitiokm@gmail.com, mira@cs.bgu.ac.il

Abstract. Model metrics measure various aspects of model complexity. They are used for evaluating the quality of models and anticipate management needs. Metrics can be useful also for directing model creation. The latter usage is particularly useful for automatic benchmark creation, since manual creation is exhausting and inflexible. Such benchmarks are needed for experimental evaluation of model processing algorithms, and for finding their strengths and weaknesses.
This paper presents analysis of model metrics, groups of model metrics, and a method for metric based automatic benchmark creation for software models. The model benchmark creation is implemented using the Alloy system. The contribution of this work is in providing means for analytical evaluation of model metrics, classification of model metrics into formal groups, and in using model metrics for automatic benchmarking of model algorithms.

Keywords:
Model metrics, meta-model, metric evaluation, metric classification, model complexity, Benchmark, model generation, UML, Alloy.

1 Introduction

Models lie at the heart of the emerging Model Driven Development (MDD) approach, in which software is developed by repeated transformations of models, which are intensively used. In this paradigm, models are no longer restricted design artifacts, but play a central role in the process of software development.

The growing role of models in software development results in a growing number of processing algorithms, targeting different aspcets of model reasoning: correctness [1], refactoring, modifications, comparison, answering queries and static analysis, testing, merging, splitting etc.

Rigorous algorithms evaluation needs, besides theoretical analysis, experimental evaluation. In this paradigm, no real world benchmarks (unlike in databases) exist. The lack of real world benchmarks is probably due to variability of the models, the relatively new status of the modeling paradigm, and lack of experimental research [2].

In order to narrow the gap, creation of synthetic benchmarks is needed. But such a creation needs parameters and metrics for evaluation. Obviously, different models possess different problems and different processing algorithms.

In this research we use metrics for model evaluation as measures for directing model benchmarking. We takle two major questions: (1) How to analytically evaluate a metric suite? Given an model processing algorithm, how to evaluate a suggested set of metrics as providing an appropriate measure? (2) Metric driven (automatic) creation of synthetic benchmark models.

For question (1): We adapt Weyuker's properties [3] for models. This adaptation provides an abstraction and leads to a criterion for determining how inclusive is a set of metrics. For question (2): We classify model metrics into groups and provide a *metric-specification pattern* for each group. Each pattern is associated with a framework for translation into an alloy specification, which functions as a model finder to generate sample models.

This paper is organized as follows: Section 2 reviews related research results; Section 3 adapts Weyuker's [3] properties for evaluating model metrics; Section 4 classifies metrics into groups, setting up a base for automatic models creation process; Section 5 demonstrates the automatic metric driven benchmark creation; Section 6 concludes the paper.

2 Background

Metrics are widely used in software development, as quantitative measures for budget planning, maintenance cost estimation, testing activities and quality prediction at early stages of software development. Metrics for model evaluation are largely inspired from metrics in object oriented software [4]. A well known suite of six metrics for object-oriented programs is that of Chidamber and Kemerer (C&K) [5]. The suite functions as a measure for class complexity, using metrics such as depth and breadth of class hierarchies, and coupling between classes.

Metrics of models are usually defined over the meta-model [6], and often specified in the Object Constraint Language (OCL) [7]. Baroni et al. [8, 9] modify the UML meta-model by creating the metrics as additional operations expressed in OCL. McQuilan [10] extends the work of Baroni, by decoupling the metrics definition from the UML meta-model, thereby generalizing the approach to any meta-model and any set of metrics. [11] suggest generic rules for generation of metrics for models of domain specific modeling languages. The generated metrics are written as OCL statements, and are tailored for the specific needs of a development process.

Mens and Lanza [12] use a generic meta-model, written as a *type graph*, where concrete models are its instances. They suggest three basic metrics that refer to the instance models as graphs. Their metrics count nodes, edges and path lengths in the instance graph, subject to associated constraints. They do not handle the issue of the metrics language, i.e., the language in which metric expressions are written. Based on their three basic metrics, a large number of standard object-oriented metrics can be defined. They also discuss high order

metrics, i.e., metrics that apply other metrics. Our approach, presented in section 4 has similarity with the [12] approach, in singling out basic kinds of metrics that support a wide variety of metrics.

Weyuker, in [3] suggests nine properties for evaluating complexity metrics of sequential software. The properties first specify essential characterization of metrics. Then, they try to characterize the impact of four typical software variations on the complexity of the resulting software. The software variations are: (1) Syntax variation; (2) Syntax similarity; (3) Reordering; (4) Combination.

The first three properties require that (1) a metric distinguishes between at least two programs; (2) a metric value does not identify infinity of programs as having the same complexity; and (3) at least two programs have the same complexity. The forth property requires that syntactic variation between semantically equivalent programs is distinguished by a metric. Three properties deal with the impact of program combination on the complexity. Property 5 requires that the complexity of a program component is not greater than that of the whole program. Property 6 deals with the syntactic interaction of combined programs. It requires that the combination of a program programs that have the same complexity, might result combined programs with different complexities. The ninth property states that the complexity of a combined program should not be less than the sum of the complexities of its components. The seventh property requires that statement reordering affects the program complexity, and the eighth property requires that renaming does not affect complexity.

Weyuker's properties are widely used for evaluation of software metrics. In particular, they have been adapted for metrics of object-oriented software. The single class metrics of C&K [5] were evaluated with respect to six properties of Weyuker. The relevance and applicability of Weyuker's properties for the evaluation of object-oriented software is discussed in [13]. In the next section we suggest an abstraction of Weyuker properties for any kind of software models.

3 Evaluation of Model Metrics Using Weyuker's Properties

Weyuker properties can be used for the evaluation of model metrics. For that purpose the four *software variations* on which they depend, should be adapted to apply to models, i.e., adapted into *model variations*. First we present an adaptation of the properties of Weyuker to metrics of software models, and then we discuss the meaning of the four model variations.

A model oriented variant of Weyuker's properties

1. **Property 1 - *Non-Coarseness.*** A metric μ satisfies *non-coarseness* if there exist two distinct models $M_1 \neq M_2$ such that $\mu(M_1) \neq \mu(M_2)$. This is an essential property of any metric as a measuring function.
2. **Property 2 - *Strengthened-non-Coarseness.*** A metric μ satisfies *Strengthened-non-Coarseness* if for every complexity value c, there is only a finite number of models M such that $\mu(M) = c$.

3. **Property 3 -** ***Non-fineness.*** A metric μ satisfies *non-fineness* if there exist at least two *distinct* models $M_1 \neq M_2$ such that $\mu(M_1) = \mu(M_2)$. That means that a metric can ignore some syntactic differences.
4. **Property 4 -** ***Syntax variation.*** A metric μ satisfies *syntax variation* if for some two different models $M_1 \neq M_2$ which are semantically equivalent $M_1 \equiv M_2$, the metric values are different $\mu(M_1) \neq \mu(M_2)$.
5. **Property 5 -** ***Combination monotonicity.*** A metric μ satisfies *combination monotonicity* if for every two models M_1 and M_2, $\mu(M_1) \leq \mu(M_1+M_2)$ and $\mu(M_2) \leq \mu(M_1 + M_2)$. $M_1 + M_2$ denotes the combination of M_1 and M_2. That is, the complexity of a sub-model cannot exceed the complexity of a whole model.
6. **Property 6 -** ***Syntax interaction in combination.*** A metric μ satisfies *Syntax interaction in combination* if there exist models M_1, M_2, M such that $\mu(M_1) = \mu(M_2)$ while $\mu(M_1+M) \neq \mu(M_2+M)$ or $\mu(M+M_1) \neq \mu(M+M_2)$. This property measures sensitivity to model combination.
7. **Property 7 -*****Reordering Sensitivity.*** A metric μ satisfies *reordering sensitivity* if there exist two models M_1, M_2 such that M_2 is obtained from M_1 by element reordering, and $\mu(M_1) \neq \mu(M_2)$. That is, the metric is sensitive to reordering.
8. **Property 8 -** ***Syntax similarity – Renaming.*** A metric μ satisfies *consistent renaming* if for every two models M_1, M_2 such that M_2 is obtained by renaming from M_1, $\mu(M_1) = \mu(M_2)$. That is, renaming does not affect complexity.
9. **Property 9 -** ***Complexity of combination.*** A metric μ satisfies *Complexity of combination* property if there exist two models M_1 and M_2 such that $\mu(M_1) + \mu(M_2) < \mu(M_1 + M_2)$. That is, syntax interaction in a combination might increase the complexity.

The Four Model Variations

Semantic variation: Models can vary syntactically, while being semantically equivalent. For example, in class diagrams, applying the transitivity of class hierarchy constraint yields class diagrams that are semantically equivalent, but have different class hierarchies: One class diagram might include the class hierarchies $C_1 \preceq C_2 \preceq C_3$, where $\preceq$ stands for class hierarchy, while the other might include also $C_1 \preceq C_3$.

Syntax similarity – renaming: Syntax similarity in models might have different realizations, the simplest being consistent element renaming. A more "courageous" syntactic modification might involve exchanging whole blocks, like exchanging the attributes of two classes.

Reordering: Reordering means changing an existing order. In sequential symbolic programs, the sequential ordering of statements is clear, and reordering means statement permutation. But in models, whose concrete syntax is visual, the model offering refers to the mutual inter-relationships among the model elements.

For example, in a class diagram model described by the meta-model in Figure 1, an association has 2 properties, each having a single class and 2 multiplicities, while a class might have multiple properties. Changing the mapping of properties to classes (moving an association end from one class to another) is a reordering operation (and clearly should affect the complexity of the class diagram).

Combination: Combination of software models raises problems of conflicts between the combined models. Different strategies have been suggested, for various models [14, 15, 16]. An intuitive *finger rule* that holds for most strategies is that they do not remove elements of the combined models. Therefore, in most cases, size metrics, that count elements satisfy the *Complexity of combination* property (no. 9).

Once the four model variation operations are determined, model metrics can be evaluated with respect to the nine Weyuker properties. Consider, for example, the class diagram model. Then the *Number of Classes* (NCM) metric satisfies satisfies property (1) and property (3), but does not satisfy property (2), since there might be infinity of class diagrams having the same number of classes. NCM satisfies the syntax variation property (no. 4) since there a class diagrams can include redundant classes that cannot be populated. As for the three combination properties (no. 5, 6, 9), *combination monotonicity* and *syntax interaction in combination* properties are satisfied: $NCM(M_1 + M_2) = NCM(M_1) + NCM(M_2) - \#(\text{common class of } M_1 \text{ and } M_2)$. Therefore, $NCM(M_i) \leq NCM(M_1 + M_2),\ i = 1,2$ thus satisfying combination monotonicity, and a case where M_1 and M have overlapping classes, where as M_2 and M do not, lead to satisfaction of syntax interaction in combination property. NCM satisfies the renaming and reordering properties since they do not affect the number of classes in the model.

4 Classification of Model Metrics

Metrics are functions that assign software products values in a comparable domain, usually numbers. Metrics can be used to measure complexity of software, or to check whether a given software product satisfies some *complexity constraints*, or to impose complexity constraints on a created software product. In this section we suggest classification of model metrics. The classification helps us in providing patterns of metrics, that are used, in the next section for automation of the metric driven model creation approach. We assume that the abstract syntax of the model under investigation is specified by a meta-model, in the form of a class diagram. As an example, we use the class diagram meta-model given in Figure 1.

The metrics classification includes three kinds: Size metrics, metrics that measure relationships between values of other metrics, and structure metrics. For each kind we provide a specification pattern. these kinds cannot be expressed in OCL since they require variables that range over the elements of the meta-

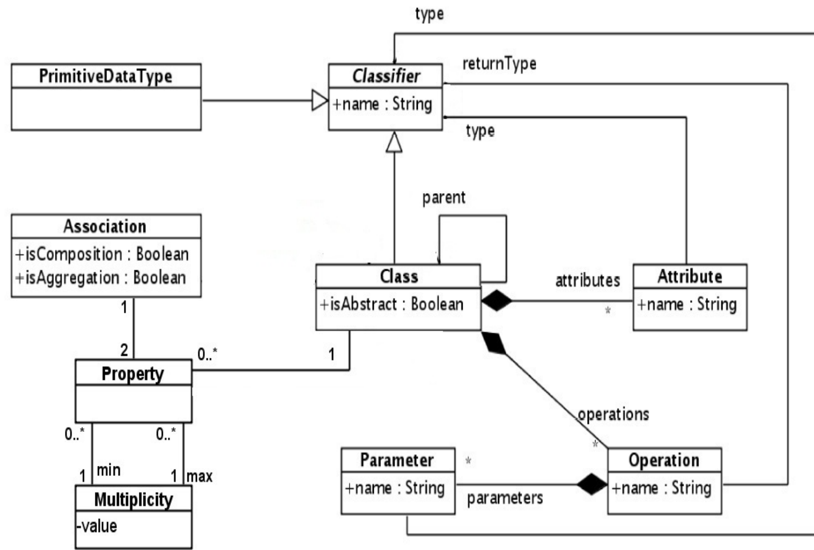


Fig. 1. Partial Class Diagram Meta-Model

model (while OCL supports only variables that range over instances of a class diagram)[1].

4.1 Size Metrics.

Size metrics measure complexity that emerges from the actual size of a model. They reflect the understanding that the size of a model affects its complexity. Examples are the metrics *Number of Classes* (*NCM*) and *Number of Associations* (*NASM*) of [6]. A generalization of these metrics add restrictions, such as *number of classes that have no super-class* or *Number of unary associations.*

A size metric has two parameters: An element (meta-class) of the meta-model, and a constraint. Its abbreviation is ***NO***, for (**N**umber of **O**bjects). Its general pattern is:

> $NO(X, \Phi)$, where X is an element of a meta-model and Φ is a boolean constraint[2].

For example the NCM metric is $NO(Class, true)$ where *true* is a predicate that is always true and the NASM metric is $NO(Association, true)$. In order to write size metrics with non-trivial constraints we need a language for expressing the constraints. Using OCL, the *number of classes that have no super-class* is $NO(Class,$ `no-super-class`$)$, where `no-super-class` is the invariant:

[1] Similar to the OCL patterns in [17].

[2] analogous to *where* statement in SQL

```
Context Class
inv: self.parent->isEmpty()
```

Similarly, the metric *number of associations with non-trivial multiplicity constraints*, i.e., multiplicity constraints different from {0..*} is $NO(Association,$ `non-trivial-multiplicity`), where `non-trivial-multiplicity` is the invariant:

```
Context Association
inv: self.property->min->select(x|x.value != 0)->size()>0 or
     self.property->max->select(x|x.value != "*")->isEmpty()
     ->size()>0
```

In [12], the meta-model (type graph) includes a meta-class *System* that represents the system. Therefore, they can represent size metrics with respect to this meta-class. For example, they capture $NO(Class, true)$ as $NC(s, System, contains, true)$.

4.2 Metric-Relationship Metrics

Metric-relationship metrics measure complexity that emerges from the inter-relationships between complexities measured by other metrics. They reflect the understanding that a sparse huge model (like a huge class diagram with no associations) has very small complexity, while a smaller dense model (like a smaller class diagram with a large number of constrained associations) might demonstrate a higher degree of complexity. In general, such metrics measure interaction between complexities measured by other metrics.

The main kind of inter-relationship metrics is the *ratio* metrics, i.e., metrics that measure the ratio between the values of other metrics. A ratio metric takes as parameters two metrics and their arguments. Its general pattern is:

$Ratio(\mu_1(params_1), \mu_2(params_2)) = \frac{\mu_1(params_1)}{\mu_2(params_2)}$ where μ_1 and μ_2 denote metrics and $params_1$ and $params_2$ are their parameters, respectively.

For example, $ratio(NO(Association,$ `non-trivial-multiplicity`), $NO(Class, true))$ is the class diagram ratio metric that measures the ratio between the number of associations with non trivial multiplicity constraints to the number of classes. In [18] we used a metric suite that includes the $NO(Class, true)$ and the $ratio(NO(Association,$ `non-trivial-multiplicity`), $NO(Class, true))$ to combine the effect of size and ratio, in an experiment for evaluating the frequency of the *Finite satisfiability* problem in class diagrams, and the performance of the ***FiniteSat*** algorithm. Another example is the *method hiding factor* metric proposed in [19], which sets the ratio of invisibilities of all model methods and total number of methods within the model. Other kinds of measuring metric interaction, like weighted balance exist, but we omit them in this paper.

4.3 Structure Metrics.

Structure metrics reflect complexity caused by internal structures in the model. An *internal structure* is a collection of inter-related model elements. The structure is created by the way the inter-relationships interact. For example, in a

class diagram, association cycles affect the complexity of the model[3]. Another example is the structure of class hierarchies, where the depth, width, and internal structure is meaningful. A tree structured hierarchy is considerably simpler than a graph structured one (like diamond hierarchies, as in Figure 2) that enables multiple inheritance. A structure metric is characterized by a *structure* and a

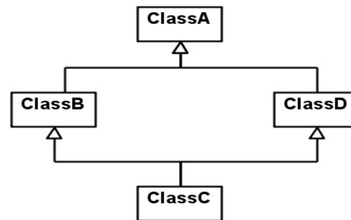


Fig. 2. Diamond class hierarchy structure

property that the metric measures. The specification of a structure metric needs an *anchor* class in the meta-model, with respect to which the structure is computed and evaluated. The anchor meta-class can have a method that computes the structure, and an attribute that evaluates the property. The general pattern is:

> $Str(X, struct, prop, mode)$, where X is a class in the meta-model, *struct* is an operation that computes the required structure with respect to an X instance, *prop* is a property of the structure, and *mode* is one of $minimum, maximum, average$.

For example, a structure metric for the maximal length association path starting from a given class in a class diagram is specified by $Str(Class, association_path, length, max)$. The smallest association cycle through a class in a class diagram is specified by $Str(Class, association_cycle, length, min)$. Similarly, $Str(Class, class_hierarchy_path, length, average)$ is the average depth of a class hierarchy structure starting from a model class. Diamond structures can be identified by comparing $Str(Class, parents, size, min)$ with $Str(Class, proper_ancestors, size, min)$. The generic metrics *NC* and *EC* of [12] can be specified as $Str(Class, contained, size, min)$ and $Str(Class, "related", size, min)$, respectively, where "*related*" can be *associated* or *aggregated* or *subclasses* or $super - classes$.

If the meta-model includes a class for the structure, e.g., a class *Generalization_set* for a class hierarchy structure, the pattern can be simplified into $Str(X, prop, mode)$.

Size metrics that involve structures require meta-classes for the structures. For example, the metric *number of association cycles* require a meta-class *Association_cycle*. Then, NO $(Association_cycle, true)$ specifies this metric.

[3] Association cycles with problematic multiplicities cause Finite Satisfiability [1]. Such structures identify a correctness pattern for class diagrams `http://www.cs.bgu.ac.il/~modeling/?page_id=32`.

5 Model Benchmark Automation

Experimental evaluation of algorithms that run over models require benchmarks for models. Since such benchmarks do not exist, there is a need to create synthetic benchmarks. Such a creation should be parameterized with specific metrics, since different algorithms affected by different models aspects.

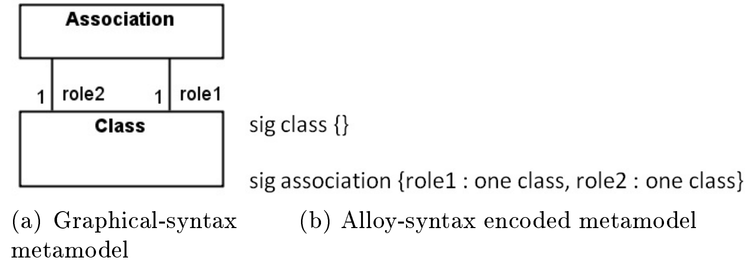
In [18] an attempt for synthetic manual benchmark creation is demonstrated. This approach is inflexible, and demands manual code writing for each and every new metric describing the benchmark set. In this section we make use of Alloy for automatic benchmark sets creation.

Alloy in short: Alloy [20] is a little language for describing structural properties. It can be used as *model finder*, based on a SAT solver. The *Alloy Analyzer* works by translating the model specified in Alloy language into a boolean expression, which is analysed by SAT solvers embedded within the Alloy Analyzer. A user-specified scope on the model elements bounds the domain, making it possible to create finite boolean formulas for the evaluation by SAT solvers. The Alloy Analyzer offers two analysis methods. The first is simulation and the second is assertion checking [21]. In this section we adopt Alloy for generating models along specified metric values. Using Alloy, a model can be built as follows:

- **Signatures**. Signatures are used to model classes of objects, that is sets.
- **Predicates**. Predicates are the means for finding model instances: Alloy produces instances that satisfy the predicate. Asking Alloy to find instances is similar to finding a model of a given schema.
- **Facts**. Facts are used to impose constraints on models. Facts are global, and always apply. Put otherwise, every instance of a model must satisfy the facts.
- **Functions**.Function is an expression that returns a result.
- **Assertions**. These are assumptions about the model that you can ask the analyzer to find counter-examples of.

After the model is built, its assertions can be verified, with an attempt to find a counter-example. Alloy performs an exhaustive search in a *limited* space eliminating the possibility of missing an instance. The scope of instance search should be specified.

In this section the idea of using a model-checker for instantiating a meta-model is implemented. That is, benchmarking according to given metric values, where metrics choise is influenced by algorithmic considerations, and examined with Weyuker's properties [3] relatively to algorithmic equivalence. As noted by McQuilan and Power [10], defing metrics is a meta-modelling activity. Hence, the first step in generating models is to define a meta model in Alloy. Second, the Alloy Analyzer is asked to find a model with the given values to the specified metrics (thus specifing the scope of the search). Actually, it is impossible to find an instance for some model without specifying scope and thereby giving values to metrics. It is also possible to specify exactly how many different instances of a meta-model should be found, generating appropriate task sample for benchmarking.



(a) Graphical-syntax metamodel

(b) Alloy-syntax encoded metamodel

Fig. 3. A simplified partial meta-model of UML in Alloy syntax.

Figure 3.b shows the Alloy encoding for the meta-model in Figure 3.a. The meta-model can be produced in two ways:

- The first, straightforward way, like the one in figure 3(b) is hand-written from scratch in Alloy Analyzer tool. In this case we must write all parts of our meta-model with needed constraints as alloy *facts*, and then find instances.
- The second is with the UML2Alloy [23] tool, which transforms part of UML/OCL class diagrams to Alloy specifications.

Recently, [24] demonstrated a way to analyse UML class models by transforming them into Alloy models, and then transform the Alloy produced instances back into UML object diagrams. [25] uses Alloy for modeling package merging in UML. In [26] the Alloy Analyzer [27] is used as a tool for verification of a newly introduced model for Web Applications. [28] uses Alloy for model-checking of visual design notations.

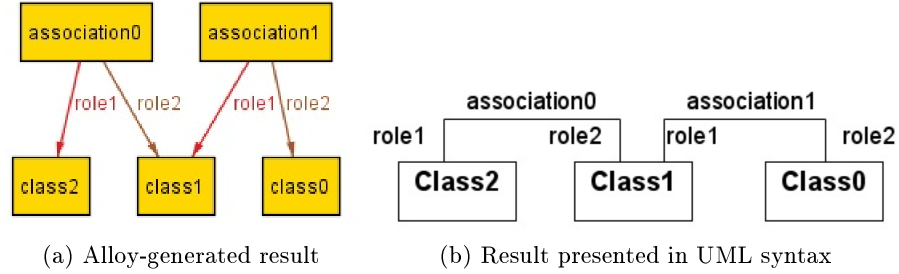
5.1 Automatic model generation for Size Metrics

For size metrics, we must encode the needed meta-model into Alloy Analyzer and specify the exact (or upper bound) number of meta-model elements to generate as well as the number of instances to generate in the overall process.

```
pred Generate{}

run Generate for exactly 3 class, exactly 2 association
```

Fig. 4. Instance finding.

Consider the meta-model in figure 3.a. The meta-model contains two class elements *Class* and *Association*, such that each instance of *Association* associated with two instances of *Class* via association $role1$ and $role2$. By specifying the *Generate* predicate and applying the *run* command, see Figure 4, we ask the



(a) Alloy-generated result (b) Result presented in UML syntax

Fig. 5. Generated result for size metrics

Alloy Analyzer to generate one instance of the metal model, such that there are 3 one instances of *Class* and two instances of *Association*. These can be seen as examples for number of classes and number of association metrics.

The result of such generation is shown in figure 5. Following the general process of generating models by metrics, we summarize the process of generation instances along regular size metrics with the following steps:

1. Specify the meta-model in Alloy Analyzer.
2. Select the Metric suite. That is, select the classes which number of instances you want to specify in the generated model.
3. Specify an empty *Generate* predicate.
4. Use *run* command, specifying number of instances to generate and exact or upper bounds for every element chosen earlier.

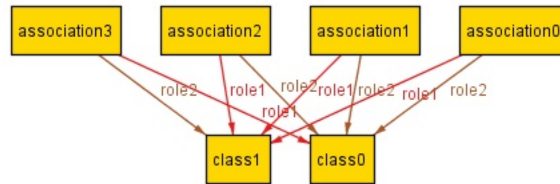
5.2 Automatic Model Generation for Metric-Relationship Metrics

In order to enforce some constraint on metric-relationship metric values, we need to specify a specific *fact* statement in Alloy. For example, consider again the meta-model in figure 3. Suppose we want to set a value for a ratio metric $\frac{NCM}{NASM} = \frac{1}{2}$, that is, the number of *Association* instances is twice the number of *Class* instances. In order to set a value to this metric, we use the *fact* keyword of Alloy, see figure 6.

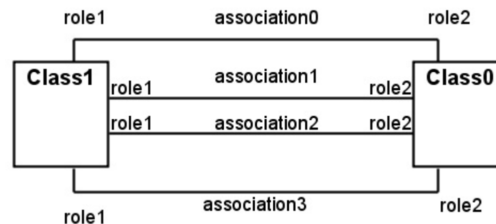
```
fact {let numclass = #class| let numassoc = #association| numassoc.div[numclass]=2}

pred Generate{}

run Generate for exactly 2 class,  10 association
```

Fig. 6. Addition of special fact statement constraining the ratio value

The result is demonstrated in Figure 7: The alloy-generated model presented in Figure 7(a) and its UML syntax version is presented in Figure 7(b).



(a) Alloy-generated result



(b) Result presented in UML syntax

Fig. 7. Generated result for ratio metrics

6 Future Work

In this paper, we presented a method for automatic metric driven benchmark creation for model correctness algorithms. A basis for function metric-driven language, for benchmarking specification and creation is set up.

This work can be expanded in a straightforward way to several directions, such as developing a fully functional language for metrics description, based on metrics patterns. As a result a proprietary language for model level metrics might be developed. It is also possible to develop an engine, for automatically translating (compiling) the language into a model checker, such as Alloy. Developing metrics which describe hierarchial data, with possibly unbounded size and recursive definition. For example: graphs, cycles, lists etc,. During this research, we noticed it is very difficult to describe complex hierarchial data structures with Alloy. It might be interesting to examine other model checkers, which can have more expressive language.

Acknowledgment: We would like to thanks Shahar Maoz for his helpful advice.

References

[1] Balaban, M., Maraee, A.: Finite satisfiability of uml class diagrams with constrained class hierarchy. (Submitted) (2011)

[2] Tichy, W.F.: Should computer scientists experiment more? IEEE Computer **31** (1998) 32–40

[3] Weyuker, E.J.: Evaluating software complexity measures. IEEE Trans. Softw. Eng. **14** (1988) 1357–1365

[4] Genero, M., Piattini, M., Caleron, C.: A survey of metrics for uml class diagrams. Journal of Object Technology **4** (2005) 59–92

[5] Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. IEEE Trans. Softw. Eng. **20** (1994) 476–493

[6] Kim, H., Boldyreff, C.: Developing software metrics applicable to uml models. In: 6th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering. (2002)

[7] Object Management Group: UML 2.0 Object Constraint Language Specification. (2006)

[8] Baroni, A., Braz, S., Abreu, F.: Using OCL to formalize object-oriented design metrics definitions. In: ECOOP'02 Workshop on Quantitative Approaches in OO Software Engineering. (2002)

[9] Aline Lúcia Baroni, F.B.e.A.: A formal library for aiding metrics extraction. In: International Workshop on Object-Oriented Re-Engineering at ECOOP. (2003)

[10] McQuillan, J.A., Power, J.F.: Towards re-usable metric definitions at the meta-level. In: PhD Workshop of the 20th European Conference on Object-Oriented Programming, Nantes, France (2006)

[11] Engelhardt, M., Hein, C., Ritter, T., Wagner, M.: Generation of Formal Model Metrics for MOF based Domain Specific Languages. In: OCL Workshop, MODELS. (2009)

[12] Mens, T., Lanza, M.: A graph-based metamodel for object-oriented software metrics. Electronic Notes in Theoretical Computer Science **72** (2002) 57 – 68 GraBaTs 2002, Graph-Based Tools - First International Conference on Graph Transformation.

[13] Misra, S., Akman, I.: Applicability of weyuker's properties on oo metrics: Some misunderstandings. Computer Science and Information Systems **5** (2008) 17–23

[14] Dirk Ohst, Michael Welle, U.K.: Merging uml documents. Interner Bericht (2004)

[15] Mens, T.: A state-of-the-art survey on software merging. IEEE Trans. Softw. Eng. **28** (2002) 449–462

[16] Rubin, J., Chechik, M., Easterbrook, S.M.: Declarative approach for model composition. In: MiSE '08: Proceedings of the 2008 international workshop on Models in software engineering, New York, NY, USA, ACM (2008) 7–14

[17] Wahler, M., Basin, D., D. Brucker, D., Koehler, K.: Efficient analysis of pattern-based constraint specifications. Software and Systems Modeling **9** (2010) 225–255

[18] Makarenkov, V., Jelnov, P., Maraee, A., Balaban, M.: Finite satisfiability of class diagrams: practical occurrence and scalability of the finitesat algorithm. In: MoDeVVa '09: Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation, New York, NY, USA, ACM (2009) 1–10
[19] Abreu, F.B.e., Melo, W.: Evaluating the impact of object-oriented design on software quality. In: METRICS '96: Proceedings of the 3rd International Symposium on Software Metrics, Washington, DC, USA, IEEE Computer Society (1996) 90
[20] Jackson, D.: Alloy: a lightweight object modelling notation. ACM Trans. Softw. Eng. Methodol. **11** (2002) 256–290
[21] Anastasakis, K.: Uml2alloy reference manual. (2009)
[22] Makarenkov, V.: Metric driven approach to benchmarking model correctness algorithms. Master's thesis, Department of Computer Science, Ben Gurion University of the Negev (2011)
[23] Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: Uml2alloy: A challenging model transformation. In: In: ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS, Springer (2007) 436–450
[24] Shah, S.M.A., Anastasakis, K., Bordbar, B.: From uml to alloy and back again. In: MoDeVVa '09: Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation, New York, NY, USA, ACM (2009) 1–10
[25] Zito, A., Dingel, J.: Modeling uml2 package merge with alloy. In: First Alloy Workshop, colocated with the Fourteenth ACM SIGSOFT Symposium on Foundations of Software Engineering. (2006)
[26] Bordbar, B., Anastasakis, K.: Mda and analysis of web applications. In: Proceeding of VLDB Workshop on Trends in Enterprise Application Architecture (TEAA 2005) Lecture notes in Computer Science. Volume 3888. (2005) 44–45
[27] Analyzer, A.: Alloy analyzer web site : http://alloy.mit.edu. (2010)
[28] Simons, A.J.H., Fernandez, C.A.F.y.: Using alloy to model-check visual design notations. In: ENC '05: Proceedings of the Sixth Mexican International Conference on Computer Science, Washington, DC, USA, IEEE Computer Society (2005) 121–128
[29] Simons, A.J.H.: Object discovery: A process for developing applications. (1998)