An MDE Performance Testing Framework Based on Random Model Generation

He Xiao, Zhang Tian, Hu Changjun, Ma Zhiyi, Weizhong Shao

Journal of Systems and Software 2016

Most of the papers available from this document appear in print, and the corresponding copyright is held by the publisher. While the papers can be used for personal use, redistribution or reprinting for commercial purposes is prohibited.
An MDE performance testing framework based on random model generation

Xiao He a,1, Tian Zhang b, Chang-Jun Hu a,1, Zhiyi Ma a,1, Weizhong Shao a,1

a School of Computer and Communication Engineering, University of Science and Technology Beijing, Beijing 100083, P.R. China
b Key Laboratory of High Confidence Software Technologies of Ministry of Education (Peking University), Beijing 100081, P.R. China

Article history:
Received 20 December 2014
Revised 22 September 2015
Accepted 20 April 2016
Available online 22 April 2016

Keywords:
Model-related operation
Performance testing
Model generation
Model-driven engineering

ABSTRACT

The scalability of model-related operations (e.g., model transformations), when they are to be applied in industrial model-driven engineering, becomes an important issue. However, there is a lack of an automated performance testing framework for those operations, since the existing ones for ordinary programs are ill-suited. Such a framework is required to provide the function of creating and organizing test cases, and the ability of generating test input of large size automatically, because large scale models are not widely available, making it hard to test the performance and coverage of those operations without any bias. This paper proposes a performance testing framework, integrated with a random model generation algorithm, for model-related operations. The framework, based on a test model, can be used to specify and arrange test cases into test suite. And the model generation algorithm can generate a random model correctly and efficiently, according to the metamodel and user-defined constraints. Finally, we present two case studies, one experiment in randomness, and two experiments in generation efficiency to evaluate the framework and algorithm. Results show that the framework is competent to support performance testing of model-related operations, and the algorithm is random and efficient enough to generate test data for performance testing.

© 2016 Elsevier Inc. All rights reserved.

1. Introduction

Model-Driven Engineering (MDE) employs models to drive the development and the maintenance of software systems. The model, the core artifact in MDE, serves as the abstraction of the software system. Then, a number of model-related operations, such as model refactoring, model synchronization, model composition, and model-to-code transformation, are applied to models to automate the development process.

Due to the increasing complexity of the system, software models are becoming larger and more complicated than they were ever before, consequently, consume much more processing time. How to query, analyze, convert, and merge large models efficiently has become a key factor. Especially in such an era of Big Data, handling big models extracted from volumes of codes or structural documents within a reasonable time is an essential ability for those model-related operations.

For example, when runtime model is applied to maintaining a running system Song et al. (2011), a bidirectional model transformation is used to keep the runtime architecture model, which reflects the logical structure of the running system, and the runtime system model, which reflects the actual structure, consistent. Developers can modify the runtime architecture model, and the changes will be propagated by the transformation to the runtime system model to affect the actual system. However, if the transformation could not be done efficiently, the system may have entered another state before the changes are produced by the transformation. Consequently, a system failure may come about when the changes are being applied, since the changes may not be valid anymore in the new system state.

Hence, the performance of a model-related operation in MDE must be systematically evaluated before it is put into practice. Evaluating the performance enables us to find out the limitations and bottlenecks of those operations for further improvement. However, there is a lack of tool support that can manage and facilitate performance testing of those operations. Besides, most of those operations are declarative, and are based on the type system of Meta Object Facility (MOF, Object Management Group (2011)) or Ecore (an industrial implementation of MOF)1 and Object Constraint
Language (e.g., OCL, Object Management Group (2012)) rather than common programming languages. Due to these particularities, current performance testing tools for ordinary application systems are not directly applicable to them.

One of the essential functions that must be provided by a performance testing tool is to generate large-size test data (i.e., the input model) automatically, since manually establishing an input model is an error-prone and time-consuming procedure. The basic requirements of generating models used in performance testing are listed as follows: (1) **Correct**: The generated model should conform to the syntactic constraints. (2) **Efficient**: The input data should be generated efficiently in order to test the efficiency of a model transformation. (3) **Randomized**: The model should be generated in a stochastic way to capture the average performance of an operation. (4) **Configurable**: The generation process should be configurable, e.g., users must be able to control the amount of elements.

This paper proposes a testing framework that meets the basic requirements of performance testing for model-related operations. It provides the abilities of defining, organizing, and performing the test cases. The paper also proposes a randomized model generation approach, which has been integrated into the framework to facilitate the test input generation. In this approach, all the elements and relationships are produced randomly within a reasonable amount of time. In addition, during this process, all metamodel-implied syntactic constraints and some semantic constraints are taken into account to assure the correctness property. It also supports some user-defined constraints guiding the generation process for better configurability. Two case studies are presented in this paper to demonstrate how to employ this framework to evaluate the performance of model-related operations. Besides, the results of three experiments are also introduced to show that our approach is efficient in generating a large correct model randomly.

The following paper is structured: Section 2 presents our performance testing framework; Section 3 defines some basic concepts and constraints for random model generation; Section 4 proposes a randomized and efficient model generation algorithm integrated into the testing framework, used to produce test data automatically; Section 5 presents two case studies and three experiments to demonstrate the feasibility and the usability of our framework, and to evaluate the randomness and the efficiency of our model generation algorithm; Section 6 discusses some issues about our approach; Section 7 compares our approach with other related work; at last, conclusion and future work are presented.

### 2. Performance test framework

#### 2.1. Framework overview

Testing the model-related operation, which is actually a program, becomes more and more important in model-driven engineering. There are two basic roles involved in this iterative task: developers and test engineers. Fig. 1 shows how they interact with each other.

1. The developer submits their model-related operation (i.e., SUT) to the test engineer.
2. The test engineer writes a test plan and creates initial test cases.
3. For each test case, test engineer must construct a valid test input data.
4. The test engineer selects some test cases and arranges them into a test suite, and performs it.
5. The test engineer collects the results and reports them to the developer.
6. The developer refines the program and then resubmits it to the developer.
7. The test engineer performs the test suite again to evaluate how much the operation has been improved, and then he or she may append some new test cases to the suite to test it incrementally and iteratively.

Hence, our framework, aiming at such a testing process, is required to own the following abilities: (1) defining and storing the test cases; (2) constructing the test data automatically; (3) arranging and performing test cases; (4) monitoring and analyzing the execution of a test case; (5) collecting and reporting results.

The architecture of our test framework is presented in Fig. 2. We have to emphasize that this framework is extensible,
because it includes some replaceable components. There are six components:

- **Test controller** serves as a control center, which is in charge of organizing the performance testing process. It requires three interfaces: 1) test case and test suite management interface; 2) test execution interface; 3) result report interface. When the user wants to manage test cases and suites, the controller will invoke the test suite management interface to serve the required functions (as shown in Fig. 3(a)). When a single test case or a test suite is to be performed, the controller will invoke the test execution interface to launch the SUT and to collect the results (as shown in Fig. 3(b)). Note that to perform a test suite the execution interface will be requested once for each test case included in the test suite. After collecting the test results, the controller will finally invoke the report interface to present them.

- **Test suite management**, an implementation of test case and test suite management interface, provides the function of specifying cases and managing test suites based on the Test Model for model-related operations, which will be discussed in Section 2.2. This component supports regression testing. It invokes a generator interface to produce test data when creating a test case (Fig. 3(a)).

- **Test data generator**, which implements the generator interface, can be used to generate a large model randomly. This component is replaceable so that test engineers can employ other generation strategies to produce test data. However, this paper proposes a randomized model generation algorithm for this specific task, which is discussed in Sections 3 and 4.

- **Execution adapter** realizes the test execution interface. It can be reckoned as a bridge to the concrete execution engine on which the SUT can be executed. Model-related operations may not be implemented using general-purpose programming languages, such as Java and C++. Their execution might rely on various kinds of execution engines. For example, a model comparison using EMF (Eclipse Modeling Framework) Compare 2 can be performed directly on JVM, while an ATL 3 transformation could only be executed on ATL Virtual Machine. This component is responsible for communicating with execution engines so as to invoke the SUT. It also requires a monitoring and analysis interface to trace the runtime information and to analyze the performance of SUT. We have implemented a default execution adapter, which can execute an SUT according to the launch configuration 4 on Eclipse platform. It prepares the input data based on a test case and then invokes the launch configuration to perform the SUT.

- **Monitoring and analysis adapter**, which implements a performance analysis interface, is invoked by Execution Adapter to monitor and analyze the performance of the SUT. A default adapter has been realized. It treats the SUT as a black box and calculates the span between the starting and the finishing time. It is also possible to realize a more fine-grained monitor for a particular kind of operations, for example, a monitor that can record the execution time of each rule in an ATL

---

2 EMF Compare Project: https://www.eclipse.org/emf/compar.
3 ATL Transformation Language: https://www.eclipse.org/atl/.
4 A program could be executed on Eclipse platform after a launch configuration has been established for it.
transformation would be conducive to finding the performance bottleneck.

- Report adapter is responsible for displaying the test results. The default implementation prints the results onto the console. It can be substituted for another one that can present results diagrammatically.

2.2. Test model

The foundation of this framework is the Test Model, a domain-specific model of testing model-related operations. It is used to define and manage test cases. The test controller can interpret a test model and perform tests automatically. Fig. 4 shows its definition.

TestProject is the root class of this domain model. When developers submit a new SUT, test engineers would create an instance of TestProject. It has three attributes: name of this project, inputBase, and outputBase. inputBase and outputBase, whose types are URI, are used to locate the folders containing all the input model files and the expected output model files respectively.

To test the SUT, we must execute it. As mentioned above, those model-related operations may only be executed on their own engines, and our framework must be able to invoke those engines. ExecutionSpecification in Fig. 4 is used to specify the essential information needed by ExecutionAdapter in Fig. 2 to configure an engine. Each subclass of ExecutionSpecification corresponds to a particular ExecutionAdapter which is responsible for interpreting the information contained in the configuration. LCBasedSpecification, a subclass of ExecutionSpecification depicted in Fig. 4, is used by the default execution adapter mentioned in Section 2.1, where the first two characters LC stand for launch configuration. Its attribute configurationName is the name of a launch configuration stored in Eclipse platform. The default execution adapter will invoke the configuration whose name is equal to this value.

If the program associated with the launch configuration has inputs from files, it is needed to specify where the inputs are read by creating instances of ExecutionParameter and linking them to the LCBasedSpecification element with inputs relationships. The name of ExecutionParameter is a unique identifier of the input file, and uri is the file path where the program will import data. It is also possible to use ExecutionParameter to define output files of the program by connecting it to ExecutionParameter element with outputs relationship, when the output is used for result analysis.

A TestProject contains a number of TestCases. Without losing generality, aTestCase consists of an id, a set of actual inputs, and a set of expected outputs.

Actual input of a TestCase is specified by TestData. Its uri specifies a file that serves as input of the SUT. It also refers to an ExecutionParameter that serves as the formal parameter. When the TestCase is performed, the file indicated by uri of TestData will be copied to the location specified by the uri of the corresponding formal parameter so that the SUT can read it.

Expected output of a TestCase is denoted by ExpectedOutput. It contains a set of URIs each of which denotes a file containing the expected output. If the SUT can produce one of those results, we think it is correct. Note that although comparing the actual output with the expected output is not necessary for performance testing, embedding this concept in the test model enables further functional testing using our framework.

Each TestProject can include a number of TestSuites. Each TestSuite consists of a set of TestExecutions, which represents one execution of a TestCase. It has a boolean attribute named passed indicating if the SUT passes through the TestCase. In the mode of regression test, our framework will copy all the TestCases referred by the previously created TestSuite to the newly established one so that those TestCases could be tested again.

When constructing a TestCase, we must be able to generate the input data automatically. To do so, we can employ a test data generator. GeneratorConfiguration specifies the configuration information of a certain test data generator. Each subclass of GeneratorConfiguration corresponds to an implementation of a data generator, which can generate a test input according to the configuration. RandomDataGenerator is a subclass of GeneratorConfiguration that works with the default test data generator in our framework. It is related to a generation ConfigurationModel which will be defined in Section 4.

3. Definitions and generation constraints

As mentioned in the previous section, one must be able to construct the test input automatically when creating a test case in our framework. To propose the model generation approach, this section defines some basic concepts and constraints used in this process.
3.1. Definitions

Definition 1 (Metamodel). Without losing generality, a metamodel $M$ can be formally defined as follows:

$$M = (T, H, A, R, C, \text{assoc}, \text{mult}, \prec)$$  (1)

In the definition,

- $T$ is the set of classes. Each class represents a type of elements.
- $H$ is the set of the abstract classes, and $H \subseteq T$.
- $A$ is the set of attributes. Each attribute $a$ in $A$ can be defined as a signature $a : t_a \rightarrow d$, where $a$ is the identifier, $t_a \in T$, and $d$ is a primitive data type. The symbol $l_d$ denotes all the possible values whose types conform to $d$.
- $R$ is the set of references among types. Each reference represents a type of relationships among elements.
- $C$ is the set of containment references, and $C \subseteq R$.
- assoc is a function $R \rightarrow T$. It maps each reference $r \in R$ to a pair (src, tar) of classes, which indicates the source and the target of $r$. For simplicity, if assoc$(r) = (s, t)$, source$(r) \equiv s$ and target$(r) \equiv t$.
- mult is a function $R \rightarrow N^2 \times N^2$, which specifies the multiplicity of each reference, where $N$ signifies all non-negative integers (including $-\infty$). For a reference $r \in R$, if $\text{dom(r)}$ and $\text{lt(r)}$ determine the lower and the upper bound of the source end of $r$, respectively, and $\text{lt(r)}$ and $\text{ut(r)}$ determine the lower and the upper bound of the target end of $r$, respectively, where $(\text{lt(r)}, \text{ut(r)}) = \text{mult}(r)$.
- $\prec$, denoting the generalization hierarchy, is a partial order on $T$. If $C_1 \prec C_2$, $C_1$ is a child (descendant) class of $C_2$. If $C_1 \prec C_2$, $C_2 \prec C_1$.

For any class $c$,

$$\cup_c \equiv \{ p| c \leq p \} \quad \cap_c \equiv \{ p| p \leq c \}$$

Definition 2 (Model). A model $M$ conforming to a metamodel $M$ can be formally defined as follows:

$$M = (E, L, \text{type}_E, \text{type}_L)$$  (2)

where $E$ is the element set, $L$ is the relationship set, $\text{type}_E$ is a function $E \rightarrow T$ mapping an element to its type, and $\text{type}_L$ is a function $L \rightarrow R$ mapping a relationship to its type.

Suppose $M = (E, L, \text{type}_E, \text{type}_L)$, we have the following definitions:

- For an element $e$ and a class $t$, $e \in e \in E \wedge \text{type}_E(e) = t$.
- For an element $e$ and a class set $X$, $e \in e \in E \wedge \text{type}_E(e) \in X$.
- For an element $e \in e \in e \in E \wedge \text{type}_E(e) \in X$.
- For elements $a, b$ and a reference $r$, we say $(a, b) \in M$ if $(a, b) \in L \wedge \text{type}_L((a, b)) = r$. We also say $(a, b)$ is a $r$-relationship. From the definition of inheritance, if $(a, b) \in M$, $\text{type}_L(a) \in \text{type}_L(b) \in \text{type}_L(b)$.
- For a reference $r$,

  $$\text{ran}_r(x) \equiv \{(x, y) \in r \in M\} \quad \text{dom}_r(x) \equiv \{y \in x \in M\}$$

- For a set of references $Y$,

  $$\text{ran}_r(Y) \equiv \{(x, y) | (x, y) \in e \in E \wedge r \in Y\}$$

  $$\text{dom}_r(Y) \equiv \{y | (x, y) \in e \in E \wedge r \in Y\}$$

If $\forall r \in Y$, the $\text{ran}_r(e)$ and $\text{dom}_r(e)$ are defined; and if $\forall r \in Y$, $\text{ran}_r(e)$ and $\text{dom}_r(e)$ are undefined.

- For a set $G$ of references, if $(x, y) \in G$, we say $x \sim_G y$; if $x \sim_G y$ and $y \sim_G z$, then $x \sim_G z$.

3.2. Generation constraints

In this paper, model generation is regarded as a process of producing a model $M = (E, L, \text{type}_E, \text{type}_L)$ according to a metamodel $M = (T, H, A, R, C, \text{assoc}, \text{mult}, \prec)$ and a set of options, i.e., some constraints. Those constraints are used to guide the generation process.

Syntactic constraints. A generated model is said to be correct if it conforms to the syntactic and the semantic constraints imposed by the metamodel. A metamodel $M$ imposes three kinds of syntactic constraints which can easily be extracted from the definitions presented in Section 3.1:

- **Element syntactic constraint**: the element type must be valid, i.e.,

  $$\forall e \in E \rightarrow \text{type}_E(e) \in T - H$$

- **Attribute syntactic constraint**: the type of the attribute value must be valid, i.e.,

  $$\forall a, e(a : t \rightarrow d) \in A \wedge e \in \text{type}_E(e) \in \text{type}_E(a)$$

- **Relationship syntactic constraint**: the relationship type and the multiplicity must be valid, i.e.,

1. $\forall (a, b) \in L \rightarrow \text{type}_L((a, b)) \in R$;
2. $\forall r \in R$, when $\text{mult}(r) = ((\text{lt}(r), \text{ut}(r))) = \text{mult}(r)$,

   $$\text{ran}_r(a) \in \text{ran}_r(b) \in \text{dom}_r(a), \text{ut}(r)$$

   $$\forall b \in \text{ran}_r(a) \rightarrow \text{dom}(b) \leq \text{us}\}$$

Semantic constraints. A correct model must also conform to the semantic constraints implied by the metamodel, though they may not be defined explicitly. We identify four most common semantic constraints on any set $G$ of references:

- **Reflexivity** $G$ is non-reflexive implies $\forall (e, e) \in (e, e) \in cM \rightarrow (e, e) \notin G$.
- **Ordering** $G$ is ordered implies for any two elements $e_1, e_2$ in $M$ the following condition always hold $e_1 \prec e_2 \rightarrow e_2 \prec e_1$.
- **Necessary** if $G$ is source-necessary, then $\text{ran}_G(a)$ is defined, $\forall (a, b) \in \text{ran}_G(a)$, $\forall (a, b) \in \text{dom}_G(b)$ is defined, $\forall (b, b) \in \text{dom}_G(b)$.
- **Uniqueness** if $G$ is target-unique, then $\text{ran}_G(a)$ is defined, $\forall (a, b) \in \text{ran}_G(a)$, $\forall (b, b) \in \text{dom}_G(b)$.

The four constraints above can also be applied to a single reference $r$ by regarding $r$ as a singleton list $[r]$.

For example, the set $C$ of all containment references is non-reflexive, ordered, source-necessary (except for the root element), and source-unique; inheritance is non-reflexive and ordered. By default, a reference is reflexive, not ordering, not necessary, and not unique.

Range constraints. When the model is going to serve as the input of performance testing, we must control its size, including the amounts of elements and relationships, and the value domains of attributes. We term these kinds of constraints range constraints. Our approach supports three range constraints:

- **Element range constraint**: for any class $c$, $\text{range}_c$ denotes an element range constraint on $c$, which prescribes that $[e \in cM]$ must be a value contained in $\text{range}_c$.
- **Relationship Range Constraint**: for any set $G$ of references, $\text{range}_G$ denotes a relationship range constraint on $G$, which prescribes that $[(a, b) \in cM]$ must be a value contained in $\text{range}_G$.
- **Value Range Constraint**: for any attribute $a : t \rightarrow d$, $\text{range}_a$ denotes a value range constraint, which limits the value domain of $a$, i.e., $\forall e \in \text{range}_c(M(a) \in \text{range}_a)$.
It is worthwhile to notice that range constraints might not always be satisfied if they conflict with others. For instance, a relationship range constraint may be defined to restrict the amount of relationships. If the required amount is smaller or larger than the model could have, our approach would determine to break this constraint at runtime to create a correct output.

4. Approach to test data generation

The overview of our approach can be depicted as Fig. 5 with four major phases below:

The first phase is to configure the generation. In this phase, users establish a configuration model, containing reference semantic constraints and user-defined constraints. User-defined constraints include both range constraints and extra constraints. The configuration model and the metamodel are used to guide model production.

The second phase is the element and attribute generation phase. Based on the syntactic and the range constraints, our approach creates model elements and sets their attributes.

In the third phase, all relationships will be generated. This phase contains three sub-phases: 1) instantiating containment references (i.e., ordered, non-reflexive, source-necessary, and source-unique references); 2) instantiating constrained references according to semantic constraints and range constraints derived from the configuration model; 3) instantiating normal references. Although there are three sub-phases, all of the three employ a unified algorithm with different parameters to handle relationship generation.

The last phase is the validation phase. All user-defined extra constraints will be checked in this phase because our approach could not solve those constraints during model generation. If the produced model satisfies those constraints, it will be returned as the final result. Otherwise, our approach will return an empty model and report errors.

4.1. Configuration

This subsection discusses how to configure the model generation process with a configuration model. Let us define the configuration model using Meta Object Facility (MOF, Object Management Group (2011)) first. Its definition is shown in Fig. 6. The class ConfigurationModel represents a configuration of our approach. It contains a reference to Class specifying the possible classes whose elements can be roots. And the property uniqueRoot specifies if there should be only one root element in the generated model. A ConfigurationModel can own four kinds of constraints, i.e., ElementRangeConstraints, RelationshipConstraints, GlobalRangeConstraints, and ExtraConstraints.

ElementRangeConstraint restricts the amount of elements owning the same type. The element type is specified by the relationship class from ElementRangeConstraint to Class (defined in MOF). Each ElementRangeConstraint may also contain some AttributeRangeConstraints each of which denotes a value range constraint. The attribute to be constrained is specified by the relationship attribute. The class RelationshipConstraint, which combines the semantic constraint and relationship range constraints, is used to guide generating relationships. Note that containment references cannot be imposed any constraints explicitly because they are constrained in a default manner.

A GlobalRangeConstraint is used to specify default range constraints, including the total size, the default element range, the default relationship range, and the default value range (for different data types, e.g., integer and string). Total size specifies the total number of elements in the generated model. Default element range specifies the number of elements for any kind. If there is no existence of an ElementRangeConstraint for a certain class, the default range will be used. Default relationship range and default value range are similar to that case.

ExtraConstraints represent the constraints not discussed in this paper. They must be written as OCL invariants. Our approach could not solve those constraints during model generation, which will be used to validate the generated model.

ElementRangeConstraint, AttributeRangeConstraint, ReferenceConstraint, and GlobalRangeConstraint are subclasses of Bound. Formally, a Bound can be defined as a set \( \{(v_i, p_i)\} \), where \( \sum_i p_i = 1 \land p_i \geq 0 \). Each pair \((v_i, p_i)\) in the Bound can be interpreted as the probability of selecting the value \(v_i\) is \(p_i\), where \(p\) is termed selection probability. To select a value from a Bound, generate a random non-

![Fig. 6. Configuration model.](image-url)
negative number $r$ first, and then return the value $v_j$ that satisfies $\Sigma_j \leq \beta_j < r < \Sigma_{j+1} \leq \beta_{j+1}$.

In the configuration model, a **Bound** is specified in the textual form. Its grammar can be defined as follows:

$$
\text{bound ::= blist} \\
\text{item ::= bitem | pbitem} \\
\text{pbitem ::= bitem : probability} \\
\text{bitem ::= blist | bRange | literal} \\
\text{blist ::= \{ item | (item + \} } \\
\text{bRange ::= \{ literal .. literal \} } \\
\text{literal ::= any constant for a data type} \\
\text{probability ::= a real number from 0 to 1}
$$

where `blist` represents a list of possible values and `bRange` represents a value range defined by a minimum and a maximum value. All the values in the same `bRange` share the same selection possibility. `bitem` represents an item (a literal, a list, or a range) whose possibility is not defined explicitly (its possibility is deduced from other items).

The **Bound** specified in this form can be converted into the set of pairs based on which the value selection process has been explained above. For example, \{(3,0.5), (5,0.2), (7..9):0.3\} can be translated into \{(3,0.5), (5,0.2), (7,0.1), (8,0.1), (9,0.1)\}

Please see Algorithm 11 in Appendix for more information.

### 4.2. Element and attribute generation

The second phase is to generate model elements and attributes. In short, for each non-abstract class $t$ in the metamodel $M$, create $s_t$ elements. Then, for each attribute $a$ of $t$ and an element $e$, randomly assign a value to $a(e)$, where $s_t$ denotes the number of $t$-elements in the produced model. The basic element generation algorithm is described in Algorithm 1.

**Algorithm 1: GenerateElements(T)**

**Input:** $T$, the set of classes

**Output:** $E$, the set of elements

1. **foreach** $t \in T$ **do**
2. $s_t \leftarrow$ the number of elements to be generated;
3. **for** $i \leftarrow 0$; $i < s_t$; $i \leftarrow i + 1$ **do**
4. $e \leftarrow$ a new instance of $t$;
5. **foreach** $a : t \rightarrow t_y$ **do**
6. range$\alpha$ $\leftarrow$ the value range constraint for $a$;
7. $val$ $\leftarrow$ a value randomly selected from range$\alpha$;
8. $a(e) \leftarrow$ val;
9. $E \leftarrow E \cup \{e\}$
10. **return** $E$;

The algorithm is not difficult. However, the difficulty is to determine $s_t$ for each class $t$. If $s_t$ is not assigned properly, the model may not be correctly generated. It is because that the metamodel and the configuration model impose some constraints as follows:

First, let

$$S_t \equiv \sum_{o \in t} s_o$$

Second, for any class $t$ whose root elements could not be root elements, $S_t$ is determined by the number of all possible containers, i.e.,

$$\sum_{r \in E(t)} (lt \times S_{r:\text{source}}) \leq S_t \leq \sum_{r \in E(t)} (ut \times S_{r:\text{source}})$$

where $L_r(t) = \{r \in C \land t.target \in \Gamma_r\}$. $U_r(t) = \{r \in C \land t.target \in \Gamma_r \cup \Gamma_{\Gamma_r}\}$, and for each $r \in L_r(t) \cup U_r(t)$, $(\langle ls, u_s, \langle lt, ut, r \rangle \rangle = \text{mult}(r))$.

Third, for two classes $s$, $t$, $S_s$ and $S_t$ are also constrained by the multiplicity of the non-containment reference $r$, where $(s,t) = \text{assoc}(r)$ and $(\langle ls, u_s, \langle lt, ut, r \rangle \rangle = \text{mult}(r)$, i.e.,

$$ut \times S_t \geq ls \times S_t \land us \times S_t \geq lt \times S_t$$

Fourth, for each class $t$, $S_t$ must be a value in range, where range denotes the bound specified by the corresponding ElementRangeConstraint or the default element range if the ElementRangeConstraint is missing. Note that this requirement is flexible because our approach may break it to ensure the correctness.

At last, if total size is defined as a GlobalRangeConstraint, the sum of all $S_t$ must meet this constraint.

To generate a valid model randomly, we must solve those range constraints (i.e., determining the value of $S_t$ for each class $t$). We can employ the constraint solver to find a random solution for a set of constraints. According to the solution, we can determine $S_t$ for each class $t$.

### 4.3. Relationship generation

The third phase is to generate the relationships of the model $M$, which is described in Algorithm 2. It has three sub-phases, i.e., generating containment relationships (line 2), constrained relationships (line 5), and normal relationships (line 7), based on the semantics and the constraints defined in the configuration model.

**Algorithm 2: EntryOfGenerateRelationships(R, C)**

**Input:** $R$, the set of references; $C$, the set of containment references

1. $L \leftarrow \emptyset$
2. GenerateContainmentRelationships($C$);
3. **foreach** $r$ in referenceConstraint $\in$ the configuration model do
4. $L \leftarrow L \cup$ cons references;
5. GenerateContainedRelationships($cons$);
6. **foreach** $r \in \Gamma \cup L$ do
7. GenerateOtherRelationships($r$);
8. **return** $L$;

First, let us consider the containment references. All containment references are non-reflexive, ordered, source-unique, and source-necessary (except for the roots), so for all containment references we can handle them as Algorithm 3. parents are all possi-

**Algorithm 3: GenerateContainmentRelationships(S)**

**Input:** $S$, the containment reference set

1. $E \leftarrow$ the set of elements;
2. $RT \leftarrow$ the root classes specified in the configuration model;
3. $parents \leftarrow E$;
4. $children \leftarrow E$;
5. possibleRoots $\leftarrow$ \{ $e \in E \land \exists t (t \in RT \land \text{type}_e (e) \leq t)$ \};
6. **if** the model could have only one root element **then**
7. $\text{numberOfRoot} \leftarrow 1$;
8. **else**
9. $\text{numberOfRoot} \leftarrow$ an integer from 1 to $|E| - 1$;
10. randomly remove $\text{numberOfRoot}$ elements included in possibleRoots from children;
11. $size \leftarrow |E| - \text{numberOfRoot}$;
12. GenerateRelationships($S$, parents, children, true, true, false, false, true, size);
ble container elements; *children* are all possible children elements. They are initialized by *E*, the element set of *M* (line 1 to 4). In line 5, we collect the possible root elements. From line 6 to 9, we calculate the number of root elements (i.e., *numberOfRoots*) based on the value of *uniqueRoot* specified in the configuration model. In line 10, we select *numberOfRoots* elements from *possibleRoots* and remove them from *children* (a root does not require a parent). The actual generation logic is realized by calling *GenerateRelationships* (in line 12), according to the semantic constraints of containment references. We will discuss *GenerateRelationships* later.

Second, we generate some relationships based on the *RelationshipConstraints* defined in the configuration model. Since a *RelationshipConstraint* specifies the semantic constraints and the range constraint on a set of references, we simply extract the necessary information and use it as the actual parameters of *GenerateReferences*. This process can be described in Algorithm 4. For each *RelationshipConstraint*, we collect the possible source and target elements (line 3 and 4), and randomly select the number of relationships to be generated (line 5). Then, similar to Algorithm 3, we call *GenerateReferences* and employ the information provided by the constraint as the parameter (line 6).

**Algorithm 4: GenerateConstrainedRelationships**

*Input:* cons, a *RelationshipConstraint*

1. *E* ← the set of elements;
2. *refs* ← the references related to cons;
3. *src* ← \[e ∈ E \land ∃r ∈ refs \land typePe(e) ≤ t.source\];
4. *tar* ← \[e ∈ E \land ∃r ∈ refs \land typePe(e) ≤ t.target\];
5. *size* ← a value selected from the range constraint of cons;
6. *GenerateRelationships*(refs, src, tar, cons.sourceUnique, cons.targetUnique, cons.targetNecessary, cons.reflexive, cons.uniqueSize);

In the last sub-phase, we handle the remainder references with default relationship constraints (i.e., reflexive and unordered). This procedure is described in Algorithm 5.

**Algorithm 5: GenerateOtherRelationships**

*Input:* *r*, a reference

*Output:* the produced relationships

1. *E* ← the set of elements;
2. *src* ← \[e ∈ E \land typePe(e) ≤ t.source\];
3. *tar* ← \[e ∈ E \land typePe(e) ≤ t.target\];
4. *size* ← a value selected from the range constraint for *r*;
5. *GenerateRelationships*(*r*, src, tar, false, false, false, true, false, size);

As mentioned above, the core generation logic is realized by the function *GenerateRelationships*, which has been called in Algorithms 3, 4, and 5. The basic idea of *GenerateRelationships* is as follows: 1) randomly select a reference *r*; 2) randomly pick a source element *e*<sb>₁</sb> and a target element *e*<sb>₂</sb> from the element set *N* constructed by Algorithm 1; 3) establish a *r*-relationship \{(e<sb>₁</sb>, e<sb>₂</sb>)\} in *M*. However, during this process, the following two problems must be tackled: 1) how to select *e*<sb>₁</sb> and *e*<sb>₂</sb> properly in order to satisfy all the constraints on relationships? 2) how to do if there is no valid *e*<sb>₁</sb> or *e*<sb>₂</sb>?

Before going on, we have to define two auxiliary functions *IsForbidden* and *SelectCandidate*. *IsForbidden* checks if two elements *e*<sb>₁</sb> and *e*<sb>₂</sb> can be connected with a *r*-relationship without violating reflexivity and ordering constraints (as described in Algorithm 6). *IsCandidate* is used to check if an element *e* can be the (source or target) end of a *r*-relationship without breaking the upper bound, as well as the necessity and the uniqueness constraints.

*IsCandidate* has two versions, i.e., *IsSrcCandidate* and *IsTarCandidate*. Algorithm 7 presents *IsSrcCandidate*. It is responsible for checking if an element *e* can be the source end or not. If *e* has fulfilled the lower bound and there exists another element *e<sb>′</sb>* not satisfying the lower bound, *e* could not be a valid source (line 4, 6, and 7). It is our algorithm that gives the elements that do not satisfy the lower bound top priority to be used to create relationships. That is intended for ensuring all the elements reach their lower bounds. *IsTarCandidate*, which selects the target end, is similar to this algorithm. However, it uses *dom<sub>≤</sub>(e)*, *lsr*, and *usr* to replace *ran<sub>≤</sub>(e)*, *lt<sb>r</sb>* and *ut<sb>r</sb>* respectively.

**Algorithm 6: IsForbidden**

*Input:* *S*, is a reference set; *e<sb>₁</sb>* and *e<sb>₂</sb>* the candidate source and target element; reflexive and ordered, semantic constraints on *S*

*Output:* whether the two semantic constraints will be violated after (*e<sb>₁</sb>, *e<sb>₂</sb>*) is created

1. if reflexive = false and *e<sb>₁</sb>* = *e<sb>₂</sb>* then return true;
2. else if ordered = true and *e<sb>₁</sb>* < *e<sb>₂</sb>* then return true;
3. else return false;

**Algorithm 7: IsSrcCandidate**

*Input:* *r*, a reference; *e*, the element to be checked; necessary and unique, semantic constraints focused by this algorithm

*Output:* whether the element that can be the source end of a new *r*-relationship

1. *S*<sb>rs</sb> = the reference set containing *r*;
2. (*lsr*, *lt<sb>r</sb>*), (*usr*, *ut<sb>r</sb>*) = *mult*(r);
3. *U* ← \{candidate elements, each of which does not violate the upper bound and the semantic constraints, and has not been marked as an invalid candidate\};
4. *C<sb>l</sb>* ← \{o ∈ *U* ∧ (∃e<sub>₁</sub> ∈ *S*<sub>rs</sub> ∧ |ran<sub>≤</sub>(o)| < *lt<sb>r</sb>* ∧ (∃e<sub>₂</sub> ∈ *S*<sub>rs</sub> ∧ |ran<sub>≥</sub>(o)| < *ut<sb>r</sb>*))\};
5. *C<sb>u</sb>* ← \{o ∈ *U* ∧ (∃e<sub>₁</sub> ∈ *S*<sub>rs</sub> ∧ |ran<sub>≤</sub>(o)| < *lt<sb>r</sb>* ∧ (∃e<sub>₂</sub> ∈ *S*<sub>rs</sub> ∧ |ran<sub>≥</sub>(o)| < *ut<sb>r</sb>*))\};
6. if *C<sb>l</sb>* ≠ ∅ then return whether *C<sb>l</sb>* contains *e*;
7. else if *C<sb>u</sb>* ≠ ∅ then return whether *C<sb>u</sb>* contains *e*;
8. else return false;

Now, let us consider the details of *GenerateRelationships*. It can be described briefly as Algorithm 8. It will not stop producing relationships until the termination condition is satisfied. When producing a new relationship, it firstly tries to find two elements that can compose a valid *r*-relationship (line 5 and 6). During this process, it employs Algorithms 6 and 7 to select proper elements. For example, as shown in Fig. 7(a), it is a simple metamodel containing two classes *A* and *B*, and a containment reference *r* from *A* to *B*. The metamodel prescribes that each *A* element must be associated with at least one *B* element. As shown in Fig. 7(b), it is an intermediate result including two *A* and two *B* elements. A *r*-relationship \{(e<sb>₁</sb>, e<sb>₂</sb>)\} has been produced, and now we try to generate the second one. According to our algorithm (line 5), only *e<sb>₃</sb>* and *e<sb>₄</sb>* are the valid candidates. Otherwise, if we chose *e<sb>₁</sb>* and *e<sb>₄</sb>* and established a relationship, we would not get a valid result because no other
Algorithm 8: GenerateRelationships(S, src, tar, srcUnique, srcNecessary, tarUnique, tarNecessary, reflexive, ordered, size)

Input: S, a reference set; src, the set of source elements; tar, the set of target elements; size, the number of relationships to be generated; srcUnique, srcNecessary, tarUnique, tarNecessary, reflexive, and ordered, the semantic constraints on S

1. L ← the set of relationships of model M;
2. repeat
3. if S = ∅ then return L;
4. r ← a reference selected from S randomly, which satisfies \( \forall e \in (\text{src}, \text{tar}) \land |\text{src}(e)| \leq |\text{tar}(e)| \Rightarrow \exists e' \in \text{tar}(e), e < e' \)
5. \( e_1, e_2 \) ← two elements selected from src and tar, where isSrcCandidate(e_1, srcUnique, srcNecessary) \& isTarCandidate(e_2, tarUnique, tarNecessary) \& ~isForbidden(S, e_1, e_2, reflexive, ordered);
6. if such \( e_1 \) and \( e_2 \) can be found then \( L \leftarrow L \cup \{e_1, e_2\} \);
7. else if only \( e_1 \) exists then
8. fix the model with \( (e_1, r) \);
9. else if only \( e_2 \) exists then
10. fix the model with \( (r, e_2) \);
11. else \( S \leftarrow S - \{r\}; \)
12 until all the elements in src and tar have satisfied their lower bounds and (all the elements have reached their upper bounds or size relationships have been produced);

\[ B \] element can be associated with \( e_2 \). Such an element selection strategy is intended for fulfilling the lower bound requirement.

If the condition in line 5 is partially satisfied, i.e., there does not exist any \( e_1 \) or \( e_2 \) satisfying the condition (line 7 and 9), we try to fix the model by removing an existing relationship so that two new relationships can be appended without breaking bound and semantic constraints. The fixing procedure, which is intended for adjusting the model to append a new relationship, will be discussed in detail in the next subsection. If the fixing job fails, mark \( e_1 \) or \( e_2 \) as an invalid candidate of \( r \).

If the condition in line 5 is totally unsatisfied, remove \( r \) from \( S \) so that it would not be chose again.

The algorithm will terminate when: 1) \( G \) is empty; 2) all elements have reached to their upper bounds (i.e. no more relationship can be produced) or size relationships have been produced. And this algorithm does terminate, because during each iteration a new relationship will be produced, an element is removed from the candidate set, or a reference is removed. This means the termination conditions will eventually be satisfied.

4.4. Model fixing

As shown in Algorithm 8, it is possible that only \( e_1 \) or \( e_2 \) (but not both at the same time) is found. Note that such a phenomenon does not always create a problem. However, if it is caused by an improperly created relationship and hinders the subsequent generation job, it must be fixed (line 8 and 10 in Algorithm 8). There are two base structures that, though not necessarily, might result in this problem.

Before going on, we define a new concept named co-evolved references.

Definition 3. Co-evolution A set \( G \) of references is source co-evolved, if and only if there exists a constant \( c_1 \) let \( \forall e \in (\text{src}) < c_1 \) hold; \( G \) is target co-evolved, if and only if there exists a constant \( c_2 \) let \( \forall e \in (\text{tar}) < c_2 \) hold.

Obviously, source-uniqueness and target-uniqueness are special cases of co-evolution.

The first base structure. Fig. 8(a) presents the first exceptional base structure: there are four classes (A, B, C, and D) and three refers (r1, r2, and r3). When r2 and r3 are source co-evolved, and r1 and r3 are target co-evolved, a problem may take place when creating a r2-relationship for a D element or when creating a r1-relationship for a B element.

For example, as shown in Fig. 8(b), a problem arose when creating a r2-relationship starting from e4, provided that r1, r2, and r3 are source-unique, and \( \forall e \in (\text{ran}(r_4)) \leq 1 \). In this case, e3 is required by the attempt to creating a r2-relationship from e4, however it has already been connected to e2 by a r3-relationship. Since e3 can only be connected to either e2 or e3 (source uniqueness), a conflict occurs.

It is obvious that the r3-relationship is the cause of this problem. To handle this conflict, a possible solution for this example, as shown in Fig. 8(c), is as follows: 1) remove this r3-relationship to release e3; then 2) create a r2-relationship (e4, e3) and a r1-relationship (e1, e2). The reason why we must create a r1-relationship is to ensure that the fixing procedure always increases the number of relationships. Such a fixing procedure is listed in Algorithm 9.

As mentioned above, for the structure in Fig. 8(a), a problem may also happen when creating a r1-relationship for a B element. It is because that all the A elements may have been consumed by r3-relationships (r1 and r3 are target co-evolved). The solution to this case is similar to the former as shown in Algorithm 12 in Appendix. Actually, it can be regarded as a dual of Algorithm 9 by reverting the direction of all the references in Fig. 8(a).

The second base structure. The second exceptional base structure is related to ordering. As shown in Fig. 9(a), there are three class (A, B, and C) and three references (r1, r2, and r3). If r1, r2, and r3 are ordered, source unique, and source necessary (actually, they
are containment references), a problem may occur when creating a r1-relationship for an A element or creating a r2-relationship for a C element.

Fig. 9(b) shows an example. If we want to create a r1-relationship from e1 to e2, there would be a conflict because e2 has been consumed by a r3-relationship from e3. This problem can also be interpreted from another point of view: assuming that we are going to create a r2-relationship for e3, there is a conflict because all B elements have become the succeeding nodes of e3.

However, the problem is interpreted, the possible solution, as shown in Fig. 9(c), remains the same. In the figure, (e1, e2) is removed, and (e1, e2) and (e2, e3) are created. The fixing procedure is listed in Algorithm 10. As for the case of creating a r2-relationship for e3, the fixing procedure, as shown in Algorithm 13 in Appendix, is similar to the logic mentioned above.

**Structure conversion.** The structures presented above are two basic cases, while a metamodel may contain more complex fragments that cannot be enumerated thoroughly. Fortunately, it is possible to convert complex cases into the two basic cases by applying a set of conversion operators. The problem that occurs in the original fragment will also happen after appropriate conversion, and the solutions that are designed for the basic cases are also applicable to the complex cases. The rest of this section proposes some conversion operators and demonstrates how to solve conflicts by converting a metamodel into basic cases using those operators.

**Operator 1 (Inheritance removal).** Assume that there are two classes c1 and c2, where c1 ⊈ c2. After applying this operator: 1) the inheritance between c1 and c2 is removed; 2) ∀r1 = (c, c2) is split into r1 = (c, c2) and r′ = (c, c1); 3) ∀r0 = (c2, c) is split into r0 = (c2, o) and r′ = (c, o). This operator is used to eliminate inferences.

When a reference r is split into r and r′, its semantic constraints are preserved. Besides, two constraints, i.e., ∀v(\(\text{ran}(r, r') \leq ut_r\)) and ∀\(\text{dom}(r, r') \leq ut_s\)), are appended, provided that \(\text{mult}(r) = (\langle L_r, ut_r \rangle, (L_s, ut_s)\)). If there has already been a constraint
4.5. Constraint validation

The final step of our approach is to validate if the model generated fulfils all the constraints, including syntactic constraints, semantic constraints, and user-defined constraints. Violating any constraint, the model will be rejected, and an error message will be reported to users. User-defined constraints (especially complex OCL constraints) are most likely to be violated. However, since our approach mainly contributes to performance testing, such complex OCL constraints are not frequently required. Besides, if there is a conflict among relationship range constraints, our approach may also break them in order to generate a syntactically correct model. In most cases, our approach is able to produce a valid model.

5. Evaluation

This section presents two case studies, one experiment in randomness, two experiments in the time costs of our algorithm. The two performance experiments in Sections 5.3 and 5.4 were carried out on a computer with Intel i7 4770 CPU, 16GB RAM, and Windows 7.

5.1. Case studies

In this subsection, we present two case studies to demonstrate the feasibility and effectiveness of our test framework. In the first case study, we used our framework to test the performance of an ATL transformation, while in the second we evaluated the performance of EMF Compare.

Performance of ATL Transformation. Van Amstel et al. (2011) described a method for testing performance of model transformations. This case study adopted their methodology to test an ATL transformation with the support of our performance testing framework and our model generation algorithm. The transformation to be tested is an ATL transformation named JavaScript2Table\(^6\), an open-source transformation from ATL Zoo. We also created a modified transformation (JavaScript2Table(M)) from the original one by replacing all the helpers with inline expressions as a contrast in this study. Table 1 shows the metric values of the two transformations.

---

5 The prototype implementation, based on Eclipse, can be found at https://bitbucket.org/ustbmde/model-generation.git.

6 http://www.eclipse.org/atl/atlTransformations/#JavaScript2Table (Sep. 20, 2015).
Table 2  Performance testing results of JavaSource2Table and JavaSource2Table(M).

<table>
<thead>
<tr>
<th>Test Suite</th>
<th>JavaSource2Table</th>
<th>JavaSource2Table(M)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Suite 1 (200)</td>
<td>0.057</td>
<td>0.175</td>
</tr>
<tr>
<td>Suite 2 (400)</td>
<td>0.213</td>
<td>1.204</td>
</tr>
<tr>
<td>Suite 3 (600)</td>
<td>0.458</td>
<td>3.962</td>
</tr>
<tr>
<td>Suite 4 (800)</td>
<td>1.144</td>
<td>9.244</td>
</tr>
<tr>
<td>Suite 5 (1000)</td>
<td>1.979</td>
<td>17.791</td>
</tr>
<tr>
<td>Suite 6 (1200)</td>
<td>3.526</td>
<td>31.640</td>
</tr>
<tr>
<td>Suite 7 (1400)</td>
<td>6.689</td>
<td>53.248</td>
</tr>
<tr>
<td>Suite 8 (1600)</td>
<td>9.032</td>
<td>81.093</td>
</tr>
<tr>
<td>Suite 9 (1800)</td>
<td>13.763</td>
<td>115.644</td>
</tr>
<tr>
<td>Suite 10 (2000)</td>
<td>19.467</td>
<td>158.799</td>
</tr>
</tbody>
</table>

To test the performance of the two transformations, a TestProject was created in our testing framework, and a launch configuration for the ATL transformation was established in Eclipse. Then, the TestProject was associated with the launch configuration via a LCBasedSpecification. After that, we created a RandomDataGenerator and a ConfigurationModel for test data generation.

A TestSuite element and 5 test cases were created in the testing framework. We appended the 5 test cases into the TestSuite. For each test case, our framework invoked the test data generator implementing the algorithm proposed in Section 4. According to the ConfigurationModel specified before to generate an input model. The first 5 models had 200 elements each. Then, we established the second TestSuite and another 5 test cases for it. In the meantime, the ConfigurationModel was modified to produce 5 new models containing 400 elements each. This procedure was repeated until we created 10 TestSuites in total.

We executed all the 10 TestSuites for the original transformation. For each TestSuite, we collected all the execution times of the five test cases and calculated the average time cost. Then, we changed the executable in the launch configuration to the modified transformation, and ran all the ten TestSuites again.

The result is presented in Table 2. The integers in brackets in the first column indicate the numbers of elements in the input model. For the original transformation, it spent 0.057 seconds in average in producing an output in suite 1, and 19.467 seconds in average in suite 10. For the modified transformation, it spent 0.175 seconds in average to produce an output in suite 1, and 158.799 seconds in average in suite 10.

It is evident that the modified transformation ran much slower than the original one when the model size increased. The only difference between the two transformations is that the modified transformation uses inline expressions instead of ATL helpers (Table 1). Since ATL will cache the results of helpers, the original transformation is more efficient. This result is also consistent with the results in Van Amstel et al. (2011).

We spent half a day finishing this example, including establishing the configuration models, generating test inputs, executing the transformation, and calculating the results. According to our experience, it would probably take us a couple of days to construct valid test inputs, if we do not use our approach and tool. This study also shows that our framework can support the performance testing method proposed by other researchers. Our framework facilitates performance testing of model transformations.

Performance of EMF Compare. Now assume that we want to evaluate the performance of EMF Compare, the state-of-art model comparison tool. EMF Compare is able to compare the differences between two models conforming to the same metamodel.

We used three different metamodels, JavaSource, BibTex, and extlibrary to test the comparison efficiency. Both JavaSource and BibTex were extracted from ATL Zoo, while extlibrary was extracted from a standard example of EMF.

For each metamodel, we constructed a TestProject. For each TestProject, we created five TestSuites, each of which included 20 TestCases. For eachTestCase, we generated 2 models to be compared by EMF Compare.

We collected the time costs of executing those TestCases and calculated the average time costs for each TestSuite (in seconds). The results are shown in Table 3. The integers in brackets in the leftmost column indicate the numbers of elements the input models of the same suite have. From the table, we can conclude that the time complexity of EMF Compare is approximately \( O(N^2) \), where \( N \) is the number of elements.

This case demonstrated that not only model transformations but also other model-related operations, such as model comparison can be handled by our framework.

Table 3  Performance testing results of EMF Compare.

<table>
<thead>
<tr>
<th></th>
<th>JavaSource</th>
<th>BibTex</th>
<th>Exlibrary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Suite 1 (500)</td>
<td>1.138</td>
<td>2.381</td>
<td>3.224</td>
</tr>
<tr>
<td>Suite 2 (1000)</td>
<td>4.371</td>
<td>8.694</td>
<td>12.664</td>
</tr>
<tr>
<td>Suite 3 (1500)</td>
<td>9.953</td>
<td>22.350</td>
<td>28.822</td>
</tr>
<tr>
<td>Suite 5 (2500)</td>
<td>29.309</td>
<td>57.520</td>
<td>76.049</td>
</tr>
</tbody>
</table>

5.2. Experiment in randomness

Our model generation algorithm is a randomized approach. This means all the produced models are randomly constructed. The property of randomness is meaningful for estimating the average performance of a model-related operation. Hence, we conducted an experiment to evaluate the randomness of our approach. And we want to know if it is possible to make the similarity of any pair of produced models smaller than 5%.

We selected three metamodels, i.e., JavaSource, BibTex, and extlibrary, which were already used previously. For each metamodel, five sets of models were generated. For each set, 20 models were generated. All the models belonging to the same model set were generated according to the identical configuration. However, models in different sets have different sizes. Then, for each pair of models in the same set, EMF Compare was used to find matches. The similarity of two models is defined by

\[
\text{#match/#total}
\]

where \text{#match} is the number of the matches returned by EMF Compare and \text{#total} is the total number of elements in the model. At last, we computed the average similarity for each set, and result is listed in Table 4.

From the table, we can learn that it is possible to control the similarity of two generated models under 5% in our approach. According to our experience, to achieve this goal, the size of every possible attribute-value range had better be ten times larger than the element amount.

Table 4  Result of randomness experiment.

<table>
<thead>
<tr>
<th></th>
<th>#1</th>
<th>#2</th>
<th>#3</th>
<th>#4</th>
<th>#5</th>
</tr>
</thead>
<tbody>
<tr>
<td>JavaSource</td>
<td>0.85%</td>
<td>0.81%</td>
<td>0.73%</td>
<td>0.86%</td>
<td>0.74%</td>
</tr>
<tr>
<td>PetriNet</td>
<td>1.32%</td>
<td>1.97%</td>
<td>0.62%</td>
<td>0.72%</td>
<td>0.51%</td>
</tr>
<tr>
<td>extlibrary</td>
<td>0.0014%</td>
<td>0.0024%</td>
<td>0.0059%</td>
<td>0.0037%</td>
<td>0.0050%</td>
</tr>
</tbody>
</table>
5.3. Experiment in performance: Comparative studies

In this experiment, we try to answer the question how faster can our approach generate a model than the solver-based approaches? Note that we did not compare our approach with other algorithm-based approaches. Their limitations have been discussed sufficiently in Wu et al. (2012). And we will discuss them in Section 7.

First, a test was conducted to compare the efficiencies of EMFto-CSP Pérez et al. (2012) and Alloy Jackson et al. (2000) with our approach’s. EMFtoCSP and Alloy, the state-of-the-art model constraint satisfaction solvers, are also used to generate test inputs for model transformations Sen et al. (2008); González et al., 2012). All the three approaches are forced to generate models conforming to the JavaSource metamodel (Fig. 12 in Appendix). And there is no extra constraint. Note that for Alloy we equivalently translated the metamodel to the Alloy specification before the experiment.

Then, every approach generated 10 models. And, each model was repeatedly generated 5 times to obtain the average time cost. Each model has a particular amount of elements: the ith model has $5 \times i$ ClassDeclarations, $20 \times i$ MethodDefinitions, $125 \times i$ Method-Invocations, and one JavaSource (the root element), i.e., $150 \times i$ elements except for the root.

The result is listed in Table 5. Each cell indicates the average time cost (in seconds) for an approach (row) to generate a model (column). During this experiment, the time costs of our approach ranged from 0.06s to 0.23s; the time costs of Alloy ranged from 0.12s to 45.04s. With regard to EMFtoCSP, it failed to produce any models (except for the first one) within a reasonable time (30 minutes).

A further test was carried out to compare the efficiencies of Alloy and our approach by using the PetriNet metamodel (since our approach is apparently better than EMFtoCSP in performance). Similar to the former experiment, the two approaches were asked to generate 10 models, whose sizes increased linearly. Each model was generated 10 times by each approach to obtain the average time costs (in seconds). The result is presented in Table 6. From the table, the time cost of our approach ranged from 0.06s to 1.76s; Alloy spent 222.87s producing the first PetriNet model (containing 600 elements) but failed to produce any larger one.

According to this experiment, it is evidently that our model generation approach is significantly more efficient than the other two. And, we believe that our approach also has performance advantage compared to other solver-based approaches.

However, we do not claim that our model generation algorithm can take the place of the constraint solver, e.g., Alloy. The qualitative analytic result is listed in Table 7.

Alloy supports first-order-logic-based constraints, while our approach can only handle some predefined model constraints defined in Section 3.2. These predefined constraints can be encoded in Alloy codes. It means they are a subset of Alloy constraints.

<table>
<thead>
<tr>
<th>Table 5</th>
<th>JavaSource model generation performance of Alloy, EMFtoCSP, and ours.</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>#2</td>
</tr>
<tr>
<td>Ours</td>
<td>0.06</td>
</tr>
<tr>
<td>Alloy</td>
<td>0.12</td>
</tr>
<tr>
<td>EMFCC</td>
<td>72.79</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 6</th>
<th>PetriNet model generation performance of Alloy and ours.</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>#2</td>
</tr>
<tr>
<td>Ours</td>
<td>0.06</td>
</tr>
<tr>
<td>Alloy</td>
<td>222.87</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 7</th>
<th>Qualitative comparison between Alloy and our approach.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alloy</td>
<td>Our approach</td>
</tr>
<tr>
<td>Constraint</td>
<td>First-order-logic-based</td>
</tr>
<tr>
<td>Efficiency</td>
<td>Very low</td>
</tr>
<tr>
<td>Suitable testing method</td>
<td>White-box</td>
</tr>
<tr>
<td>Suitable testing goal</td>
<td>Correctness</td>
</tr>
<tr>
<td>Correctness of produced model</td>
<td>Yes</td>
</tr>
<tr>
<td>Randomness of produced model</td>
<td>Difficult and inefficient</td>
</tr>
</tbody>
</table>

Alloy is more expressive and can handle more kinds of constraints. It is suitable to white-box testing which verifies the correctness of an MRO. However, it cannot produce a large model efficiently. It is infeasible to produce a model containing thousands of elements, which is used to test the performance of an MRO with Alloy. As illustrated by this experiment, our algorithm produces a model significantly faster than Alloy does. Our algorithm is more suitable to performance testing of MROs.

If we do not consider other constraints, both Alloy and our approach can produce correct models. Besides, our approach can also produce random models (as tested in Section 5.2). Whereas, in Alloy, it is difficult and inefficient to obtain random models due to its searching strategy.

5.4. Experiment in performance: Algorithm complexity

In this experiment, we try to answer the question what law does the performance of our algorithm follow?

We employed our approach to generate a set of models according to 6 metamodels (without extra constraints). For each metamodel, our approach was required to produce five models, whose sizes ranged from 1 MB to 5 MB (approximately). And, every model was produced 5 times to obtain an average time cost (in seconds). The six metamodels are JavaSource, expbrary, BibTex, PetriNet, TextualPathExp, and MySQL. The first three have been used previously; PetriNet, TextualPathExp, and MySQL were also extracted from the transformations in ATL Zoo. To control the size of the model, the element and relationship range constraints of each model are fixed integers. And, all the unnecessary spaces in the output XML files were eliminated.

The result of this experiment is shown in Table 8. Obviously, for different metamodels, the time costs vary a lot. The performance curves for the six metamodels are depicted as in Fig. 11. For JavaSource, the performance curve follows $y = 8.2474x^2 + 1.841x + 33.44$; for PetriNet, the curve follows $y = 11.16x^3 + 0.961$; for expbrary, their curve follows $y = 2.7509x^2 + 0.5955x + 116.8$; for BibTex, the curve follows $y = 13.415x^2 + 160.8$; for TextualPathExp, the curve follows $y = 41.332x^2 + 360$; for MySQL, the curve follows $y = 0.0276x^2 + 132$.

\footnote{In our previous work, we did not remove the spaces. Hence, we redid the experiment for the first three, i.e., the results are different from the old ones.}
we new count. attributes, discuss conflict constraints, produce 6.

Discussion

From our approach, the value of the exponent \( p \) ranged from 1.9961 to 2.36, and its average value amounted to 2.14878 approximately. The exponent did not vary significantly.

Table 8

<table>
<thead>
<tr>
<th></th>
<th>JavaS</th>
<th>PetriN</th>
<th>Extlib</th>
<th>BibTex</th>
<th>TextualPE</th>
<th>MySQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 MB</td>
<td>8.6</td>
<td>11.1</td>
<td>2.8</td>
<td>13.6</td>
<td>43.1</td>
<td>6.27</td>
</tr>
<tr>
<td>2 MB</td>
<td>35.1</td>
<td>46.5</td>
<td>10.9</td>
<td>63.4</td>
<td>204.6</td>
<td>25.35</td>
</tr>
<tr>
<td>3 MB</td>
<td>93.3</td>
<td>93.7</td>
<td>27.2</td>
<td>122.3</td>
<td>531.5</td>
<td>58.97</td>
</tr>
<tr>
<td>4 MB</td>
<td>155.4</td>
<td>169.2</td>
<td>48.5</td>
<td>281.6</td>
<td>1021.9</td>
<td>118.9</td>
</tr>
<tr>
<td>5 MB</td>
<td>303.4</td>
<td>293.3</td>
<td>74.9</td>
<td>434.9</td>
<td>2032.7</td>
<td>193.41</td>
</tr>
</tbody>
</table>

JavaS = JavaSource; PetriN = PetriNet; Extlib = Extlibrary; TextualPE = TextualPathExp

From the result, we can learn the following things:

1. When the model size increased linearly, the times cost followed a power law \( y = k \times x^p \).
2. The exponent \( p \) ranged from 1.9961 to 2.36, and its average value amounted to 2.14878 approximately. The exponent did not vary significantly.
3. For different metamodels, the coefficient \( k \) changed drastically. This implies that it is influenced by the complexity of the metamodel.

6. Discussion

Algorithm correctness. Our model generation algorithm can produce a model conforming to the syntactic constraints, semantic constraints, and user-defined range constraints, when there is no conflict. It will be complicated to prove this rigorously. We just discuss this qualitatively. First, our approach generates elements, attributes, and relationships based on the metamodel. Hence, the model produced satisfies the syntactic constraints (multiplicity constraints are considered below). Second, when producing relationships, our approach also takes semantic constraints into account. In line 5 of Algorithm 8, the elements selected to create a new relationship must satisfy the four semantic constraints. And it is not difficult to prove that the fixing procedures in Algorithms 9, 12, 10, and 13 will not violate the any semantic constraints, since we always check them before creating any changes. If the range constraints do not contain any conflict, Algorithms 1 and 8 will not stop until all of them are fulfilled.

Threats to validity. There are two major threats to the validity of our approach. For one thing, some advanced features in the metamodel may be an obstacle to applying our model generation algorithm. As defined in Section 3.1, only basic features, such as classes, attributes, and unidirectional references, in a metamodel are able to be handled in this paper. Advanced features, including derived attributes, union and support of references, multidirectional references, and package import, are not supported. When a metamodel includes one of such advanced features, it cannot be instantiated by our algorithm. Although some complex metamodels, such as UML metamodel, may have those features in most cases, we can bypass those features by replacing them with basic features, according to our experiences. For another thing, some model-related operations may have special preconditions, which may be too complex to be written in OCL on input models. For example, a model transformation can only handle structural workflow models, while the constraint that the model must be structural cannot be easily achieved by a random model generator. This means that our framework based on random model generation cannot deal with the operations having such special input preconditions. It will be our future work to investigate this problem in detail.

7. Related work

7.1 Model transformation testing

Baudry et al. (2010), discussed the barriers to testing model transformation, and test data generation is one of the challenges. This paper attempts to fill this gap by proposing a testing framework of model transformations and a random model generation algorithm.

Küster et al. (2009), proposed incremental development of model transformation chains based on automated testing. They defined four test design techniques and a test framework for transformation chain. Their test framework, based on JUnit,
supporting model comparison and invariant validation, is comparable to our framework. Both of the frameworks support automated testing of model transformations. However, instead of functional testing, our framework mainly focuses on performance testing. It also integrates with an automated test data generation algorithm, which is not addressed in their work.

Giner and Pelechano (2009), proposed a template of test case specifications to capture requirements for transformations, and to guide the development and documentation of model transformations. The test model proposed in this paper shares some commonalities with Giner’s template, in spite of that they employ different concrete syntax to specify a test case. Our test model can also serve as a test case management mechanism, which enables us to arrange test cases into test suites.

Shelburg et al. (2011), presented an approach to regression testing for model transformations. They argued that a transformation that is changed owing to the evolution of the metamodel requires regression testing. To do so, they proposed a multi-objective optimization algorithm to generate test inputs that maximizes the coverage of the new metamodel by refactoring the existing one. We believe that our approach can cooperate with their approach. Our approach can be used to create and manage test cases, while their approach can refactor the test cases to meet the requirements of regression testing.

Guerra proposed a specification-driven test generation approach Guerra and Soeken (2013). In this approach, one must specify the transformation properties (invariants, pre- and post-conditions) using a declarative specification language named PassMo. Then, those properties are solved by a proper constraint solver to derive the test input and the oracle function. Finot et al., discussed the oracle problem in transformation testing and developed a technology called partial test oracle that employs a set of model fragments (partial oracle) to determine the correctness of the test output Finot et al. (2013). Those efforts, which promoted transformation testing, targeted at white-box and functional testing. However, the test framework and the test generation algorithm proposed in this paper concentrated on performance testing of model transformations.

Van Amstel and Van Den Brand (2011), proposed three complementary techniques for the analysis of model transformations, including metrics collections, structure and dependency analysis, and metamodel coverage visualization. Their paper mainly focuses on static analysis of model transformations in order to improve maintainability. Our work discusses performance testing of model-related operations, including but not limited to model transformations.

Van Amstel et al. (2011) also reported some experimental results about the performance of QVT and ATL transformations. They compared the performance with different languages and engines, different implementation styles, and different inputs. Their paper described the methodology of performance testing in MDE. Their results can be used as a guide to improving the performance of transformations. Our work differs from theirs. This paper concentrates on the performance testing framework and the test input generation algorithm, which are ignored in their paper. However, we believe that the two papers complement one another. The framework and the model generation algorithm proposed in this paper can support the testing method described in their paper, as we have demonstrated in Section 5.1.

7.2. Model generation

Model generation (or metamodel instance generation) broadly with two technical paradigms: solver-based and algorithm-based. The basic idea of solver-based approaches is as follows: 1) translate the metamodel and constraints into a formal specification that can be accepted by a model checker, a constraint solver, or a SAT/SMT solver; 2) employ those solvers to solve the specification and obtain a valid instance; 3) translate the instance found by solvers back to a model. 

Alloy Jackson et al. (2000) is a mature model checker for relational logic. Since a model can be regarded as a set of elements and relationships, research on model generation with Alloy has been very active. Anastasakis et al. (2010) discussed how to translate a UML class diagram into Alloy. They proposed some rules to map the UML class diagram and the limited OCL constraints onto Alloy Language. Sen et al. (2008) proposed a tool called Cartier to generate test models for testing model transformations. McQuillan and Power (2008) proposed a metamodel for the measurement of object-oriented systems. Users can define some metrics for an object-oriented system using this metamodel, and then the metrics are translated into Alloy to generate a valid model.

ECU/PS®, proposed by Apt and Wallace (2007), is a model constraint solver. Cabot et al. (2008) proposed an approach to translate a UML model with OCL constraints into ECU/PS®. Their approach is supported by the tool EMFtoCSP Pérez et al. (2012). And, it has been applied to test model generation González et al., (2012).

Soukén et al. (2010, 2011) also proposed an approach to verify a UML model with OCL constraints using a SAT solver. They employed bit-vector theory to encode the metamodel and the constraints.

It is not difficult to find that solver-based approaches are more flexible because they can deal with more kinds of declarative and semantic constraints. They are suitable for model verification and white-box testing. However, they cannot generate large models efficiently compared to our algorithm-based approaches. Hence, they are less suitable to handle performance testing of model-related operations.

Algorithm-based paradigm. Mougnot et al. (2009) proposed a unique generator for huge models based on Boltzmann method Duchon et al. (2004). They translated a metamodel into a tree specification, where classes were nodes and containment references were edges. During the translation, multiplicities of containment references are also considered. Then, they employed Boltzmann method to generate a valid tree conforming to the specification. Their approach is quick efficient, however, they did not discuss how to generate non-containment references. Semantic constraints are also neglected in their approach. So their approach might produce an invalid model, e.g., a class diagram containing an inheritance circle.

Brottier et al. (2006) also proposed an algorithm of model generation for testing model transformations. Their algorithm builds a model according to certain coverage criteria by combining a set of model fragments. However, they did not discuss how to create those fragments.

Ehrig et al. (2009) introduced a graph-grammar-based model generation approach. In their approach, metamodells and constraints are encoded as a set of graph transformation rules. By executing those rules, a model can be generated. Even though graph-grammar based approach is also declarative syntactically, we do not adopt it because the most frequently-performed operation during graph transformation is graph pattern matching, which has been proven to be an NP-complete problem in theory and is the performance bottleneck to generate large model efficiently. On the contrary, ours is more configurable and takes more constraints (such as semantic constraints) into account.
Preliminary study. Our preliminary study on the random model generation algorithm has been reported in our earlier paper He et al. (2014) published in COMPSAC’14. In the earlier paper, the basic ideas of the generation algorithm, as well as some experiment results, were presented. This paper, as an enhancement of our earlier, proposed a performance testing framework for model-related operations. The framework, based on a test model, provides the ability of specifying, managing, and performing test cases. The model generation algorithm is also integrated into the framework to produce test data. This paper also refined the model generation algorithm and discussed the procedure of model fixing in detail. At last, more case studies and experiments were presented in Section 5 compared to the earlier paper.

8. Conclusion and future work

Since the scalability of the model-related operations gained lots of attentions, testing its performance has become a vital task in the model-driven engineering. The paper contributes to this field in the following three aspects:

1. An extensible performance testing framework and a test model, which enable us to specify and organize test cases, and to carry out automated performance test;
2. A model generation algorithm, which can help us generate random inputs, used for eliminate the average time cost efficiently and correctly;
3. Two case studies, one experiment in randomness, and two experiments in generation efficiency, which demonstrated the feasibility of our testing framework, and evaluated the randomness and the efficiency of our model generation algorithm.

In the future, we will try to handle more kinds of constraints (especially more semantic constraints) in model generation. Then, we will study if our model generation algorithm can produce models with restrictions, e.g., the process model that must be structural, as mentioned in Section 6, and will investigate to what extent our approach can overcome this limitation. Third, we will try to improve the usability of our prototype tool. Fourth, we will carry out more case studies and experiments to quantify how much our approach is better than others. At last, we plan to investigate whether the framework and the generation algorithm are suitable for black-box testing of model-related operations or not, owning to the white-box testing technology for some operations may not be available at present.

Acknowledgements

This work was supported by the National Program on Key Basic Research Project of China (973 Program) (Grant No. 2013CB329601), the National High Technology Research and Development Program of China (863 Program) (Grant No. 2012AA011022), the National Natural Science Foundation of China (Grant Nos. 61300009, 61272159), China Postdoctoral Science Foundation funded project (Grant No. 2013MS40050), Fundamental Research Funds for the Central Universities (Grant No. FRF-TP-14-040A2).

The authors are also grateful to Ms. Tingxian Qiu from China and Prof. Xuedong Shi from Beijing Information Science and Technology University for their efforts of polishing this paper.

Appendix

Algorithm 11 shows how to do the translation. Given a Bound bound, Flatten(bound, 1) returns the logic form.

\[
\text{Flatten}(item, dPos) = \begin{cases} \text{literal}(\{\text{item}, \text{dPos}\}); & \text{if item is a literal} \\ \text{default}(\{\text{item}\}); & \text{if item is a default} \\ \text{Flatten}(\text{item} \rightarrow \text{dPos}); & \text{if item is a relationship} \\ \text{Flatten}(\text{item} \cdot \text{dPos}); & \text{if item is a composition} \\ \text{Flatten}(\text{item} \bigtriangleup \text{dPos}); & \text{if item is a difference} \\ \text{Flatten}(\text{item} \bigtriangleup \text{dPos}); & \text{if item is a symmetric difference} \\ \text{Flatten}(\text{item} \bigvee \text{dPos}); & \text{if item is a union} \\ \text{Flatten}(\text{item} \bigwedge \text{dPos}); & \text{if item is a intersection} \\ \text{Flatten}(\text{item} \sim \text{dPos}); & \text{if item is a symmetric difference} \\ \end{cases}
\]

Algorithm 12: FixingModelForCase1′(r, e0)

Input: e0, a candidate target element; r, a reference to be initialized
1. S ← the set of references being focused on;
2. E ← the set of elements in the model;
3. L ← the set of relationship in the model;
4. cons ← the relationship constraint on S;
5. foreach (e0, e1) in the model do
6. if ¬IsForbidden(S, e0, e1, cons.reflexive, cons.ordered), and
7. IsSrcCandidate(r, e0, cons.tarUnique, cons.tarNecessary),
8. then (e0, e1) is removed then
9. foreach e0 ∈ E and r′ ∈ S do
10. if ¬IsForbidden(S, e0, e1, cons.reflexive, cons.ordered) and
11. IsSrcCandidate(r′, e0, cons.tarUnique, cons.tarNecessary) and
12. IsTarCandidate(r′, e0, cons.srcUnique, cons.srcNecessary),
13. then (e0, e1) is a r-relationship and (e0, e1) a r’-relationship;
14. mark e1 as an invalid target of r;
Algorithm 13: FixingModelForCase2(\(r, e_r\))

Input: \(e_r\), a candidate target element; \(r\), a reference to be initialized

1. \(E \leftarrow \) the set of elements in the model;
2. \(L \leftarrow \) the set of relationship in the model;
3. \(S \leftarrow \) the set of references being focused on;
4. \(\text{cons} \leftarrow \) the relationship constraint on \(S\);

foreach \(e_t \in E\) do

if there exist \(e_t, e_s, e_g \in E\) and \(r' \in S\) that \(e_t \rightarrow s e_s, (e_s, e_r) \in L\), and \(e_r \leftarrow s e_t\) hold then

if \(\text{IsSrcCandidate}(r, e_t, \text{cons.tarUnique}, \text{cons.tarNecessary})\) and
\(\text{IsTarCandidate}(r', e_t, \text{cons.srcUnique}, \text{cons.srcNecessary})\) and
\(\text{IsSrcCandidate}(r', e_t, \text{cons.srcUnique}, \text{cons.srcNecessary})\) and
\(\neg\text{IsForbidden}(S, e_t, \text{cons.reflective}, \text{cons.ordered}),\) provided that \((e_s, e_t)\) was removed then

if the type constraints are satisfied then

\[ L \leftarrow L - (e_s, e_r) + \{(e_s, e_t, e_r)\}\] where \((e_s, e_r)\) is a \(r'\)-relationship and \((e_s, e_t)\) a \(r\)-relationship;

return;

mark \(e_t\) as an invalid source of \(r\);

References


Dr. Xiao is an assistant professor at School of Computer and Communication Engineering, University of Science and Technology Beijing. He is now a visiting professor in Johann Bernoulli Institute for Mathematics and Computer Science, Faculty of Mathematics and Natural Sciences, University of Groningen. He gained his doctorate in computer science from Peking University in 2012. His main research interests include model transformation languages, model generation, metamodeling and domain-specific modeling.

Dr. Tian Zhang is an associate professor at the Computer Science and Technology Department of Nanjing University, China. At the same time, he is also a research fellow at the State Key Laboratory for Novel Software Technology. He obtained the permanent position in Nanjing University after graduation. His overall research interests relate to model driven aspects of software engineering, with the aim of facilitating the rapid and reliable development and maintenance of both large and small software systems. Currently, he is in charge of two NNSF projects of China and a Jiangsu Province Research Foundation.

Dr. Chang-Jun Hu is a full professor at school of Computer and Communication Engineering, University of Science and Technology Beijing. He gained his doctorate in computer science from Peking University in 2001. He obtained the permanent position in University of Science and Technology Beijing in 2004. His main research interests include domain-specific software engineering, parallel computing, and data engineering.

Dr. Zhiyi Ma is an associate professor at Department of Computer Science and Technology, Peking University. He received the Ph.D. degree in computer science in 1999. His research interests focus on metamodeling, software modeling method, and model transformation. He is the author of eleven books and more than 100 academic papers.

Prof. Weizhong Shao is a full professor at Peking University. His main research interests include object-oriented technology and UML, operation systems, component-based systems.