Testing Model Transformation Programs using Metamorphic Testing *

Mingyue Jiang\textsuperscript{1, 3}, Tsong Yueh Chen\textsuperscript{1}, Fei-Ching Kuo\textsuperscript{1}, Zhi Quan Zhou\textsuperscript{2}, Zuohua Ding\textsuperscript{3} \\
\textsuperscript{1}Department of Computer Science and Software Engineering \\
Swinburne University of Technology, Hawthorn, VIC 3122, Australia \\
\textsuperscript{2}School of Computer Science and Software Engineering \\
University of Wollongong, Wollongong, NSW 2522, Australia \\
\textsuperscript{3}Laboratory of Scientific Computing and Software Engineering \\
Zhejiang Sci-Tech University, Hangzhou, Zhejiang, 310018, China

Abstract

Model transformations are crucial for the success of Model Driven Engineering. Testing is a prevailing technique of verifying the correctness of model transformation programs. A major challenge in model transformation testing is the oracle problem, which refers to the difficulty or high cost in determining the correctness of the output models. Metamorphic Testing alleviates the oracle problem by making use of the relationships among the inputs and outputs of multiple executions of the target function. This paper investigates the effectiveness and feasibility of metamorphic testing in testing model transformation programs. Empirical results show that metamorphic testing is an effective testing method for model transformation programs.

Keywords: Metamorphic Testing, Model Transformation, Software Quality, Software Testing, Test Oracle

1. Introduction

Model transformation, which refers to the automatic process of transforming one model into another, is a vital element of Model Driven Engineering (MDE). In MDE, model transformations are usually used to transform models between different languages or different abstraction levels. In this way, models are automatically transformed and refined until code of final software is produced. The success of MDE critically depends on the correctness of model transformation programs as an incorrect transformation will result in incorrect models and the final software.

Testing is a prevailing technique of verifying the correctness of model transformation programs. A major challenge in the testing process is the oracle problem: In general, it is difficult to obtain test oracles for model transformation programs \cite{8}. We propose the technique of Metamorphic Testing (MT) to alleviate the oracle problem in testing model transformation programs. MT has been successfully applied to detect real-world faults \cite{3, 5}. In MT, programs are tested against their expectedly necessary properties. A major difference between MT and all the other testing methods for model transformation is that the properties used by MT are relationships among the inputs and outputs of multiple executions of the target program (known as metamorphic relations), whereas the properties used by the other methods focus on the input and output of a single execution. Another difference is that when testing model transformations, metamorphic relations (MRs) can be extracted from informal specifications, whereas most of the other methods rely on formal specifications.

2. Model Transformation

Model transformation is a critical activity in MDE, which is about the generation of target models from source models. A framework of model transformation is given in Fig. 1. The source metamodel (M Ma) and the target metamodel (M Mb) describe the static information of models, which are manipulated by the model transformation. The source (M a) and target (M b) models conform to their respective metamodels. The transformation model (M t) refers to an implementation (program) of the model transformation, and M Mt is the metamodel of Mt. The model transformation program (Mt) takes a source model as input and produces a target model as output.

There are different transformation languages, of which a popular one is the ATLAS Transformation Language (ATL) \cite{6}. We conducted a case study using a popular model transformation program written in ATL, namely,
Class2Relational, which is an “advanced example” open-sourced in the ATL Transformations Zoo¹ and is often used as a subject program by various experimentations [4].

In Class2Relational, Class model is the source model and Relational model is the target model. The Class and Relational models conform to the Class and Relational metamodels, respectively (see Fig. 1). In a Class model, each DataType represents a primitive data type, and each Class has a name and a set of Attributes, each of which can be single-valued or multi-valued and has either DataType or Class as its type. In a Relational model, each Table contains a name, a reference to its key Columns and a set of Columns, each of which is described by its name and type. The following describes the requirements of how Class2Relational should transform a Class model into a Relational model:

(1) For each DataType, a Type is created.
(2) For each Class, a Table (Type1) is created. Their names are identical. The Table contains a key Column, whose name is “objectId” and type is a specific type (In this example, it refers to Integer). Each Attribute of the Class is also manipulated, which is described in the following.
(3) For each single-valued Attribute of type DataType, a Column is created, and their names and types are identical.
(4) For each multi-valued Attribute of type DataType, a Table (Type2) is created. Two Columns of the Table are also created. One is the identifier Column (with a specific type) and the other contains name and type of the Attribute.
(5) For each single-valued Attribute of type Class, a Column is created. The name of the Column is the Attribute’s name +“id”, and the type of the Column is a specific type.
(6) For each multi-valued Attribute of type Class, a new Table (Type3) is created, which has two Columns with specific types (one is the identifier Column, and the other is named attribute.name + “id”).

(7) The name of the Table (Type2, Type3) is set to str1+ “_”+str2, str1 represents the name of the Class which contains the Attribute, and str2 represents the name of the Attribute. The Table’s identifier Column is named str14+“id”.

The model transformation program class2relational.atl was written according to the above requirements. An example Class model is given in Table 1 (left column), written in the XML Metadata Interchange (XMI) format. After executing class2relational.atl with this Class model as input, the output model, that is, the corresponding Relational model, is shown in Table 1 (right column). Obviously, it is not difficult to manually verify the correctness of the transformation. It should be noted that real-world models are much larger and much more complex than the above example. Checking the correctness of the transformations of real-world models is therefore a very difficult task.

3. Metamorphic Testing

Metamorphic Testing (MT) [3] is a methodology designed to alleviate the oracle problem. Different from conventional testing strategies, MT uses some specific properties known as Metamorphic Relations (MRs) involving multiple test cases and their outputs.

Let $p$ be a program implementing function $f$. To test $p$, suppose a set of test cases $T=\{t_1, t_2, \ldots, t_n\}$ ($n > 0$) have been generated using some test case selection strategies (such as black-box, white-box or random testing). Test cases in $T$ are referred to as original test cases. Based on the knowledge of $f$, some MRs can be identified. For each MR, a set of follow-up test cases can be generated for $T$. Suppose $t_i'$ is a follow-up test case for the original test case $t_i$, then $(t_i, t_i')$ is called a metamorphic test group [12]. MT runs the original and follow-up test cases and checks whether the outputs satisfy the MRs, regardless of the availability of an oracle for each individual test case.

4. Application of Metamorphic Testing to Model Transformation

The procedure is outlined as follows: First, identify MRs and construct a set of original test models. For each MR, generate follow-up test models based on the original test models. Then execute the model transformation program using both the original and follow-up test models, and

---

¹http://www.eclipse.org/atl/atlTransformations
Tables of MRs can be identified:

<table>
<thead>
<tr>
<th>MR1.1</th>
<th>Reset of values of some attributes of the test model.</th>
</tr>
</thead>
<tbody>
<tr>
<td>MR1.2</td>
<td>Suppose Attr is an attribute of Class Cla in C1, and C2 is constructed by reversing the value of Attr.multivalued.</td>
</tr>
<tr>
<td>MR1.3</td>
<td>Suppose Attr is an attribute of Class Cla in C1, and C2 is constructed by changing Attr’s type.</td>
</tr>
</tbody>
</table>

### 2. Insertion of an element into the test model.

**MR2.1:** Construct C2 by adding a DataType into C1, then $T_2.#Type\_Columns = \{T_1.#Type+1\}$, and $T_2.#Tables = T_1.#Tables$.

**MR2.2:** Construct C2 by adding a Class into C1, then $T_2.#Type\_Tables = \{T_1.#Type\_Tables+1\}$, $T_2.#Columns = T_1.#Columns$, $T_2.#Types = T_1.#Types$, and $T_2.#specific\_Columns = T_1.#specific\_Columns$.

**MR2.3:** Construct C2 by adding an Attribute to C1.

#### MR2.3.1: The added Attribute is a single-valued Attribute of DataType, then $T_2.#Columns = \{T_1.#Columns+1\}$, $T_2.#Tables = T_1.#Tables$, $T_2.#Types = T_1.#Types$, and $T_2.#specific\_Columns = T_1.#specific\_Columns$.

#### MR2.3.2: The added Attribute is a multi-valued Attribute of DataType, then $T_2.#Columns = \{T_1.#Columns+2\}$, $T_2.#Tables = \{T_1.#Tables+1\}$, $T_2.#Types = T_1.#Types$, $T_2.#Type\_Tables = T_1.#Type\_Tables$, $T_2.#Type\_Columns = T_1.#Type\_Columns$, and $T_2.#specific\_Columns = T_1.#specific\_Columns+1$.

#### MR2.3.3: The added Attribute is a single-valued Attribute of Class, then $T_2.#Columns = \{T_1.#Columns+1\}$, $T_2.#Tables = T_1.#Tables$, $T_2.#Types = T_1.#Types$, and $T_2.#specific\_Columns = T_1.#specific\_Columns+1$.

#### MR2.3.4: The added Attribute is a multi-valued Attribute of Class, then $T_2.#Columns = \{T_1.#Columns+2\}$, $T_2.#Tables = \{T_1.#Tables+1\}$, $T_2.#Types = T_1.#Types$, $T_2.#Type\_Tables = T_1.#Type\_Tables$, $T_2.#Type\_Columns = T_1.#Type\_Columns$, and $T_2.#specific\_Columns = T_1.#specific\_Columns+1$.

---

### 1. Reset of values of some attributes of the test model.

**MR1.1:** If we modify the values of some attributes of C1 to obtain C2 (the modified values are legal), then $T_2.#Type\_Tables = T_1.#Type\_Tables$, and $T_2.#Types = T_1.#Types$.

**MR1.2:** Suppose Attr is an attribute of Class Cla in C1, and C2 is constructed by reversing the value of Attr.multivalued.

- If Attr.multivalued is true (It is false in C2), then we have:
  
  $\left( T_2.Tag.Columns \setminus T_1.Tag.Columns \right) = \{ Col \mid \text{Col.name contains Attr.name}\}$, where Tag is the Table whose name equals Cla.name and \ is the set difference operator, which will be used hereafter in this paper.

  $T_2.#T_{s1} = (T_1.#T_{s1} + 1)$, where $T_{s1}$ is a set composed of Tables whose name contains Attr.name.

  and $T_2.#T_{s2} = (T_1.#T_{s2} + 1)$, where $T_{s2}$ is a set composed of Tables whose name contains Cla.name.

- If Attr.multivalued is false, then $T_1.Tag.Columns \setminus T_2.Tag.Columns = \{ Col \mid \text{Col.name contains Attr.name}\}$,

  $T_2.#T_{s1} = (T_1.#T_{s1} + 1)$, and $T_2.#T_{s2} = (T_1.#T_{s2} + 1)$.}

---

### 2. Insertion of an element into the test model.

**MR2.1:** Construct C2 by adding a DataType into C1, then $T_2.#Type\_Columns = T_1.#Columns$, $T_2.#Types = T_1.#Types+1$, and $T_2.#Tables = T_1.#Tables$.

**MR2.2:** Construct C2 by adding a Class into C1, then $T_2.#Type\_Tables = T_1.#Type\_Tables+1$, $T_2.#Columns = T_1.#Columns$, $T_2.#Types = T_1.#Types$, and $T_2.#specific\_Columns = T_1.#specific\_Columns$.

**MR2.3:** Construct C2 by adding an Attribute to C1.

**MR2.3.1:** The added Attribute is a single-valued Attribute of DataType, then $T_2.#Columns = T_1.#Columns+1$, $T_2.#Tables = T_1.#Tables$, $T_2.#Types = T_1.#Types$, and $T_2.#specific\_Columns = T_1.#specific\_Columns$.

**MR2.3.2:** The added Attribute is a multi-valued Attribute of DataType, then $T_2.#Columns = T_1.#Columns+2$, $T_2.#Tables = T_1.#Tables+1$, $T_2.#Types = T_1.#Types$, $T_2.#Type\_Tables = T_1.#Type\_Tables$, $T_2.#Type\_Columns = T_1.#Type\_Columns$, and $T_2.#specific\_Columns = T_1.#specific\_Columns+1$.

**MR2.3.3:** The added Attribute is a single-valued Attribute of Class, then $T_2.#Columns = T_1.#Columns+1$, $T_2.#Tables = T_1.#Tables$, $T_2.#Types = T_1.#Types$, and $T_2.#specific\_Columns = T_1.#specific\_Columns+1$.

**MR2.3.4:** The added Attribute is a multi-valued Attribute of Class, then $T_2.#Columns = T_1.#Columns+2$, $T_2.#Tables = T_1.#Tables+1$, $T_2.#Types = T_1.#Types$, $T_2.#Type\_Tables = T_1.#Type\_Tables$, $T_2.#Type\_Columns = T_1.#Type\_Columns$, and $T_2.#specific\_Columns = T_1.#specific\_Columns+1$.
3. Deletion of data from the test model according to the output model

**MR3.1:** Suppose Col is a Column of Table Tab (Tab is a Table of Type1) in the output mode of C1. Construct C2 by deleting information related to Col of C1.
- If Col is related to a single-valued Attribute, then
  \[(T_1:Tab.Columns \setminus T_2:Tab.Columns) = \{\text{Col}\}\]
- If Col is related to a multi-valued Attribute, then
  \[(T_1:Tables \setminus T_2:Tables) = \{T \mid T.name = \text{Tab.name}+\,.\,.\,.+\text{Col.name}\}, \text{and} (T_1:Columns \setminus T_2:Columns) = \{Col \mid \text{Col.name either contains Tab.name or Col.name}\}\]

**MR3.2:** Suppose Tab is a Table of Type1 in the output model of C1, and C2 is constructed by deleting information related to Tab of C1. Then we have: \[(T_1:Tables \setminus T_2:Tables) = \{T \mid T.name = \text{Tab.name}+\,.\,.\,.+\text{str}, \text{where str can be empty}\}\]

4. Interchange of data in the test model

**MR4** Suppose Cla1 and Cla2 are two Classes of C1, and Attr1 and Attr2 are Attributes of Cla1 and Cla2, respectively. C2 is constructed by interchanging the data of Attr1 and Attr2 (that is, in C2, Attr1 becomes an Attribute of Cla2 and Attr2 becomes an Attribute of Cla1).
- If Attr1 and Attr2 are both single-valued Attributes, then
  \[T_2:Columns \setminus T_1:Columns, T_2:\#Tables = T_1:\#Tables,\]
  \[\text{DiffTable} = (T_2:Tables \setminus (T_2:Tables \setminus T_1:Tables)) = \{\text{Tab Tab.name = Cla1.name or Cla2.name}\}\]
- If one of these two attributes (namely, Attr1) is single-valued and the other (namely, Attr2) is multi-valued, then
  \[T_2:Columns = T_1:Columns, T_2:\#Tables = T_1:\#Tables,\]
  \[\text{DiffColumns} = (T_2:Columns \setminus (T_2:Columns \setminus T_1:Columns)) = \{\text{Col Col.name contains Cla1.name}\}\]
  \[\text{DiffTable} = (T_2:Table1:Tables \setminus (T_2:Table1:Tables \setminus T_1:Table1:Tables)) = \{\text{Tab Tab.name = Cla1.name or Tab1.name contains Attr1.name and DiffTable.size = 2}\}\]
  \[\text{DiffTable2} = (T_2:Table2:3:Tables \setminus (T_2:Table2:3:Tables \setminus T_1:Table2:3:Tables)) = \{\text{Tab Tab.name contains Attr2.name and Cla1.name and DiffTable2.size = 1}\}\]

5. Empirical Evaluation

5.1. Experimental procedure

We conducted empirical evaluation of MT using the model transformation program class2relational.atl, which has 107 lines of code and contains 6 ATL rules and 1 ATL helper. Using the MRs described in Section 4, the testing procedure consists of the following three steps:
1. **Generation of original test models.** The set of original test models were generated randomly in such a way that (i) they all conform to the source metamodel, (ii) all elements of the source metamodel are covered, and (iii) different original test models have different values in the same attributes in order to maximize diversity.
2. **Construction of follow-up test models.** Different MRs will result in different follow-up test models. These models were generated automatically.
3. **Verification of test results.** This step was also performed automatically by our test script against the MRs.

A total of 100 Class models were generated as the original test models for testing the subject program class2relational.atl. No violation of MRs was detected. This is expected as class2relational.atl is a popular and open-source program. In order to evaluate the fault-detection effectiveness of MT, we then applied *mutation analysis* [7] to generate 20 non-equivalent mutants from class2relational.atl. Details of the mutants are shown in Table 2, where Mi denotes the i\(^{th}\) mutant.

5.2. Results of experiments

We applied MT to test every mutant using the 100 original test models. Results of experiments are summarized in Table 3 in terms of the violation ratio which is defined as the ratio of violated metamorphic test groups among all used metamorphic test groups. The last row shows the average violation ratio for each individual MR, and the last column shows the average violation ratio for each individual mutant. It is observed that every mutant has some violated metamorphic test groups. In other words, all seeded faults are detected.

Table 3 shows that the average violation ratios of MRs range from 0.00 to 0.54. This result is consistent with many other MT studies, which reported that different MRs can have very different fault-detection effectiveness. Table 3 also shows that the fault-detection effectiveness of an MR is mutant dependent. Consider MR2.3.1, for instance, it has varied violation ratios for M3, M9, M19, M20, which are 1.00, 0.08, 0.00 and 1.00, respectively.

The effectiveness of MT can be further analyzed using metamorphic test groups. For each mutant, 100 \(\times 12 = 1,200\) metamorphic test groups have been executed. Therefore, there is a total of 1,200 \(\times 20 = 24,000\) metamorphic test groups. The total number of violated metamorphic test groups is 5,240, which gives the overall effectiveness of MT (in terms of violated metamorphic test groups) to be 5,240/24,000 = 22%. This result shows that MT is quite effective because a failure will be revealed after running about 5 metamorphic test groups on average.
5.3. A further analysis of the effectiveness of MRs

Table 3 shows that the fault-detection effectiveness of different MRs can be very different: MR4 was violated by every mutant, but MR1.1 was never violated. The most effective MR is MR4. Its average violation ratio is 0.54. That is, on average, more than half of its metamorphic test groups can reveal a failure. MR4 is highly effective because it makes use of more information of the transformation requirements than the remaining MRs. MR4 checks almost all data items of the Relational model, taking their concrete values into consideration, instead of just comparing the numbers of some elements. The other MRs (except the worst one, MR1.1) generate follow-up test models by adding or deleting some elements, or resetting the attributes’ values of some elements of the original test model. Their average violation ratios range from 0.10 to 0.47. They are less effective than MR4 because they check only certain parts of the Relational model. We analyzed each MR together with all the mutants that violated it. For any given pair of mutant and MR, a high violation ratio will be intuitively expected if the fault in the mutant is relevant to the MR.

MR1.1, which constructs follow-up test models by changing the values of some arbitrary attributes, was the least effective MR: It did not detect any violation. The rea-
son for this is twofold. First, in each and every metamorphic test group generated by MR1.1, the original and follow-up test case executions are almost identical in the sense that the same statements of the subject program are exercised (and in the same sequence). The original and follow-up output models generated in this way are therefore very similar. As a result, MR1.1 is very likely to be satisfied. This observation confirms the findings of Chen el al. [2] and Cao el al. [1]: an effective MR should make the original and follow-up test case executions as different as possible. Secondly, MR1.1 checks the test results at a quite high abstraction level by ignoring many details of the output models. Consequently, even if an output model is incorrect, the incorrect data item buried in the output model is not checked by MR1.1 and hence a violation cannot be detected. This finding shows that an effective MR should look at the details of the output as much as possible.

We have obtained two useful guidelines. First, MRs whose original and follow-up test case executions are very different, are likely to have a higher chance of detecting a failure than those whose original and follow-up test case executions are similar. Secondly, an effective MR should involve detailed information from the requirements specification as much as possible and as complete as possible.

6. Discussions and Conclusion

We propose to apply Metamorphic Testing (MT) to alleviate the oracle problem in testing model transformation programs. To evaluate the effectiveness of the proposed approach, a case study has been conducted using Class2Relational and mutation analysis. The empirical results show that MT can effectively detect model transformation faults. We used Metamorphic Relations (MRs) involving four kinds of operations, namely, addition of elements, deletion of elements, alteration of attribute’s values, and interchange of elements. We have obtained two guidelines for applying MT to model transformation programs. The first guideline is to select MRs whose original and follow-up test case executions are significantly different. The second guideline is to select MR that involves as many details of the model transformation as possible from the transformation requirements. These two guidelines are appropriate for the selection of MRs for any model transformation programs.

Many applications have a model transformation component or have been developed using model transformations. Examples of the former include software development tools that use model transformations to generate the application code [10]. Examples of the latter include context-aware pervasive systems [9] and secure XML data warehouses [11] which are developed using Model Driven Development (MDD) method. Obviously, the correctness of the model transformations will affect the quality of the final systems. MT can be used to test the model transformations in such applications.

References