An Approach for Debugging Model Transformations Applying Spectrum-Based Fault Localization

Javier Troya, Sergio Segura, José Antonio Parejo, and Antonio Ruiz-Cortés

Department of Computer Languages and Systems
Universidad de Sevilla, Spain
{jtroya, sergiosegura, japarejo, aruiz}@us.es

Abstract. Model transformations play a cornerstone role in Model-Driven Engineering as they provide the essential mechanisms for manipulating and transforming models. The use of assertions for checking their correctness has been proposed in several works. However, it is still challenging and error prone to locate the faulty rules, and the situation gets more critical as the size and complexity of model transformations grow, where manual debugging is no longer possible. Spectrum-Based Fault Localization (SBFL) is a technique for software debugging that uses the results of test cases and their corresponding code coverage information to estimate the likelihood of each program component (e.g., statements) of being faulty. This paper describes a proposal for applying SBFL for locating the faulty rules in ATL model transformations. The approach aims at automatically detecting the transformation rule that makes an assertion fail.

Keywords: Model Transformation, Debugging, Spectrum, Fault Localization

1 Introduction

Model transformations (MTs) are the cornerstone of MDE [2], as they provide the essential mechanisms for manipulating and transforming models. They are an excellent compromise between strong theoretical foundations and applicability to real-world problems, and most MT languages are composed of model transformation rules.

It is important in MDE to maintain and test MTs as it is done with source code in classical software engineering. However, checking whether the output of a MT is correct is a manual and error-prone task, known as the oracle problem. In order to alleviate the oracle problem, the formal specification of MTs has been proposed by the definition and use of *contracts* [5,1], i.e., assertions that the execution of the MTs must satisfy. These assertions can be specified on the models resulting from the MTs, the models serving as input for the MTs, or both, and they can be tested in a black-box manner.

However, even when using the assertions as oracle to test if MTs are correct, it is still challenging to debug them and locate what parts of the MTs are wrong. The situation gets more critical as the size and complexity of MTs grow, where manual debugging is no longer possible, so there is an increasing need to count on methods, mechanisms and tools for debugging them.

Spectrum-Based Fault Localization (SBFL) is a popular technique used in software debugging for the localization of bugs [6]. It uses the results of test cases and their corresponding code coverage information to estimate the likelihood of each program component (e.g., statements) of being faulty. A program spectrum details the execution information of a program from a certain perspective, such as branch or statement coverage. SBFL entails identifying the part of the program whose activity correlates most with the detection of errors.

This papers proposes to apply SBFL for debugging ATL model transformations. We follow the approaches in [5,1] and use the previously described contracts (assertions) as oracle function to determine the correctness of MTs, as it is explained in the next section. Then, in Section 3 we describe our next steps.

2 Approach

Let us use as running example the classical *Class2Relational* model transformation[1], which is composed of six transformation rules. Let us also consider we have the three OCL assertions shown in Listing 1.1 that are satisfied for the original *Class2Relational* model transformation. These OCL assertions check (i) if the number of instances of type *DataType* and *Type* is the same in the source and target models, respectively, (ii) if there is a *Column* in the target model for every *Attribute* of type *DataType* with its same name, and (iii) if there is a *Column* in the target model for every *Attribute* of type *Class* with its same name concatenated with 'Id'. Finally, let us consider a mutant of the model transformation where in the third rule, named *DataTypeAttribute2Column*, a fault has been introduced, causing the second OCL assertion to fail.

Listing 1.1: OCL assertions for the *Class2Relational* MT.

```
SrcDataType.allInstances()->size()=TrgType.allInstances()->size()
SrcAttribute.allInstances()->select(a|a.type.oclIsKindOf(SrcDataType))->forAll(
      a|TrgColumn.allInstances()->exists(c|c.name=a.name))
SrcAttribute.allInstances()->select(a|a.type.oclIsKindOf(SrcClass))->forAll(a|
      TrgColumn.allInstances()->exists(c|c.name=a.name+'Id'))
```

For applying SBFL with the aim of locating the faulty rule, a so-called *coverage matrix* and an *error vector* have to be created. For this, we need a set of *test cases* (input models), namely a test suite, that trigger the model transformation and produce a set of output models, so that we can check if the OCL assertions hold for the $< input, output >$ models pairs. Input modes should have a certain degree of variability among them so that different models exercise different transformation rules.

The coverage matrix for our running example is shown in the left-hand side of Table 1. Horizontally, the table shows the transformation rules in which we aim to locate bugs. Vertically, the table shows 10 test cases: automatically generated input models. A cell is marked with "●" if the transformation rule of the row has been exercised by the test case of the column. The information about the rules triggered by a given test case can be collected by inspecting the trace model, e.g., using Jouault's *TraceAdder* [4]. The final row depicts the error vector with the outcome of each test case, either successful ("S") or failed ("F"). In the example, the result of executing the MT with tc_1 and tc_{10} makes the assertion fail.

Once a coverage matrix and error vector are constructed, a number of techniques can be used to rank the transformation rules according to their *suspiciousness*, that is, their

[1] https://www.eclipse.org/atl/atlTransformations/Class2Relational/ExampleClass2Relational[v00.01].pdf

Table 1: Tarantula [3] suspiciousness values for the *Class2Relational* MT when the second OCL assertion fails

T. Rule	tc_1	tc_2	tc_3	tc_4	tc_5	tc_6	tc_7	tc_8	tc_9	tc_{10}	N_{CF}	N_{CS}	Suspiciousness	Rank
tr_1	•	•	•	•	•	•	•	•	•	•	2	8	0.5	5
tr_2	•	•	•	•	•	•	•	•	•	•	2	8	0.5	5
tr_3 (**BUG**)	•									•	2	0	1	1
tr_4	•										1	0	1	1
tr_5			•							•	1	1	0.8	4
tr_6	•		•							•	2	1	$0.\overline{8}$	3
Test Result	F	S	S	S	S	S	S	S	S	F				

probability of containing a fault. A popular fault localization technique is *Tarantula* [3], which for a program statement is computed as $(N_{CF}/N_F)/(N_{CF}/N_F + N_{CS}/N_S)$, where N_{CF} is the number of failing test cases that cover the statement, N_F is the total number of failing test cases, N_{CS} is the number of successful test cases that cover the statement, and N_S is the total number of successful test cases. The suspiciousness score of each statement is in the range [0,1], i.e., the higher the suspiciousness score of each component, the higher the probability of having a fault. The values of N_{CF}, N_{CS} and the *Tarantula* suspiciousness value for each statement are given in the twelfth, thirteenth and fourteenth columns of Table 1, respectively. The values for N_F and N_S are are 2 and 8, respectively. The last column in the table indicates the position of the statement in the suspiciousness-based ranking where top-ranked statements are more likely to be faulty. In the example, the faulty rule tr_3 is ranked first.

It is noteworthy that suspiciousness techniques may often provide the same value for different statements, being these tied for the same position in the ranking, e.g., rules tr_3 and tr_4 in Table 1. Under this scenario, different approaches are applicable such as measuring the effectiveness in the best and worst scenarios, using an additional technique to break the tie, or using some simple heuristics such as alphabetical ordering [6].

3 Highlights and Next Steps

An important aspect of this approach is that it can be fully automated, so that given a MT, a set of assertions and a set of source models, our approach indicates the violated assertions and uses the information of the MT coverage to rank the transformation rules according to their suspiciousness of containing a bug. Figure 1 exemplifies this process, where it can be seen that a rank for rules suspiciousness is generated for each non-satisfied OCL assertion. The user can pick any of these ranks in order to locate and fix the faulty rule. Then, the approach can be executed again with a partially fixed model transformation, so that less assertions are now likely to fail, until all OCL assertions are satisfied.

Another strong point of the approach is that it can be applied to other model transformation languages as long as they are composed of model transformation rules and the execution results can be stored in traces.

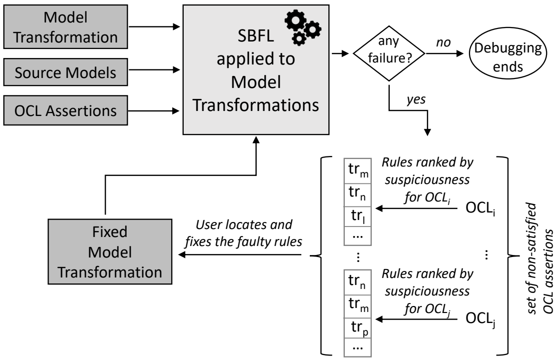
Model Transformation
Source Models
OCL Assertions
SBFL applied to Model Transformations
any failure?
no
Debugging ends
yes
Rules ranked by suspiciousness for OCL$_i$
OCL$_i$
Rules ranked by suspiciousness for OCL$_j$
OCL$_j$
tr$_m$
tr$_n$
tr$_l$
tr$_p$
set of non-satisfied OCL assertions
User locates and fixes the faulty rules
Fixed Model Transformation

Fig. 1: Debugging of a MT applying our SBFL approach

We plan to study in depth how useful SBFL is for locating faults in model transformations. Indeed, *Tarantula* is not the only technique, so we want to study different techniques and determine which ones are more effective in the context of model transformations, using for instance the EXAM score [7]. Also, we would like to apply this approach in several model transformations and to study the results when several faults are injected in more than one transformation rules. We also want to investigate how the test cases influence the results. For instance, we want to study the influence of a variation in the size of the test suite and in the variability of the input models composing the test suite. Finally, we aim at comparing our approach with related approaches [1].

Acknowledgement This work has been partially supported by the European Commission (FEDER) and Spanish Government under CICYT project BELI (TIN2015-70560-R), and the Andalusian Government project COPAS (P12-TIC-1867).

References

1. Burgueño, L., Troya, J., Wimmer, M., Vallecillo, A.: Static Fault Localization in Model Transformations. IEEE Tansactions on Software Engineering 41(5), 490–506 (May 2015)
2. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Systems Journal 45(3), 621–646 (2006)
3. Jones, J.A., Harrold, M.J.: Empirical Evaluation of the Tarantula Automatic Fault-localization Technique. In: Proc. of the 20th IEEE/ACM International Conference on Automated Software Engineering. pp. 273–282. ASE '05, ACM, New York, NY, USA (2005)
4. Jouault, F.: Loosely Coupled Traceability for ATL. In: Workshop Proc. of ECMDA (2005)
5. Vallecillo, A., Gogolla, M.: Typing model transformations using tracts. In: Proc. of 5th Int. Conf. on Theory and Practice of Mod. Trans. (ICMT 2012). pp. 56–71. Springer (2012)
6. Wong, W.E., Gao, R., Li, Y., Abreu, R., Wotawa, F.: A Survey on Software Fault Localization. IEEE Transactions on Software Engineering 42(8), 707–740 (2016)
7. Xie, X., Chen, T.Y., Kuo, F.C., Xu, B.: A Theoretical Analysis of the Risk Evaluation Formulas for Spectrum-based Fault Localization. ACM Tr. Softw. Eng. Meth. 22(4), 31:1–31:40 (2013)