Partial Test Oracle in Model Transformation Testing

Olivier Finot, Jean-Marie Mottu, Gerson Sunyé, J. Christian Attiogbe

Partial Test Oracle in Model Transformation Testing

Olivier Finot, Jean-Marie Mottu, Gerson Sunyé, and Christian Attiogbe

LINA CNRS UMR 6241 - University of Nantes
2, rue de la Houssinière, F-44322 Nantes Cedex, France
firstname.lastname@univ-nantes.fr

Abstract. Writing test oracles for model transformations is a difficult task. First, oracles must deal with models which are complex data. Second, the tester cannot always predict the expected value of all the properties of the output model produced by a transformation. In this paper, we propose an approach to create efficient oracles for validating part of the produced output model.
In this approach we presume that output models can be divided into two parts, a predictable part and a non-predictable one. After identifying the latter, we use it to create a filter. Before providing a (partial) verdict, the oracle compares actual output model with the expected output model, returning a difference model, and uses the filter to discard the differences related to the unpredictable part. The approach infers the unpredictable part from the model transformation specification, or from older output models, in the case of regression testing.
The approach is supported by a tool to build such partial oracles. We run several experiments writing partial oracles to validate output models returned by two model transformations. We validate our proposal comparing the effectiveness and complexity of partial oracles with oracles based on full model comparisons and contracts.

Keywords: Test, Partial Oracle, Model Comparison

1 Introduction

Model transformations are among the key elements of Model Driven Engineering. Since a transformation is often implemented to be reused several times, any implementation error impacts on all the produced models. Therefore, it is important to ensure that the implementation is correct w.r.t. its specification.

Software testing is a well-known technique for ensuring the correctness of an implementation. In the precise case of model transformations, a test consists of a transformation under test (TUT), a test input model and a test oracle. The role of the oracle is to ensure that the output model, produced by the TUT, is correct w.r.t. the transformation specification. Two methods are mainly used to implement the test oracle: (i) comparing the output with an *Expected Model* or (ii) using constraints to verify *Expected Properties* on the output model.

The tester is often able to predict the expected value for part of the output model only. For instance when part of the specification is very complex. In this case, the tester can only predict the part corresponding to what she knows about the specification. In other cases, the specification may allow several different, or *polymorphic*, outputs for the same input model. Therefore, the tester can predict the expected value of the output model's part that does not change from one variant to another. Finally if the TUT performs a model refactoring, then the tester can predict the part that should not be modified by the transformation. We are interested in using this predictable part with a partial test oracle to partially validate the correctness of output models.

Several approaches exist to write oracles for model transformation testing [1]. Full model comparison requires the expected model to be comprehensive. Contracts express constraints between any input and output models but considering few properties each. Assertions or patterns are suited to individually validate properties of one output model. However, we need to entirely validate the predictable part of the output model, not just a compilation of properties.

The contribution of this paper is an approach to implement test oracles issuing partial verdicts when part of the output model is predictable. The approach relies on filtered model comparison with partial expected models.

The partial expected model is compared with the output model generated by the TUT. The observed differences are filtered in order to reject those concerning the unpredictable part of the output model. To create the filter we need to precisely identify the unpredictable part. Elements are considered as belonging to the unpredictable part based the meta-elements they are instance of. Therefore, we propose to filter the comparison's result with a pattern extracted from the transformation's output meta-model.

Along with this approach, we propose a tool that automatically produces partial verdicts. The tool's inputs are an output model, a partial expected model, and patterns made of meta-model excerpts. We apply our approach to test two model transformations with polymorphic outputs. We create 94 test models; obtaining 94 partial verdicts with our partial oracles, detecting 4 bugs. We define 94 partial expected models with 2,632 elements, 70% less than for the classical model comparison approach.

This paper is organized as follows. Section 2 discusses the control of part of an output model by an oracle. Section 3 details our approach to define a partial oracle for part of the output model. We then present our implementation of the approach in Section 4. Section 5 presents the experiments we ran on two case studies and discuss the results. Section 6 discusses existing contributions on the topic of the verification of model transformations.

2 Test Oracle for Models Transformations

In this section, we discuss the situation where the tester is able to predict part of an output model but she does not have any oracle function suited to use it.

2.1 Test Oracle for Model Transformations

Figure 1 depicts the process of model transformation testing. The input and output data are models that conform to meta-models. The tester selects an input model, then the TUT transforms it, obtaining an *output model*. Finally, she writes test oracles aimed at validating the output model, ensuring that it is correct w.r.t. the TUT's specification.

A test oracle consists of two elements: *oracle function* and *oracle data* [1]. The oracle function analyses the output model and uses the oracle data to produce the verdict. For instance when comparing the actual result with the expected one, the oracle function is the comparison and the oracle data is the expected result. In previous work [1], we have defined several oracle functions to test model transformation. These oracle functions use *comparison* with an *expected model* or *constraints* expressed either between the input and output models or on the output one only.

Writing oracle data is a blackbox activity: the tester only relies on the specification not on the implementation. She should not be influenced by any fault made by the developer.

2.2 Partial verdict for Model transformation Testing

A test oracle may produce a partial verdict when only part of the specification is considered or part of the output data is validated. In model transformation testing, one may want to write partial oracles because the tester can only predict part of the expected output model in many situations. We distinguish three such situations:

1. The transformation's specification can be large and output models are complex data. The tester could predict only part for which she can handle the complexity.
2. The transformation can be endogenous, modifying partially the input model, e.g., model refactoring. Therefore, part of the input model remains unchanged and could be used as oracle data to check that the transformation is side-effect free.
3. The transformation can return polymorphic outputs: instead of an unique output solution, several variants of the expected model exist. Most of the time, those variants are semantically equivalent, but syntactically different.

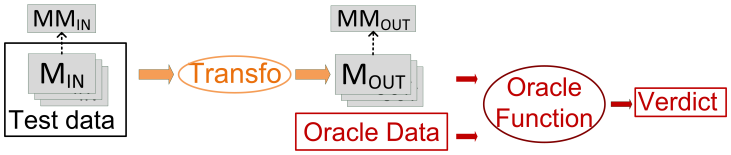

Fig. 1. Model Transformation Testing

This variability usually comes from the model transformation's specification. The tester cannot predict which variant should be produced by the transformation's implementation and she should consider them all.
The flattening of a state machine is an example of such a transformation. Its input is a hierarchical state machine, the output is another state machine expressing the same behavior without any composite state. The input model presented in Figure 2 can be transformed into the output model presented in Figure 3(a). With such state machines, the number of final states is not limited to only one. Thus, the model presented in Figure 3(b) is also a correct output for the transformation of the input presented in Figure 2.

In such situations, the tester is able to predict the expected value for part of the output model with limited effort, while the remaining part is unpredictable or too difficult to be predicted. We envisage being able to write a partial oracle, using predictable part of the expected model as oracle data.

2.3 Existing Oracle Functions and Partial Verdict

Considering *partial expected model* as oracle data requires a suitable oracle function. Several oracle functions have been proposed for model transformation testing [1], but are they suitable for partial verdict?

A first set of oracle functions considers *model comparison*. Testing frameworks implement such approach: (i) Lin et al. developed a testing engine [2] based on DSMDiff, a model comparison engine, (ii) EUnit [3] compares models with EMFCompare (compliant with the principles exposed by Cicchetti et al. [4]). They compare the output model with an expected model. The latter is obtained manually by the tester, or from a reference transformation (e.g., previous version or regression testing), or a reverse transformation returns an input model from the output model to be compared with the test model (this last approach is limited to injective transformations which are rare and require the existence of such a transformation, which cannot be developed only for testing, especially when the transformation returns polymorphic outputs).

Hence, using such comparison approaches is not suited to get partial verdict from part of an expected model. The comparison would identify differences concerning the unpredictable part of the output model: (1) when this part is too complex to be predicted by the tester, (2) when it concerns part transformed by a refactoring, (3) when it may have many variants with a polymorphic model.

Those differences should be manually analyzed to get the verdict of the test. We face this issue considering the three situations where we want to use partial

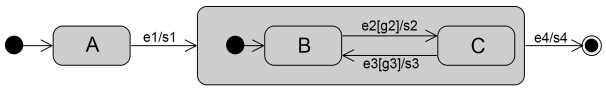

Fig. 2. Example M^{in}, of Hierarchical State Machine

expected models as oracle data. Such complete model comparison requires comprehensive expected models. Moreover, in case of polymorphic output models, it requires all the variants to be compared with the output model because only one of them could be equal. Therefore this oracle function is not effective to write partial oracle.

A second set of oracle functions considers *properties to be checked* on the output models. Contracts are constraints between input and output models. Cariou et al. [5], verify model transformations using contracts. Their contracts are composed of constraints (i) on the output model, (ii) on the evolutions of model elements from the input model to the output model. Defining contracts between input and output model is at least as complex as writing the transformation itself; thus they are as error prone as the transformation. Contracts are not suited to provide the partial verdict we envisage. They are only appropriate to control a few requirements of the specification: no output with composite states for instance. Vallecillo et al. [6] reach the same conclusion and present Tracts, partial contracts for this purpose.

A third set of oracle functions considers *assertions or patterns* (e.g., OCL constraints on the output model of one test case). It would allow controlling dedicated properties of one model. However a model is not just a big set containing many properties, the organization of these properties, i.e., their structure, is also important. Our goal is to globally control the predictable part and not a compilation of properties, thus assertions are not suited to our needs.

To sum up, existing oracle functions are not suited to our needs: controlling part of the output model using a partial expected model as oracle data. We seek for appropriate oracle function.

3 Filtered Model Comparison for a Partial Verdict

We propose a new approach to obtain a partial verdict for the test of model transformations. We define a partial oracle function considering part of the output model, the one the tester can predict.

The obtained verdict is only partial but it is still a good piece of information for the tester. Using this oracle function she is able to detect bugs in the trans-

(a) Variant M_1^{out} with One Final State

(b) Variant M_2^{out} with Two Final States

Fig. 3. Possible Results for the Flattening of M^{in}

formation under test. Furthermore, this partial verdict requires less effort to be obtained and consumes less resources than a complete one.

3.1 Partial Oracle Data to Focus on Part of the Output Model

The oracle data, which is provided by the tester, consists of two elements: a partial expected model and a set of patterns defining which part of the model is not considered by the oracle. The partial expected model is part of an output model of the TUT. It can also be a comprehensive output model if the tester can provide it. In particular, when the output model is polymorphic, she may provide one variant of the model.

The patterns define which elements of the output model are not considered in the models to produce the partial verdict. Elements belong to the unpredictable part based on the meta-elements (EClass, EAttribute, EReference) they are instance of. Those meta-elements are extracted from the output meta-model, thus our patterns are *meta-model fragments*.

In the example presented in Figure 3, several variants exist, expressing the same semantics; the output model is polymorphic. The final states as well as the transitions targeting them, change from one variant to another, they are not predictable. In Figure 3(a), both transitions have the same target, while in Figure 3(b) each one has a different target. Thus, the unpredictable part of these output models is defined as shown in Figure 4: the instances of FinalState and those of Transition targeting a FinalState[1].

3.2 Model Comparison and Filtering to Control the Predictable Part

We define a partial oracle function by entirely comparing the output model with one partial expected model, then filtering the result of this comparison. Model comparison is already implemented by several tools (e.g. EMFCompare), our proposal focuses on filtering the comparison's result.

In our proposal, any observed difference concerning the unpredictable part is taken off the comparison's result. The verdict is "pass" if the filtered comparison's result is empty, because in this case, there is no difference between produced and expected models' predictable part. Otherwise, the test fails and reveals a fault in the model transformation.

[1] we also consider the guards, actions and events supported by a transition in the experimentation (Section 5).

(a) A FinalState (b) A Transition targeting a FinalState

Fig. 4. Patterns defining the Unpredictable Part for our Flattening Transformation

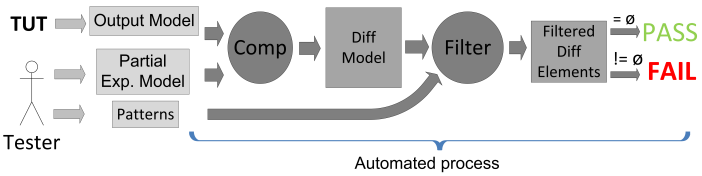

Fig. 5. Our Approach to Produce a Partial Verdict

This result is interesting because the tester detects faults with only a partial expected model when the classical model comparison approach (see Section 2.3) needs at least a comprehensive one before detecting any fault. The filtering patterns, which are written once, are used for every test of a given transformation. Additionally, they are built in a familiar way for a MDE tester, extracted from meta-models. Thus, unlike using specific matching language like ECL (Epsilon Comparison Language) [7], or specializing the model comparison engine for each transformation, she does not have to learn additional language.

Figure 5 summarizes our approach. The tester identifies the unpredictable part, and writes patterns defining it. She provides partial expected models which are compared with TUT's output models. The result of this comparison is then filtered using the patterns. The verdict is produced after observation of this filtered comparison's result.

4 Implementation

In this section, we describe the implementation of the proposed approach and its application to the running example of the paper.

4.1 The Technical Framework

Technically, our oracle function allows testing transformations generating XMI[2] models, as in the Eclipse Modeling Framework[3]. The choice of the EMF framework is due to the number of provided tools, as well as the fact that it is widely used in the academia.

We use EMFCompare[4] to compare our models. For each comparison, EMFCompare produces two result models: the *Match model* for the elements matched between the two models, and the *Diff model* for the differences. In the Diff model, an observed difference is defined as an instance of `DiffElement`. It can concern an `EClass`, an `EAttribute` or an `EReference`.

[2] `http://www.omg.org/spec/XMI/`
[3] `http://www.eclipse.org/emf/`
[4] `http://www.eclipse.org/emf/compare/`

For the filter, we perform pattern matching on the Diff model returned by EMFCompare. We look for any difference concerning an element of the predictable part. The test passes if nothing is matched.

The meta-model excerpts, written by the tester, are Ecore meta-model fragments. Each fragment defines one meta-element (e.g., a transition targeting a final state Figure 4(b)). We need to be able to filter any element that can be concerned by a difference observed by EMFCompare. Therefore, our meta-model fragments can be composed of instances of `EClass`, `EAttribute` or `EReference`.

The pattern matching engine we use is EMF Incquery[5]. With Incquery, patterns are written in a textual form. Thus, we transform our meta-model fragments into Incquery patterns. We wrote a Java transformation, available on a public repository [8], for this purpose. The transformation is integrated into the filter (Figure 5). It takes as input the meta-model fragments defining the unpredictable part of the output model, along with the name of the element the tester wants to be filtered, as well as the root meta-class of the fragment. The root meta-class is the one from which the transformation starts browsing the fragment. For instance, in the fragment from Figure 4(b), Transition is the element the tester wants to filter as well as the fragment's root meta-class. It produces Incquery rules that can be automatically applied on the result of the comparison to return the differences concerning the predictable part only.

4.2 Automatic Treatment of the Patterns in three steps

First Step Since we look for differences between two models, we need to define how a difference is represented in the Diff model. This is the role of the pattern *isDifference*. It basically matches any element for which a difference was observed (an element referenced in a instance of `DiffElement`). This pattern is generic and is generated independently from the TUT.

Second Step In the second step, we generate the Incquery patterns corresponding to the fragments defined by the tester, one pattern for each of the fragments. This step's result for the fragments from Figure 4 is presented in Listing 1.1. For each fragment, we create the header of the pattern with its name (the fragment's name) and its parameter, where the matched elements are collected.

We browse the fragment's elements, starting by the root meta-class specified by the tester. The pattern's body starts with the declaration of a variable instance of this root meta-class. For the pattern from Figure 4(b), the root meta-class is Transition, thus the declaration of the variable T (since Transition is also the meta-element the tester wants to filter, the variable T is the parameter of the pattern).

Then we declare the meta-class' attributes if any. Afterwards, for each reference we declare a variable corresponding to the meta-class target of this reference. Then we declare the link between this new variable and the current

[5] `http://viatra.inf.mit.bme.hu/incquery`

meta-class. We can see in Listing 1.1 the declaration of the FinalState F, as well as the link between T and F.

Finally the same process is applied to this referenced meta-class, until there is no more reference to follow.

```
// A FinalState F
pattern finalState(F) = {
        FinalState(F);
}

// A transition targeting a final State
pattern finalTransition(T) = {
        Transition(T);
        FinalState(F);
        Transition.target(T, F);
}
```

Listing 1.1. Expression of the Non Considered Part with Incquery

Third Step In the third and last step, we generate the pattern that provides the result of the filtered comparison. In the first step, we defined an element about which a difference is observed. In the second step, we defined the unpredictable part's elements. In this final step, we are looking for any of the unpredictable part's elements about which a difference is observed. Therefore, we need to combine the patterns from the previous steps into a new one. In Listing 1.2 we are looking for an instance of FinalState or of Transition targeting a FinalState, about which a difference is observed.

```
/* A is a Difference which is not about a FinalSate
  or a Transition targeting one
  If the result is empty then the test pass for the common part */
pattern verdictPassIfAEmpty(A) = {
        find isDifference(A);
        neg find finalState(A);
        neg find finalTransition(A);
}
```

Listing 1.2. Pattern for the Verdict of the Considered Part

5 Experiments and Discussion

We validate our approach by running several experiments. We build partial test oracles for two model transformations. After describing the case studies, we detail the test protocol set up for these experiments. Afterwards, we discuss the obtained experimental results and potential threats to the validity of our experiments and approach.

5.1 Case Studies

State Machine Flattening transformation This transformation, which flattens UML state machines to remove composite states, is used as an illustrative

example throughout this paper. The implementation of this transformation was developed by Holt et al. [10] in Kermeta. We specify that input model is valid only with no orthogonal state, no pseudostate other than initial state, no hyperedge, and transitions leaving a composite state containing a final state have no trigger.

UML Activity Diagram to CSP transformation This transformation transforms UML activity diagram into model of CSP[6] program.

Part of the output models is unpredictable since models of CSP programs are polymorphic. Indeed, from the specification proposed by Bisztray et al. [11], we identify two elements, which can introduce variations in the output model:

- A decision node is transformed into an n-ary condition. In this situation, the operands can be permuted and several links can be combined. According to the authors, provided that the guards' conditions are distinct, for the UML norm, several syntactic combinations are semantically equivalent.
- A fork node becomes a combination of concurrency operators which are commutative; changing the order of operands does not modify the program's semantics.

For instance, let us consider a fork node transformed into three parallel processes `(F1, F2, F3)`, they can be organized in twelve variants of the same polymorphic model. Firstly, they can be permuted in six different ways:

`F1‖F2‖F3` `F1‖F3‖F2` `F2‖F1‖F3` `F2‖F3‖F1` `F3‖F1‖F2` `F3‖F2‖F1`

Secondly, the concurrency being a binary operator, to compose three processes, one of the two operands is the composition of two processes:

- `(Fi ‖ (Fj ‖ Fk))`
- `((Fi ‖ Fj) ‖ Fk)`

We have implemented this transformation in ATL[7].

Writing partial oracles, the unpredictable part of these CSP models is composed of the binary operators (conditions and concurrencies) and their operands. Therefore, we create 3 patterns which are excerpts of the CSP meta-model. For space reason they are available on [8].

5.2 Testing Protocol

In this experimental section, we want to answer several questions:
Question 1: Can a tester write partial expected models and patterns?
Question 2: Do the patterns and part of expected models can be processed by partial oracle function to produce partial verdict?
Question 3: Is the proposal more appropriate than other oracle functions to produce partial verdict considering part of expected model?

[6] CSP = Communicating Sequential Processes [9]
[7] http://www.eclipse.org/atl

Answering the first question, we produce a set of test models for our cases studies, then we write corresponding partial expected models, and patterns.

Answering the second question, we transform the test input models with the two TUT and we check that partial oracle return partial verdicts.

Answering the third question, we compare the size of the partial expected models with comprehensive models that would have been written without our proposal. Moreover, we check that it would be simpler to get the same partial verdicts with our proposal than with contracts.

The test model creation method was introduced by Fleurey et al. [12] and validated by Sen et al. [13], it relies on the partition of the input domain. We define partitions of the values of each attribute and reference of the input meta-model. Then, different strategies combine those partitions defining model fragments. Finally, for each model fragment, at least one correct input model is created.

We create 94 test models: 30 for the first transformation, 64 for the second. For the first case study, we create 10 fragments using the IFClassΣ strategy, then create 3 test models for each model fragment. For the second case study, using IFCombΠ strategy we create 8 model fragments. We define all the possible combinations, eliminate invalid ones, and produce the models.

5.3 Results

In this subsection, we present results answering the three previous questions. We already answer partially **Question 1** in Section 4.2 when we introduce the patterns for one case study.

Part of the experiments' results is shown in Table 1. For each partial expected model, we present the number of its elements, the number of elements in the complete expected model, the proportion of the predictable part in the model, as well as the number of existing variants and the partial verdict of the test.

UML State Machine Flattening The second result answering **Question 1** is the writing of the partial expected models and the patterns for the filter rejecting unpredictable part of the models. This transformation returns polymorphic models, therefore we can create comprehensive expected models but some of them are only one variant among several possibility. Out of the 30 input models created, 10 do not have any composite state to be transformed. Therefore, their corresponding expected models are exactly the ones we expect. In the opposite, each of the 20 other expected models is only one variant: for instance, we can use the one of the Figure 3(b). Those models are available in [8].

Answering **Question 2**, four test cases produce a failure partial verdict, two of them because of a missing transition in the output model. The other two failed for a missing guard on a newly created transition; more precisely, the guard was created but not linked to the transition.

The three following paragraphs answer **Question 3**. While with most of our test models the number of variants for the transformation is quite small (1, 2 or 5), we can see that one of the output models has 93 variants (model 9 in the

table). In this case without our approach, using the classical model comparison approach would require 93 expected models (about 3,906 elements), along with 93 model comparisons to obtain a verdict.

One could argue that since the majority of the output model elements belong to the predictable part, the tester should just create one model with the predictable part then copy it as much as needed, and then only add the unpredictable part. However in this case with 93 variants, the tester still would have to create 1,238 model elements (29 + (42 - 29) * 93) and 93 comparisons would still be needed, before getting the least verdict.

Whereas the transformation is 500 lines, we write 510 lines worth of contracts. Three of them concern the unpredictable part of the output model. Even though the contracts are more complex than the transformation, they do not cover the whole output models' predictable part. For instance, the transitions' effects are not controlled with the contracts. On the contrary, our approach permits to entirely control the predictable part, which is present in our partial expected models.

Activity Diagram to CSP The experiment on this second transformation answer the **three questions** again. Four test cases produce a failure partial verdict, when two or more join nodes are present in the input model only the first is correctly transformed. The maximum number of variants is 96 for 3 models. Without our approach, the tester would have to create the 96 expected models: 4,800 elements or 715 elements (43 + (50 - 43) * 96) by copying the predictable part elements.

Once again, we write contracts which are as complex as the transformation (218 lines of contracts and 210 of transformation), but do not entirely control the output models' predictable part. For instance they do not control the order in which the instances of ProcessAssignment are defined, when our partial expected models do. Two of the contracts we write partially control the unpredictable part.

Discussion We obtain partial verdicts from the use of partial expected models and patterns with our oracle function, thus answering Question 2. To obtain partial verdicts we create only 94 partial expected models (2,632 elements) and 8 patterns (18 elements) and perform 94 model comparisons, instead of 835 models (36,184 elements or 8,677 elements by copying the predictable parts) and 835 model comparisons for the classical model comparison approach. The gain here is of 93% in terms of model elements (70% if copy of the predictable part) and of 89% in terms of model comparisons. Our case studies handle simple models with an average of 35 and 37 elements per model. The gain for the tester would be greater with transformations handling more complex models. It would be decisive if she would manually write those models' variants.

We write incomplete contracts that are as complex as our transformations. However, these contracts do not entirely control the predictable part. Contracts controlling a whole transformation are at least as error-prone as the transformation's implementation. Thus global contracts are not suited for our needs.

(a) State Machine Flattening

expected model	#considered part elements	# elements in a comprehensive variant	%considered part	#variants	partial verdict
1	32	36	89%	1	pass
4	35	42	83%	2	pass
5	31	35	89%	1	fail
7	32	42	76%	5	pass
8	32	39	82%	2	fail
9	29	42	69%	93	fail
14	12	17	71%	2	fail
17	18	25	72%	5	pass
...					
avg	28	35	79%	6.71	
sum	582	734		157	

(b) UML To CSP

expected model	#considered part elements	# elements in a comprehensive variant	%considered part	#variants	partial verdict
4	32	38	84%	2	pass
7	36	42	86%	4	fail
8	36	42	86%	4	fail
10	35	42	83%	6	pass
13	29	34	85%	12	pass
16	40	48	83%	24	fail
17	40	48	83%	24	fail
18	43	50	86%	96	pass
...					
avg	32.03	37.3	87%	10.6	
sum	2050	2338		678	

Table 1. Observed Results for our Case Studies

Answering the third question, we can conclude that our proposal is more appropriate than the other oracle functions to provide a partial verdict considering part of an output model.

One could argue that we do not control the correctness of the whole output model. However, first, our partial oracles find errors in those transformations, one of them not being ours [10]. Second, the predictable part is a significant part of our output models (over 61%). Third and last, in both case studies, elements in the predictable part are transformed. The transformations do not only act on the unpredictable part of a model. In the first transformation, simple states are not modified, but incoming or outgoing transitions of the composite states are (relation between A and B in Figure 3). In our second transformation, we transform from one language to another, the input and output meta-models are different. So this partial verdict is a good piece of information for the tester.

Also to fill the gap of our only partial verdict, we could use contracts. While global contracts are too complex, we could use smaller ones to control the unpredictable part. This way we could benefit from our approach and add simple contracts to obtain a complete verdict.

5.4 Threats to Validity

We have successfully applied our approach to build partial test oracles for two model transformations. A major threat to the validity of our experiments is the question of the representativity of our case studies. Can we conclude about the efficiency of our approach for any model transformation? While with our case studies we do not cover the whole range of possibilities of model transformations, they still are quite distinct from one another. One, a refactoring transformation, modifies only part of the input model; the other transforms the input model into

a completely different one. Also the first one directly transforms UML models, its inputs and outputs conform to the UML model.

Another threat, this time to the global usability of our approach is the problem of the identification of the predictable part. Unfortunately this step is still manual, with the tester having to understand the transformation's specification. Yet she can find clues, for instance elements that indicate possible polymorphism, such as binary operators for polymorphic outputs, like in our second case study. If part of the specification is too complex for her to handle, then she should be able to describe it. When performing regression testing of a refactoring transformation, the part of the meta-model appearing in the specification is the one modified by the transformation, therefore this part is the unpredictable part.

6 Related Work

Model transformation testing has been studied several times.

Model transformations can be seen as graph transformations, Darabos et al. [14] tested such transformations. They identified and classified most common faults in erroneous transformations.

In Section 2.3, we discussed the use of generic contracts for the oracle. Braga et al. [15] [16], specify a transformation using a meta-model. In order to verify a transformation they use contracts expressed as a transformation meta-model along with a set of properties over it. Similarly, Büttner et al. [17] model an ATL transformation for its validation. Cabot et al. [18] also build on this concept of transformation model, to which they add an invariant. Guerra et al. [19] developed transML, a family of modeling languages for the engineering of transformations. They generate both test inputs and oracles, relying on a transML specification of the TUT. This specification can be seen as contracts. As we already mentioned the main drawback of contracts is that they are as complex to write as the transformations and thus as error prone.

Tiso et al. [20] are particularly interested in testing model to text transformations. They use assertions to check properties on the produced output. They argue that the assertions' writing should be white box, because they have to know of some choice made by the developer. Several syntaxes exists for the same semantics, this is a case of polymorphic outputs. As discussed in Section 2.1, the oracle is strictly black box, thus their approach is not suited.

7 Conclusion and Future Work

In this paper, we presented an approach to write a partial oracle to validate a part of output models. This approach produces a partial verdict by comparing the output model with an expected one. Running experiments on two transformations with polymorphic outputs, we measured our approach against two classical oracle functions, global model comparison and contracts. These experiments showed that our approach is more appropriate to produce a verdict when considering part of the output model.

Beyond this work, the next step is to complete the partial verdict we obtain with our approach. As discussed in Section 5.3, our idea is to use small contracts to control the unpredictable part's properties.

References

1. J.-M. Mottu, B. Baudry, and Y. Le Traon, "Model transformation testing : oracle issue," in *MoDeVVa'08.*, 2008.
2. Y. Lin, J. Zhang, and J. Gray, "A testing framework for model transformations," *Model-driven software development*, pp. 219–236, 2005.
3. A. García-Domínguez, D. Kolovos, L. Rose, R. Paige, and I. Medina-Bulo, "Eunit: a unit testing framework for model management tasks," *MoDELS*, 2011.
4. A. Cicchetti, D. Di Ruscio, and A. Pierantonio, "A Metamodel Independent Approach to Difference Representation," *JOT*, 2007.
5. E. Cariou, N. Belloir, F. Barbier, and N. Djemam, "Ocl contracts for the verification of model transformations," *ECEASST*, 2009.
6. A. Vallecillo, M. Gogolla, L. Burgueño, M. Wimmer, and L. Hamann, "Formal specification and testing of model transformations," in *SFM*, 2012, pp. 399–437.
7. D. Kolovos, "Establishing correspondences between models with the epsilon comparison language," in *Model Driven Architecture - Foundations and Applications*, 2009, vol. 5562, pp. 146–157.
8. O. Finot, J.-M. Mottu, G. Sunyé, and C. Attiogbe, "Experimentation material," https://sites.google.com/site/partialverdictmt/.
9. C. A. R. Hoare, "Communicating sequential processes," 1978.
10. N. Holt, E. Arisholm, and L. Briand, "An eclipse plug-in for the flattening of concurrency and hierarchy in uml state machines," Tech. Rep., 2009.
11. D. Bisztray, K. Ehrig, and R. Heckel, "Case study: Uml to csp transformation," *AGTIVE*, 2007.
12. F. Fleurey, B. Baudry, P.-A. Muller, and Y. Le Traon, "Qualifying input test data for model transformations," *SOSYM*, 2009.
13. S. Sen, B. Baudry, and J. Mottu, "Automatic model generation strategies for model transformation testing," in *ICMT 09.*, 2009.
14. A. Darabos, A. Pataricza, and D. Varró, "Towards testing the implementation of graph transformations," *ENTCS*, vol. 211, 2008.
15. C. Braga, R. Menezes, T. Comicio, C. Santos, and E. Landim, "On the specification, verification and implementation of model transformations with transformation contracts," in *SBMF*, 2011.
16. C. de O. Braga, R. Menezes, T. Comicio, C. Santos, and E. Landim, "Transformation contracts in practice," *IET Software*, vol. 6, no. 1, pp. 16–32, 2012.
17. F. Büttner, J. Cabot, and M. Gogolla, "On validation of atl transformation rules by transformation models," in *MoDeVVa*, 2011.
18. J. Cabot, R. Clarisó, E. Guerra, and J. De Lara, "Verification and Validation of Declarative Model-to-Model Transformations through Invariants," *JSS*, vol. 83, 2010.
19. E. Guerra, "Specification-Driven Test Generation for Model Transformations," in *ICMT*, 2012, pp. 40–55.
20. A. Tiso, G. Reggio, and M. Leotta, "Early experiences on model transformation testing," in *Proceedings of 1st Workshop on the Analysis of Model Transformations (AMT 2012 co-located with MoDELS 2012)*, 2012.