On Reducing Model Transformation Testing Overhead

Roberto Rodriguez-Echeverria[1], Fernando Macias[2,3], and Adrian Rutle[2]

[1] University of Extremadura, Cáceres, Spain,
rre@unex.es
[2] Bergen University College, Bergen, Norway,
{Adrian.Rutle, Fernando.Macias}@hib.no
[3] University of Oslo, Norway

Abstract. Model-Driven Engineering is essentially based in metamodel definition, model edition and the specification of model transformations (MT). In many cases the development and maintenance of these transformations are still carried out without the support of proper methods and tools to reduce the effort and cost related to them. As a result, several model transformation testing approaches have been defined over the past decade. However, they usually introduce some kind of overhead on transformation development or maintenance which has not been properly considered heretofore. In this work, we will present MoTe, a novel contract-based model transformation testing approach specifically designed to reduce testing overhead. First, we present main overhead sources in core stages of contract-based model transformation testing.
Then, we give a detailed introduction of the main features and design decisions behind MoTe in order to reduce overhead in all those main stages. Furthermore, we will discuss some significant results from a comparative case study.

Key words: Model-Driven Engineering, Model transformation testing, Contract-based specification, Testing Oracle.

1 Introduction

Over the past decade, several model transformation testing (MTT) approaches have been defined [3,2,12,9]. They allow engineers to specify and run regression tests to assess the behavior of their model transformations. However, these approaches usually introduce enough overhead to hinder their wider application. In model transformation testing, contract-based approaches are organized in three main stages: (1) contract specification, (2) test oracle execution, and (3) result interpretation.

Different activities in these stages may introduce significant overhead. However, it has never been considered before. Reducing testing overhead is a primary goal for our approach in order to become a viable alternative in industrial scenarios.

In this paper, we present the contract-based MTT approach, named MoTe, which we briefly introduced in [11], as one of the main parts of our approach to model transformation evolution assistance. In our previous work we focused on result interpretation, while contract specification and execution are elaborated in this paper.

This paper is organized as follows. In Section 2 we introduce the main aspects causing overhead in MTT. In Section 3 we present a domain-specific language addressing the specific concepts on which MoTe relies. Section 4 introduces MoTe's execution process and technology. Section 5 highlights the main contributions of MoTe concerning result interpretation. In Section 6 we show an example of applying MoTe in a comparative case study. Section 7 discusses related and future work, and outlines a conclusion.

2 Testing overhead

Testing overhead is obviously a necessary tradeoff between development cost and degree of behaviour verification. In order to reduce this overhead we should first identify how it is produced. We will decompose testing overhead according to the three main stages of contract-based MTT to gain some insight about how the different activities in those stages may introduce overhead from a MT developer's viewpoint. We have intentionally not included test-model generation because we are not interested in that stage now.

Contract specification refers to the process of defining invariants between input and output models, and it is achieved by using concepts, tools and other artefacts for edition and maintainance of contracts. At this stage we then focus on specification overhead. Overhead decomposition may be defined by analysis the following aspects related to specification:

- **Language for contract specification.** Some MTT approaches [12] propose OCL to define invariants between input and output models (contracts), while others [8] argue that OCL is not the right choice to make. Generally, using a DSL has clear advantages, e.g., simplifying learning and understability, and reducing maintenance cost. DSL usage may then clearly reduce edition overhead.
- **Concrete syntax.** Assuming the existence of a DSL, some approaches [8] argue graph-based syntax is clearly more appropriate for input/output patterns definition, which are key for contract specification. However, user diversity [4] is a more adequate property to satisfy when arguing about user experience, so that every user could use the syntax that suits her better for a concrete project. As a result, forcing a MT developer to use a syntax could conceivable be a source of specification overhead.
- **Contract verification approach and tooling.** Contract conflicts, i.e., contradictory contracts or redundant contracts, should be identified in order to properly assist the MT developer in contract specification. The complexity of this activity and the kind of tools needed are dependent on language complexity. So a tradeoff between language expressiveness and contract verification effort should be made in order to avoid unnecessary specification overhead for this reason.

Test oracle execution refers to the process and technology used for actually running the contract checking (execution overhead). A further overhead decomposition may be defined by analyzing the following aspects related to execution:

- **Invasive process.** Basically, should input or output models be modified in order to facilitate invariant checking? For example, Tracts [12] uses an OCL interpreter

which cannot use two different models as input, hence, input and output should be combined into a single model to check invariants. Obviously, invasive proccesses delve into an increment of the execution overhead.
- **Technology.** Execution technology may introduce some significant overhead in the case of requiring specific tools built upon a different technology stack than the one used by MT developers. In this case, we can consider configuration overhead, but also learning overhead. Both may be drastically reduced when keeping the same technology stack for test oracle artefacts, which may then be easily inspected.

Result interpretation refers to how much effort is needed to make results actionable, i.e., useful to make a decision or to take an (automatic) action. If testing results are not focused on rapidly addressing the cause of the error, then its interpretation may have a negative impact on testing overhead. A further overhead decomposition may be defined by analysing the following:

- **Testing style.** Different testing styles yield different result reports. Contract-based MTT approaches usually follow unit testing style, so they basically report about contract (test) failing or passing. Result reporting is just a list of passing and failing tests, which should usually be manually analysed (interpretation overhead). Nevertheless, this testing style might not be always the best choice and alternative styles should then be considered. Result reporting should be designed for quick interpretation according to the MT developers' main goal.
- **Type of results.** A binary value is a low-level type of result which demands human intervention for its proper interpretation inside of a context. Although binary values might be enough for unit testing, other testing styles might demand more meaningful results which allow easily understanding how a MT is behaving. In this sense, metrics (aggregated numeric values) might give additional information about an error, such as its extension for a real input or some hints about its type.
- **Focus of results.** Test result reports that are focused on contracts are only useful to locate failure, while result reporting focused on output may give extra information, e.g., how many output elements are affected by a specific error. Furthermore, we believe output-focused result reporting aligns better with the way MT developers think, because they are thinking about the output, not in the contracts, when they build MTs. Therefore, output-centred result reporting may reduce interpretation overhead.
- **Result actionability.** This aspect may encompass a range of different probable actions, but the most common may be to detect the source of error and to identify proper repairing actions. In general, we assume that the higher the degree of result actionability the less overhead is generated on result interpretation.

3 Contract Specification

According to the necessities of MoTe computation, we have defined a DSL which provides the developer with a concise syntax specifically designed for the definition of testing contracts focused on our metrics calculation. Moreover, given that rule-based MTs have been the norm in our main application scenarios heretofore, we decided to

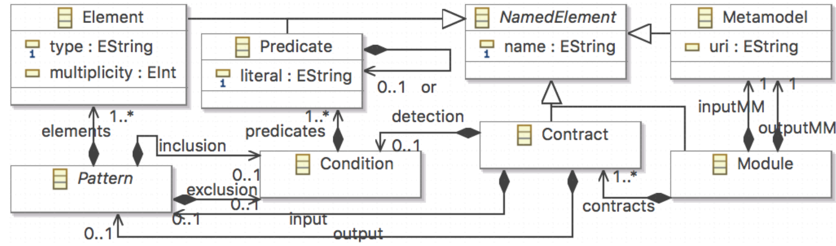

Fig. 1. MoTe Metamodel

define a textual concrete syntax for our language. Nevertheless, regarding user diversity, the definition (reutilization) of graph-based syntaxes remains as a subject of study.

3.1 Language

The abstract syntax of MoTe is defined as an Ecore metamodel which is partially shown in Fig. 1. Its primary element is `contract` which allows setting a relationship between an input pattern and an output pattern. Basically, contracts allow defining constraints between input and output models, and hence constraints over the transformation.

The main components of a contract are input pattern, output pattern and detection criteria. Input and output patterns are defined according to a collection of pairs (`Type:variable`) and some conditional criteria (inclusion and exclusion). Every pair indicates an existing Type in the input or output metamodels, and also a name (or variable) that is convenient to be able to define conditional expressions related to input or output elements. Therefore, those names will be used as variables in the definition of the inclusion and exclusion criteria, which allow expressing pattern characteristics in a positive or negative way. It is important to stress that only input metamodel types are allowed in the input pattern, while correspondingly only output metamodel types are allowed in the output pattern. As a result, criteria defined for the input pattern cannot contain expressions based on output elements, and viceversa. Only detection criteria can specify expressions concerning elements from input and output elements, because its mission is to express conditions that input-output relationships must hold (invariants).

Although the definition of a contract demands to specify input and output patterns, it is not necessary that both are defined for two special cases: preconditions and postconditions. To define a precondition with a contract only its input pattern should be specified. Meanwhile, to define a postcondition with a contract only its output pattern should be specified. Detection criteria is not necessary in those special cases because there is no need to define a relationship between input and output. For the sake of brevity, we do not consider preconditions and postconditions in the rest of this work.

3.2 Concrete syntax

The textual concrete syntax of MoTe is defined using EMFText [1]. This simple syntax permits specifying all the relevant concepts of our approach. As shown in the following

listing, input and output patterns are specified as a list of (`Type:variable`) pairs, while all the different criteria are specified as a list of predicates.

```
1  contract <cname>{
2   input: (<iT1>:<iv1>, ... <iTN>:<ivN>)
3    inclusion: <p_name>(<p_expresion>)...
4    exclusion: <p_name>(<p_expresion>)...
5
6   output: (<oT1>:<ov1>, ... <oTN>:<ovN>)
7    inclusion: <p_name>(<p_expresion>)...
8    exclusion: <p_name>(<p_expresion>)...
9
10  detection: <p_name>(<p_expresion>)...
11 }
```

Regarding criteria specification, in order to reduce overhead we consider simplicity as a key property for the DSL, which may be seen as a threefold target:

- **Simple syntax.** Criteria specification and understandability should be easy. Contracts are mainly defined at a more abstract level than actual transformations. Therefore, contracts should be easier to define than actual transformations.
- **Simple checking.** Contract verification should be performable from a pragmatical point of view (decidable semantics).
- **Simple debugging.** The more information we get about the violated contract the better. Therefore, a fine-grained definition of criteria should be provided, so that every violated condition can be traced back.

In this work, predicate expressions are used for the specification of all the different criteria of a contract (inclusion, exclusion and detection). Basically, each predicate has got a name and allows defining equality expressions. Moreover, predicates can be logically connected by conjunction or disjunction operators. For example, the following predicates, which could conceivably be part of a detection criteria, state that input and ouput elements should have the same name and color:

```
p1(input.name = output.name) and p2(input.color = output.color)
```

3.3 Contract verification

The semantics of our DSL is based on graph transformations (GT) [7], more precisely, typed attributed graph transformations with type-inheritance [6]. This version of GT is inspired by UML- and EMF- like models in which graph nodes can have (i) attributes (with values from a predefined domain, e.g., String or Integer), and (ii) inheritance edges which are used in the same sense as in object oriented models.

In GT we use production rules of the form $p : L \rightarrow R$, where L and R are graphs and p is a graph morphism which maps elements from L to R. For attributed GT, one can pose conditions over values of the attributes while picking/calculating the matches (during application). These conditions are defined as expressions over elements from the source and target graphs as seen in [9,8]. We formalise the contracts as production rules in attributed GT. The input pattern corresponds to L while the output pattern corresponds to R. Exclusion and Inclusion criteria correspond to NAC and PACs, respectively. The detection criteria corresponds to the graph morphism from L to R and the conditions

which are posed on this morphism. By employing GT as the formal foundation, we can rely on the rich theory and results of GT for verification of the contracts, e.g., one can reason about conflicts, contradictions and relations between contracts [8].

4 Test oracle execution

As presented in [11], our test oracle excution consists on the computation of precision and recall metrics for every relation between input and ouput patterns defined by a contract. Every contract defines a partition of the candidate set of input patterns into four subsets. Inside the candidate set, two other subsets are defined: Transformed and Excluded. All the input patterns inside the former are those actually generating output patterns. To find them we propose the definition of detection criteria. And, on the contrary, the input patterns contained in the Excluded subset are those that should not be transformed. Using those three sets, it is possible to obtain the number of True Positives (TPs), False Positives (FPs) and False Negatives (FNs) generated by the MT according to each contract and eventually to calculate the values of precision and recall for each output pattern. As a result, we can get an overall measure of the (light) correctness of all the transformation rules generating each output pattern.

4.1 Process

Metrics computation is implemented by a two-step process: (1) a new model is generated by a model transformation as a result of querying input and output models and marking every input element as TP, TN, FP or FN; and (2) this result model is queried in order to aggregate all the results and provide the developer with an easy-to-understand comprehensive report. This process is executed seamlessly for the MT developer. Input and output preprocessing is not necessary in order to execute our test oracle.

Result models have only one type of element which represents the result of checking a contract for an input pattern. These elements are basically composed of an id attribute, a reference to the input pattern (main element), a reference to the output pattern (main element), a reference to the contract checked and a value attribute with the result of checking the contract involved (TP, FP, TN, FN).

4.2 Execution technology

In order to compute the metrics, contracts should be compiled to a representation that may be runnable on an execution engine. It would be also possible to interpret those contracts by means of a specialized engine, however this alternative is not discussed in this work. For the context of this work, we have selected ATL [10] as the target representation, although any other language could be selected to not force developers altering its technology stack. The contracts in our DSL could be straightforwardly compiled to an ATL transformation. To this end, basically (exceptions not commented here), an ATL matched rule should be generated for every contract with the following structure (we are supposing all the elements have a name attribute):

```
1  rule <contract_name> {
2  from
3   input :<input_type>
4          <input_inclusion_criteria>
5  to
6   result : resultMM!Result (
7    inputType  <- inType,
8    outputType <- outType,
9    contract   <- contract,
10   value      <- thisModule->
11    resultValue(input.isExcluded,
12               input.isTransformed)
13  ),
14
15  inType : <input_type> (
16   name <- <input_type>.name
17  ),
18
19  outType : <output_type> (
20   name <- <output_type>.name
21  ),
22
23  contract : resultMM!Contract (
24   name <- '<contract_name>'
25  )
26
27  do {
28   result.inputType.object <- input;
29   result.outputType.object <-
30    thisModule->
31     getOutput(<output_type>,
32              input.name);
33  }
34 }
```

As shown, the ATL rule template has got 4 different explicit parameters: contract name, input type, output type, and input inclusion criteria. Additionally, it has got 3 "implicit" parameters: input exclusion criteria, output inclusion criteria, and detection criteria. The latter parameters are used to generate final `isExcluded` and `isTransformed` helpers for the input type.

Each ATL rule generates a result element into the result model. Basically, each result element stores: a relationship between a concrete input element and a concrete output element (`inputType` and `outputType`); the type of result obtained according to the contract (i.e., TP, TN, FP or FN); and the name of the contract. Additionally, every result is assigned a different `id` to facilitate likely debugging tasks. A helper (`resultValue`) is in charge of returning the corresponding value of the result with respect to `isExcluded` and `isTransformed` additional helpers, which are responsible for implementing exclusion and detection criteria.

As aforementioned, the output of this ATL transformation is a result model which has labelled every input element with respect to the contract in which it is located and, additionally, has related them with their corresponding output elements in the case they have been generated, i.e., TP and FP cases.

5 Result Interpretation

5.1 Testing style

Contrary to other common contract-based model testing approaches providing unit testing for MTs, our approach is more an acceptance testing solution because we provide developers with easy to interpret and comprehensive information about the behaviour of the MT according to her requirements (expressed as contracts). Therefore she can make an educated decision whether the MT behaviour is acceptable or not. From our point of view, different application scenarios require different degrees of correctness to consider a MT acceptable, e.g., in a Model-driven Reverse Engineering scenario some degree of incorrectness might be acceptable when it is clearly located and it is extremely less expensive to build a postproccesor to solve the problem than modifying the MTs involved.

Table 1. Possible results

Case	1	2	3	4	5	6	7	8
Pr	100	t	100	t	0	0	NA	NA
Rc	100	100	t	t	0	NA	0	NA

5.2 Type of results

MoTe computes precision and recall metrics as numeric results of the test oracle to provide an evaluation of MT actual behaviour compared to expected behaviour. To simplify result interpretation we reduce the set of possible values for the metrics to 4 values: 100 for perfect, 0 for worst result, t for a value in the range of 0 and 100 (threshold), NA for division by zero. According to these values, Table 1. presents the categorisation of all possible result combinations. Case 1 represents the perfect situation: all expected elements have been generated and all generated elements were expected. Cases 1, 2, 3 and 4 define the most common situations, while cases 5, 6 and 7 are specializations of them. Case 8 is an exceptional case: none of the contracts for that output element is applicable on the input model. As a result, categorised meaningful results are yielded by MoTe, reducing interpretation overhead.

5.3 Focus and actionability of results

In contrast to other approaches, MoTe reports results focusing on output instead of contracts as the aggregation criteria. Therefore, developers get a summary report for each output pattern taking into account all the different contracts constraining the generation of each pattern. This summary basically consists of the result case obtained for this output pattern. When some error case is yielded, a detailed report can be obtained to know the number of TP, FP, TN, FN marks or, even, to receive precise information about the specific predicate of a contract not satisfied for a concrete input-output instance.

Moreover, results are basically serialized as CSV files. Therefore, they can be visualised or further proccessed by mainstream tools, such as spreadsheets. At the same time, given the categorisation of each result and all the additional information about errors provided, MoTe is designed to produce results with high degree of actionability. For example, in [11] we presented a proccess for suggesting repairing actions built upon the results produced by our testing oracle.

6 Case Study

To validate our approach we have performed the case studies proposed by [5]. In this section, for the sake of brevity, we just present a single example of this comparative study. For example, the following code illustrates the specification of Tract 10 for the UML2ER case[4], which constraints the generation of `StrongReference` elements.

[4] http://atenea.lcc.uma.es/index.php/Main_Page/Resources/MTB/UML2ER

UML2ER - Mutant03								
Output	Input	TP	FP	TN	FN	Pr	Rc	Case
ERModel	Package	1	0	0	0	100	100	1
EntityType	Class	4	0	0	0	100	100	1
Attribute	Property	2	0	0	0	100	100	1
WeakReference	Property	2	0	0	0	100	100	1
StrongReference	Property	0	0	0	1	#¡DIV/0!	0	7

Fig. 2. Mutant 03 results

```
1  contract C10{
2   input : (Property:i)
3    exclusion: p1(i.complexType = null) or p2(i.isContainment = false)
4   output : (StrongReference:o)
5   detection: p3(i.name = o.name)
6  }
```

All MoTe contracts for the UML2ER case have been compiled to ATL and our test oracle has been executed with the same input model. For example, it has produced the results presented in figure 2 for mutant 03 which modifies a filter in rule 8 (`StrongReference` rule). We can observe that case 7 is the result obtained for `StrongReference`, which indicates that the only candidate input element, according to the previous contract, has not been transformed (FN).

7 Related Work, Future Work and Conclusions

To the best of our knowledge, this is the first work discussing overhead in model transformation testing. Therefore, we only discuss about related contract-based MT testing approaches herein. Although several approaches can be found in the literature [2], we are just comparing MoTe to Tracts [12,5] and PAMOMO [8] mainly because they are two of the best documented approaches. Table 2 summarizes our comparison according to the overhead aspects we have previously defined.

In this work we have presented MoTe, a novel contract-based model transformation testing approach specially designed to reduce testing overheadA testing framework for model transformations.

As future work we plan to study results' actionability in order to automatically repair MTs. Further research on MTT overhead analysis and reduction also appears as part of our short-term purposes. For example, human-based studies to assess whether MoTe reduces interpretation overhead.

Acknowledgements

Partially funded by Junta de Extremadura, Order 129/2015, 2nd of June.

References

1. EMFText project. `http://emftext.org/`. [Online; accessed 15-July-2016].

Table 2. Tracts, PAMOMO and MoTe comparison

	Tracts	**PAMOMO**	**MoTe**
Language	OCL	DSL	DSL
Concrete syntax	Textual	Visual	Textual currently
Contract verification	None	Graph theory	Graph theory/FOL
Process	Invasive	Seamless	Seamless
Technology	USE tool	Different stack	Same stack
Testing style	Unit testing	Unit testing	Acceptance testing
Type of results	Binary	Binary	Metrics
Focus of results	Contracts	Contracts	Output
Result actionability	Low	Low	High

2. Lukman Ab. Rahim and Jon Whittle. A survey of approaches for verifying model transformations. *Softw Syst Model*, 14(2):1003–1028, June 2013.
3. Benoit Baudry, Sudipto Ghosh, Franck Fleurey, Robert France, Yves Le Traon, and Jean-Marie Mottu. Barriers to Systematic Model Transformation Testing. *Commun. ACM*, 53(6):139–143, June 2010.
4. Pierre Bourque and Richard E. Fairley. *Guide to the Software Engineering Body of Knowledge - SWEBOK v3.0*. IEEE CS, 2014 version edition, 2014.
5. Loli Burgueño, Javier Troya, Manuel Wimmer, and Antonio Vallecillo. Static Fault Localization in Model Transformations. *IEEE Transactions on Software Engineering*, 41(5):490–506, 2015.
6. M. Cerioli, T. Margaria, M. Wermelinger, Juan de Lara, Roswitha Bardohl, Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. Fundamental aspects of software engineering attributed graph transformation with node type inheritance. *Theoretical Computer Science*, 376(3):139 – 163, 2007.
7. Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2006.
8. Esther Guerra, Juan de Lara, Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, and Wieland Schwinger. Automated verification of model transformations based on visual contracts. *Automated Software Engineering*, 20(1):5–46, 2013.
9. Esther Guerra and Mathias Soeken. Specification-driven model transformation testing. *Softw. Syst. Model.*, 14(2):623–644, 2013.
10. Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A model transformation tool. *Science of Computer Programming*, 72(1-2):31–39, 2008.
11. Roberto Rodriguez-Echeverria and Fernando Macías. A Statistical Analysis Approach to Assist Model Transformation Evolution. In *18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 226–235, Ottawa, Canada, 2015.
12. Antonio Vallecillo, Martin Gogolla, Loli Burgueño, Manuel Wimmer, and Lars Hamann. Formal Specification and Testing of Model Transformations. In *Formal Methods for Model-Driven Engineering*, pages 399–437. Springer Berlin Heidelberg, 2012.