Testing MOFScript transformations with HandyMOF

Jokin García, Maider Azanza, Arantza Irastorza, Oscar Díaz

Onekin Research Group, University of the Basque Country (UPV/EHU)
San Sebastian (Spain)
(jokin.garcia,maider.azanza,arantza.irastorza, oscar.diaz)@ehu.es

Abstract. Model transformation development is a complex task. Therefore, having mechanisms for transformation testing and understanding becomes a matter of utmost importance. Understanding, among others, implies being able to trace back bugs to their causes. In model transformations, causes can be related with either the input model or the transformation code. This work describes *HandyMOF*, a tool that first eases the transition between the effect (i.e. generated code file) and the causes (i.e. input model and transformations) and then provides the means to check the transformation coverage obtained by a test suite. The challenges are twofold. First, the obtainment of input model suites which yield to a quantifiable transformation coverage. Second, providing fine-grained traces that permit to trace back code not just to the transformation rule but to the inner 'print' statements. A transformation that generates *Google Web Toolkit (GWT)* code is used as the running example.

1 Introduction

Transformations rest at the core of *Model Driven Engineering (MDE)*. As any other piece of software, transformations need to be designed, programmed and tested. This last step becomes even more important if we consider that each transformation can potentially generate multiple applications, to which its errors would be propagated [15].

Nevertheless, testing model transformation has proved to be a tough challenge [1]. Compared to program testing, model transformation testing encounters additional challenges which include the complex nature of model transformation inputs and outputs, or the heterogeneity of model transformation languages [17]. To face this situation, both black-box techniques [3,5,16] and white-box techniques [6,8,10] have been proposed. These two approaches are complementary and should be carried out in concert. In black-box techniques the challenge rests on coming up with an adequate set of input models. On the other hand, white-box techniques capture the mechanics of the transformation by covering every individual step that makes it up [1]. We concentrate on the latter, particularly focusing on *Model-to-Text (M2T)* transformations, which have received little attention. Specifically, *MOFScript* language [1] is used along the paper.

[1] http://modelbased.net/mofscript/

The drawback of white-box testing approaches is that they are tightly coupled to the transformation language and would need to be adapted or completely redefined for another transformation language [1]. While standards [14] or well established languages [9] exist in *Model-to-Model (M2M)* transformation languages, the situation is more blurred in M2T transformations. This is the reason why, while aiming at the same goals as white-box testing (i.e., covering every step of the transformation), we opted to realize it using a mixed approach. The model test suite is generated using black-box techniques and then both input models and the generated code are traced to the transformation. The purpose is twofold: (1) if a bug is detected in the generated code, it can be traced back to the transformation line that generated it, and (2) the transformation coverage obtained by the model test suite can be calculated based on transformation lines being transited.

Consequently our approach heavily rests on trace models. Broadly, trace models need to capture a ternary relationship between the source model elements, the transformation model elements, and the generated code. We chose MOFScript as the M2T transformation language as it already supports traceability between source model elements and locations in generated text files [12]. That is, it is possible to trace back the generated code from the source elements. Unfortunately, the third aspect (i.e. the transformation model elements) is captured at a coarse-grained granularity: the transformation rule. This permits coverage analysis to be conducted at the rule level (i.e., have all transformation rules been enacted?) but it fails to provide a deeper look inside rules' code. It would be similar to programming language testing stopping at the function calls without peering within the function body. Transformation rules might in themselves be complex functions where conditional statements and loops abound. Rule-based coverage might then fail to consider the diversity of paths which are hidden in the rule's body.

On these grounds, we complement MOFScript's native trace model with a second model that enables traceability between fine-grained transformation elements (e.g., 'print' and 'println' statements) and locations in generated text files. An algorithm is introduced to aggregate trace models to ascertain which 'print' statements have not yet been visited during testing so that designers can improve their testing model suites to obtain full coverage. These ideas are realized in *HandyMOF*, a debugger for MOFScript transformations. A video of MOFScript at work is available[2]. We start by setting the requirements.

2 Setting the Requirements

A common methodology for code testing generally comprises a number of well known steps: the creation of input test cases (i.e., the test suite), running the software with the test cases, and finally, analyzing the goodness of the results. Next paragraphs describe some of the challenges brought by transformation testing.

[2] http://onekin.org/downloads/public/screencasts/handyMOF

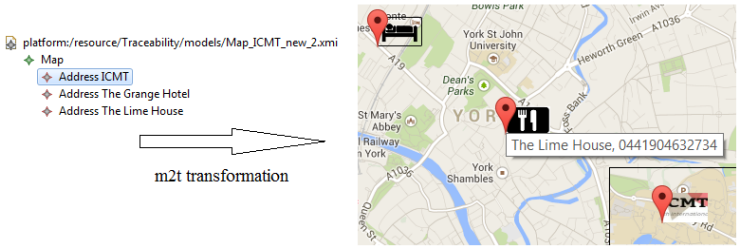

Fig. 1. Input map model and desired output

Creation of test suites. Obtaining the appropriate *test suites* becomes critical to ensure that all the transformation variations are covered, and hence, representative code samples are obtained. So far, different proposals have been made for black-box testing of transformations, based on metamodel coverage [4,16]. Specifically, *Pramana* is a tool that implements black-box testing by automatically generating 'model suites' for metamodel coverage [16].

```
1  var index:Integer = 1;
2  ec.Map::main() {
3    [...]
4    f.println("public void onModuleLoad() {");
5    f.println("MapWidget map = new MapWidget();");
6    f.println("map.setSize(\"1000\", \"500\");");
7    f.println("map.setZoomLevel(14);");
8    ec.objectsOfType(ec.Address)->forEach(ecc:ec.Address) {
9      ecc.address();
10   }
11   f.println("RootPanel.get(\"mapContainer\").add(map);");
12   f.println(" }");
13 }
14 ec.Address::address(){
15   f.println("LatLng point"+ index +"= LatLng.newInstance(" + self.latitude + "," + self.longitude + ");");
16   f.println("MarkerOptions markeroptions" + index + " = MarkerOptions.newInstance();");
17   f.print("markeroptions" + index + ".setTitle(\"" + self.name + ", " + self.description);
18   if(self.description = "restaurant"){
19     f.print(", " + self.telephone);
20   }
21   f.println("\");");
22   f.println("Marker marker" + index + " = new Marker(point" + index + ", markeroptions" + index + ");");
23   f.println("LatLng sw" + index + " = LatLng.create("+self.latitude +","+self.longitude+");");
24   f.println("LatLng ne" + index + " = LatLng.create("+ self.latitude +","+self.longitude +");");
25   self.pictures->forEach(pic){
26     f.println("LatLngBounds bounds"+index+" = LatLngBounds.create(sw"+index+", ne"+index+");");
27     f.println("GroundOverlay go"+index+" = new GroundOverlay(\""+  pic +"\", bounds"+index+");");
28     f.println("map.addOverlay(go" + index + ");");
29   }
30   f.println("map.addOverlay(marker" + index + ");");
31 index += 1; }}
```

Fig. 2. Map2GWT transformation

However, black-box testing approaches do not guarantee that the generated samples cover all the branches of the transformation. This calls for tools like *Pramana* to be complemented with white-box testing approaches where the unveiling of the transformation code provides additional input to obtain the test suite.

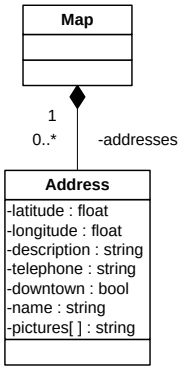

Fig. 3. Map metamodel

As an example, consider a model that is transformed to markers in Google maps (see Figure 1). Markers represent *Points of Interest (POI)*. A conference page contains the locations of the venue and the main hotels or restaurants available in the area. Those markers are captured through a *Map* metamodel (Figure 3). Transformation rules are defined to handle the two elements of the *Map* metamodel, namely, *Map* and *Address*. The output is a Google map where markers are depicted together with their pictures, if available. Besides, if the marker stands for a restaurant, the phone is shown as part of the marker's content. This last rule illustrates the need for white-box testing. The significance of 'restaurant' as a key value for changing the transformation flow cannot be ascertained from the string-typed property 'place'. Therefore, the use of metamodel-based test suite generators like *Pramana* does not preclude the need to check that all paths of the transformation have been traversed.

Analyzing the goodness of the results. In the testing literature, an oracle is a program, process or body of data that specifies the expected outcome for a set of test cases as applied to a tested object [2]. Oracles can be as simple as a manual inspection or as complex as a separate piece of software. We focus on assisting manual inspection. This requires means for linking code back to generators (i.e., MOFScript rules), and vice versa. MOFScript's native trace model provides such links at the rule level. However, a rule-based granularity might not be enough. The *address* rule (see Figure 2 - lines 14-31) illustrates how transformation complexity is tied to the complexity of the metamodel element to be handled or the logic of the transformation itself. This results in 'print' statements being intertwined along control structures such as iterators and conditionals. A rule-based granularity encloses the whole output within a single trace, failing to indicate the rule's paths being transited. A print-based granularity will account for a finer inspection of the transformation code. This in turn, can redound to the benefit of coverage analysis and code understanding. This sets the requirement for fine-grained traces.

3 The HandyMOF Tool

The previous section identifies two main requirements: semi-automatic construction of test suites, and fine-grained linkage between transformations and generated code. These requirements guide the development of *HandyMOF*, a debugger for MOFScript included as part of Eclipse (see Figure 4). The canvas of *HandyMOF* is basically divided in two areas:

- *configuration area,* where the testing scenario is defined. This includes: (1) the project folder, (2) the transformation to be debugged (obtained from the *transformation* folder in the project), and (3), the input model to be tested (obtained from the trace models that link to the chosen transformation).

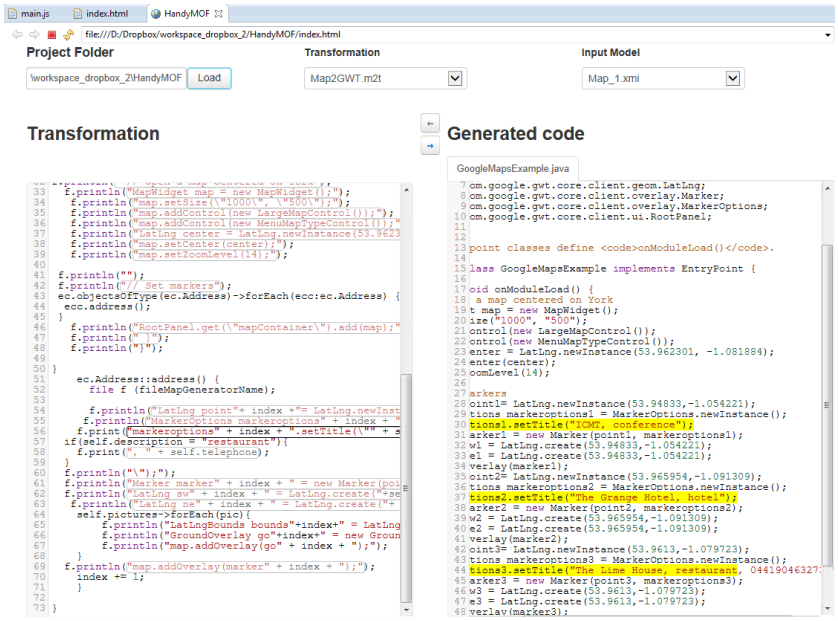

Fig. 4. *HandyMOF* as a debugger assistant: from transformation to code

– *inspection area.* Previous configuration accounts for a transformation enactment that can output one or more code files. The inspection area permits to peer at both the transformation and the code files. The output reflects a single transformation enactment (the one with the input model at hand). Figure 5 shows the case for the input model *Map_ 1.xmi*. In this case, only one code file is generated (i.e. *GoogleMapsExample.java*). Additional code files would have been rendered through additional tabs.

The added value of *HandyMOF* basically rests on two utilities. First, it permits to selectively peer at the generated code. To this end, both the transformation and the generated files are turned into hypertexts. Code is fragmented in terms of 'traceable segment' (i.e. set of characters outputted by the enactment of the same 'print', see later). Finally, both MOFScript print statements and 'traceable segments' are turned into hyperlinks. In this way, debugging answers are just a click away. Answers to questions such as 'which code does this print statement generate?' or ' which print statement caused this traceable segment?' are highlighted by just clicking on the respective hyperlink. Figures 4 and 5 illustrate two debugging scenarios:

1. Inspecting the output of a given 'print': which code snippet results from the enactment of this 'print'? Click on the print statement ('Transformation' textarea, line 56) and the answer is highlighted.
2. Tracing back a code snippet to its generator (i.e. 'print' statement), respectively. Which 'print' statement causes this code snippet? Click on the code

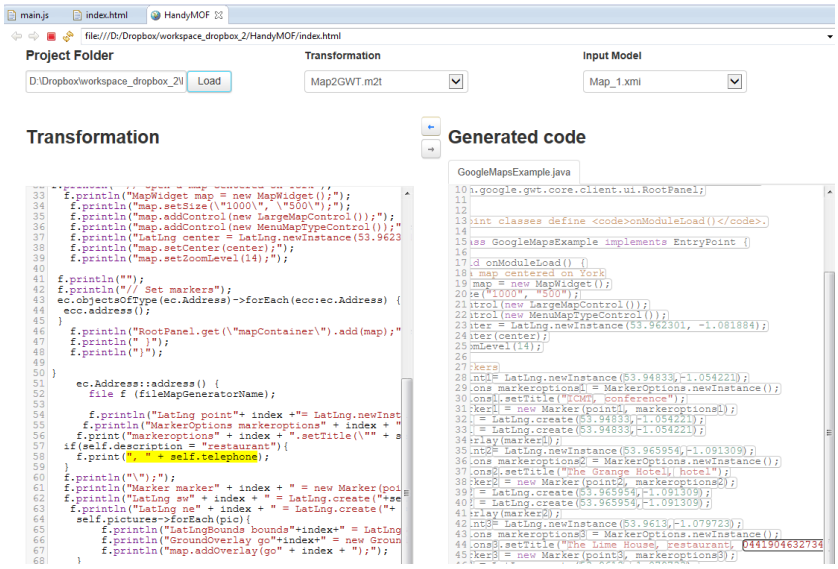

Fig. 5. *HandyMOF* as a debugger assistant: from code to transformation

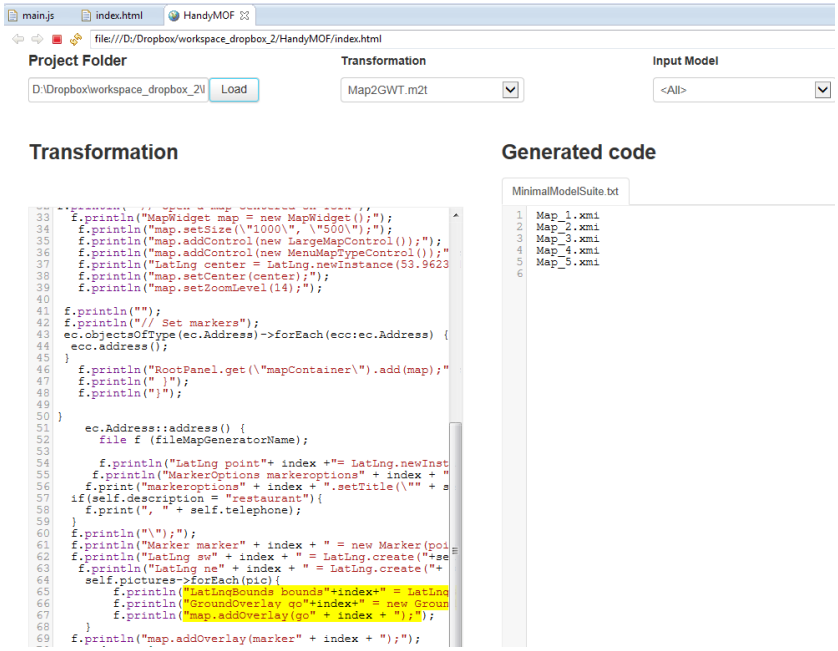

Fig. 6. *HandyMOF* as a testing assistant.

snippet ('Generated code' textarea, line 44) and the answer is highlighted ('Transformation' textarea, line 58).

The second utility is the role of *HandyMOF* as a coverage analysis assistant. First, by identifying 'holes' in the *Pramana* generated model suite in terms of 'print' statements not yet visited by any input model. Second, by identifying the smaller set of model inputs that provides the larger coverage (see later), hence coming up with a *minimal model suite* which can speed up future testing. The process starts by selecting 'all' as for the input model configuration parameter (see Figure 6). This triggers the algorithm for the obtainment of the minimal model suite. The output is reflected in two ways. First, it renders the model identifiers of such suite. Second, it aggregates the resulting trace models, collects the visited 'print' statements, and in the inspection area highlights those 'print' statements not yet transited. This helps developers to elaborate additional input models to increase transformation coverage. As can be seen in Figure 6, when <all> input models are selected, *HandyMOF* returns the minimal model suite (right) and highlights those 'print' statements not yet covered by any input model sample (left).

4 The HandyMOF Architecture

Figure 7 depicts the main components and flows of *HandyMOF*. The Project Explorer handles the folder structure. *Pramana* provides input models from the corresponding metamodel. Finally, *HandyMOF* consumes input models and transformations to obtain its own trace models, that complement MOFScript's native ones, and the generated code files.

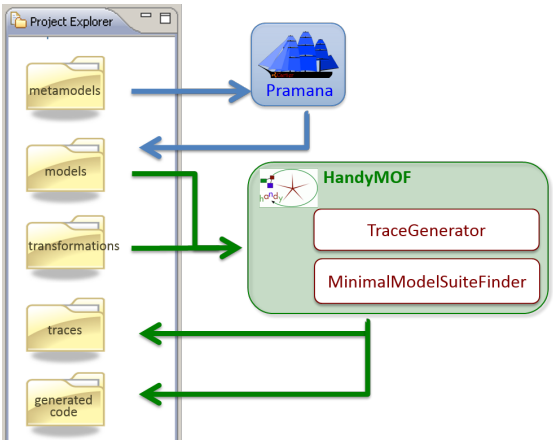

Fig. 7. *HandyMOF*'s Architecture

An important question is whether this approach can be generalized to other M2T transformation languages. Basically, *HandyMOF* rests on two main premises. First, the existence of a trace model that links the input model with the generated code. Second, the existence of a transformation metamodel (and the corresponding injector) that permits to move from the transformation text to its corresponding transformation model, and vice versa. Provided these characteristics are supported, *HandyMOF* could

be extended to languages other than MOFScript. Next subsections delve into the main components of *HandyMOF*, namely the *Trace Generator* and the *Minimal Model Suite Finder*.

4.1 Trace Generator

The goal of this component is to trace the input model, the generated code and the M2T transformation. It leverages on the trace natively provided by MOFScript that links the input model with the generated code. The metamodel for *HandyMOF*'s traces is first described, followed by how these traces are generated.

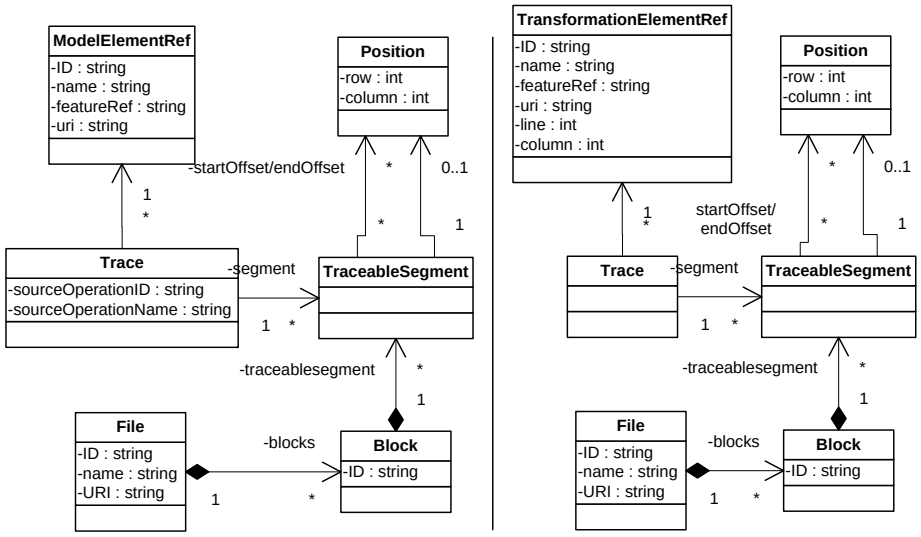

Fig. 8. MOFScript's Traceability Metamodel (left, obtained from [16]) and *HandyMOF*'s trace metamodel (right)

HandyMOF's Trace Metamodel

MOFScript's trace metamodel defines a set of concepts that enable traceability between source model elements and locations in generated text files (see Figure 8 left) [12]. A trace contains a reference to the operation (transformation rule) that generated the trace and references the originating model element and the target traceable segment. The model element reference contains the 'id' and 'name' for the originating element. It also contains a feature reference, which points out a named feature within the model element (such as 'name' for a property class).

On the other hand, the generated code file is captured in terms of 'blocks'. Blocks are identifiable units within a file. A block contains a set of segments which are relatively located within the block in terms of a starting and ending offset.

This metamodel nicely captures traces from source model elements to the generated code file through traceable segments. Unfortunately, traceable segments are related to their transformation rule counterparts rather than to the inner 'print' statements. We claim that a finer granularity might help a more accurate debugging in the presence of large transformation rules. On these grounds, we complement the natively provided MOFScript trace model with our own trace model where 'traceable segments' are linked back not just to transformation rules but to the transformation's 'print' statements. Figure 8 right depicts *HandyMOF*'s trace model. Differences stem from the granularity of traceable segments. MOFScript traceable segments account for rule enactments. In *HandyMOF*, these segments are now partitioned into fine-grained segments: one for each enacted 'print' statement. Figure 9 illustrates the two complementary traces for a simple case: between model and code (above) and between transformation and code (below). In this case, as the 'println' is composed of seven parts, seven traces will be given, one for each. As the 'print' is executed three times (one to create a location for a conference, one for an hotel and the other for a restaurant), we can see that those traces are tripled. The position of the 'print' in the transformation to be the same, as captured in *TransformationModelElement*.

Obtaining Trace Models in *HandyMOF*

The process starts by generating the test model suite, in our case this is achieved using *Pramana*. Once the model suite is obtained the next step is to link the M2T transformation with the code that is generated from these models. The first obstacle rests on the generated code being plain text, so that the trace model links the transformation elements with the position where the related code fragment starts (see Figure 8 right). This position can be different depending on the input model and depends on the execution flow. As a case in point, imagine an *if-then-else* statement in the transformation. Each branch may have a different number of 'print' statements. As a consequence, the position where the first statement after the 'if' starts may vary depending on the executed branch. The same holds for loops, depending on the input model they may be executed a different number of times thus changing the position where the rest of the statements start.

So additional information is required for a particular model, e.g. whether a conditional instruction is true or false, or the number of iterations, to know which specific statements have been executed and how many times. This data is collected in a tracing file.

Transformations are also models and can thus be analyzed or be the input of another transformation. Therefore, in this proposal, the original M2T transformation will be used to get internal information of its execution, and save it in the execution trace model. More specifically, taking the original M2T transformation as input, a *Higher Order Transformation (HOT)* transformation will

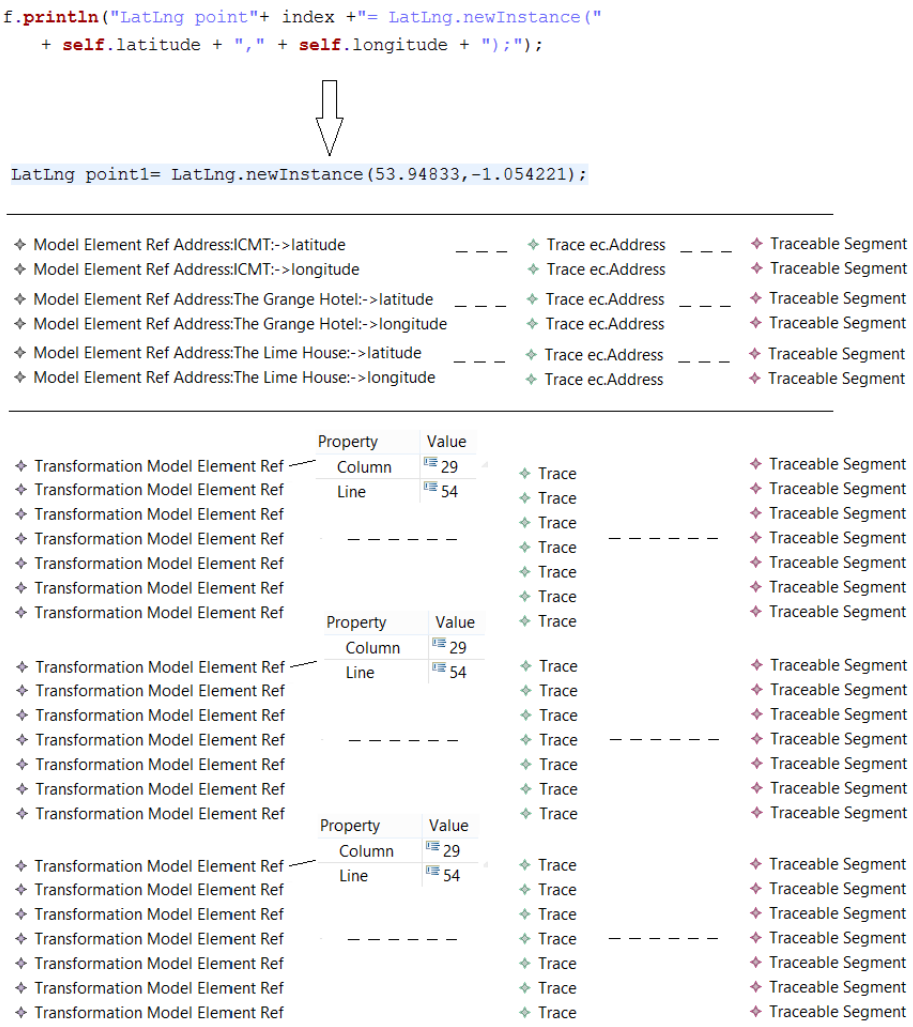

Fig. 9. Complementary trace model: between model and code (above) and between transformation and code (below)

modify it, e.g. inserting counter variables in each iterator and flags to mark conditional instructions. As a result, this leveraged transformation not only outputs the code but also the *execution tracing model*.

That execution tracing model, and the trace between the input model and the generated code, along with the original M2T transformation, are used to get the trace model between the M2T transformation and the code corresponding to each input model. An ATL M2M transformation is in charge of this trace generation, calculating the length of each 'print' from the transformation to set the offset values in the 'Segment' elements; and having into account how many times each 'print' is executed.

4.2 The Minimal Model Suite Finder

In order to analyze the M2T transformation and to see to what extent its statements have participated in the code generation, the use of input models is unavoidable. The goal is to get the input models that obtain a 100% coverage of the transformation code. However, to the best of our knowledge no tool exists that, given an input domain metamodel and a M2T transformation, generates the models that provide full coverage of the transformation. As a result, we opted for using *Pramana* (formerly known as *Cartier*) [16], a tool that implements black-box testing for metamodels [16].

Are models generated by *Pramana* enough to obtain our goal? *Pramana* serves engineers by generating *model suites* for *metamodel coverage* but its purpose is not *transformation coverage*. However, transformations have embedded semantics that need to be considered if the goal is the latter. Different conditions present in if statements or loops require specific test cases that may not be generated if the criteria is merely metamodel coverage. As a case in point, the if statement in Figure 2 (line 18) checks whether the *Address* corresponds to a restaurant. Among the many test cases that can be generated from the metamodel, this statement requires one with precisely that value in the *description* attribute to obtain *transformation coverage,* which is not guaranteed if the generation of the test cases does not take the transformation into account. Hence, as in program testing where black-box testing and white-box testing approaches are used in concert, we need to cater for both metamodel and transformation coverage.

The proposal of this work is the use of trace models for the analysis of *transformation coverage*. What is needed is to link the code samples with the transformation, via the tracing models obtained by the trace generator module. We need to see how much coverage has been reached using the input models generated by *Pramana*.

Hence, the task of the *MinimalModelSuiteFinder* module (see Figure 7) is to quantify the transformation coverage, and to rule out those input models whose transformation only enacts transformation statements that have already been traversed by previous models. The goal of the module is then to minimize set of input models and obtain the higher coverage percentage of the transformation code (specifically, the 'print' instructions that generate the target code). We

name this set the *minimal model suite*. While not optimal, the presented greedy algorithm permits to reduce the test suite size.

```
1 helper def : getModels (availableModels : Sequence(Trace!TraceModel),
2   minimalModelSuite : Sequence(Trace!TraceModel),
3   coveredPrints : Sequence(String)) : Sequence(Trace!TraceModel) =
4 minimalModelSuite->append(availableModels->select(e|self.bestModel(e, availableModels, coveredPrints))->first()),
5 coveredPrints->append(availableModels->select(e|self.bestModel(e, availableModels, coveredPrints))
6     ->first().trace ->collect(e|e.line))->flatten())
7  if coveredPrints.size() = self.numberOfLines or availableModels.size()=0 then
8    minimalModelSuite
9  else
10   self.getModels(availableModels->excluding(availableModels->select(e|self.bestModel(e, availableModels,
11   coveredPrints))->first())
12 endif ;
```

Fig. 10. Minimal model suite algorithm (main rule)

Figure 10 shows one of the functions of the algorithm used in obtaining this suite. It is a recursive function that finishes when all lines are covered or there are not more input models to use (line 7). The algorithm can be summarized as follows:

1. The best model is added to the list of selected models (*minimalModelSuite*) (line 4).
2. The prints covered by the best model are added to the list of covered prints (*coveredPrints*) (lines 5-6).
3. The best model (i.e., the one that covers most prints) is excluded from the available models (*availableModels*) (lines 10-11).

Using these two modules the interface of *HandyMOF* can be used to check the correspondence between the M2T transformation and the generated code, be it on a single instance (see Figure 5) or for the complete model suite to check the obtained coverage (see Figure 6).

5 Related work

This work sits inbetween testing and traceability for M2T transformations. Testing wise, no standard or well established proposal exists for M2T transformation testing [17]. Wimmer et al. present an extension of *tracts* [7] to deal with model-to-text transformations [18]. Their approach is complementary to ours as it focuses on black-box testing (i.e., it considers the specification of the transformation, not its implementation). Our work highlights the complementariness of black-box and white-box testing techniques. Black-box testing approaches do not capture the mechanics of the transformation [1], which is precisely where we intend to aid. McQuillan et al. propose white-box coverage criteria for transformations [11]. Although their work centers in ATL [9] (i.e., a model-to-model transformation), their coverage criteria could be applicable to our case as well.

We focus on instruction coverage (more precisely on coverage of instructions that produce an output in the generated code). Gonzalez et al. present a white-box testing approach for ATL transformations [8]. It follows a traditional white-box testing strategy where input models are created based on the inner structure of the transformation. This involves a coupling between the approach and the transformation language. This might not be problem for M2M transformation languages (where ATL has become *de facto* standard) but rises portability issues for M2T transformations where no predominant language exists. This is why we opt for a mixed approach where input models are generated using black-box testing on the search for transformation-language independence. This approach, albeit less precise, can be applied to any language provided adequate traces can be obtained. This moves us to traceability.

	ModelTo Code	TransfTo Code	TransfTo Model	Mechanism	Aim
Acceleo	element to block	rule to code	rule to element	---	transf./code sync. & model/code sync.
Epsilon (EGL)	---	rule to block	---	API	transf. auditing & model/code sync.
Mof2Text	element to block	---	---	---	---
MOFScript	element to block	rule to block	---	Trace mm	model/code sync.
xTend	element to code	---	---	---	---

Fig. 11. Traceability comparison.

Table of Figure 11 compares main M2T tools and their traceability support. Values are obtained from the literature or grasped from videos or forums. Comparison is set in terms of trace availability for model-to-code, transformation-to-code and transformation-to-model. The underlying mechanisms and the pursued aim is also included. Within the model-to-code and transformation-to-code options, a *'block'* is nothing more than a piece of code, i.e. an identifiable unit within a file. In these proposals, code blocks to generate and to be traced must be delimited by special keywords in the transformation. When *'code'* is indicated in the table, there is a traceability but no information about the underlying details.

Mof2Text specification (i.e. the OMG standard for MOF M2T Transformation Language) [13] provides support for tracing model elements to text parts. Specifically, a trace block relates text that is produced in a block to a set of model elements. Some text parts may be marked as *protected* in order to be preserved and not overwritten by subsequent M2T transformations. *MOFScript* implements that proposal and handles the traceability between generated text and the original model, aiming to be able to synchronize the text in response to

model changes and vice versa. *MOFScript* does not specify any language-specific mechanisms to support traceability, but a metamodel manages the traces from a source model to target generated text files. Central in this trace model is the logical segmentation of a file into blocks; thus, a trace contains a reference to the transformation rule that generated the trace and references to the originating model element and to the target traceable segment.

Another implementation of OMG's M2T specification is the *Acceleo* code generator. Acceleo Pro Traceability [3], a tool complementary to the generator, enables round trip-support: updates in the model or the code are reflected in the connected artefacts. Since this is a commercial tool, restricted information describing the solution is available.

Epsilon [4] is a platform for model management where several task-specific languages are integrated, among them one is *Epsilon Generation Language* (*EGL*), which is a template-based code generator, i.e. their proposal for M2T transformations. EGL provides a traceability API that facilitates exploration of the executed templates, affected files and protected regions that are processed during a transformation. Like previous work, this tool does not have support for transformation coverage either.

As far as we know, the Xtend language does not create traces automatically. And last but not least, JET [5], Velocity [6], and StringTemplate [7] are other M2T languages, that with JSP-like or Java-based templates render source code including java, HTML, XML, SQL, and so on. No information has been found about traceability in these platforms.

6 Conclusions

This work presented a proposal for white-box testing of M2T transformations. Due to the heterogeneity of M2T transformation languages, the test suite is generated using black-box testing and then, the generated code is traced back to the transformation and the input model. Main outcomes include: (1) if a bug is detected in the generated code, it can be traced back to the generating 'print' statement, (2) each generator statement (i.e., 'print') can be traced to the generated code line, and (3) the transformation coverage obtained by the test model suite can be calculated in terms of visited 'prints'. If the obtained coverage is not complete, the developer can create input models that cover the missing transformation lines. This is realized in *HandyMOF*, a tool for debugging MOFScript transformations.

This proposal could be generalized for any transformation language fulfilling our both premises, namely, the existence of a transformation metamodel (and its injector) and a trace model linking the input model with the generated code.

[3] http://www.obeo.fr/pages/obeo-traceability/en
[4] https://www.eclipse.org/epsilon/
[5] https://www.eclipse.org/modeling/m2t/?project=jet
[6] http://veloedit.sourceforge.net/
[7] http://sourceforge.net/projects/hastee/

The part of the tool that would need to be reimplemented in case of exporting the idea to other languages is the *trace generation* module, that would have to be adapted to language structures of the new transformation language. Both the interface and the coverage analysis are reusable.

Future work includes guiding transformation developers in creating the missing input models from the unvisited 'prints'. We also contemplate integrating *HandyMOF* with other testing approaches to provide an integrated solution.

Acknowledgments.

This work is co-supported by the Spanish Ministry of Education, and the European Social Fund under contract TIN2011-23839. Jokin enjoyed a grant from the Basque Government under the "Researchers Training Program". We thank Cristóbal Arellano for his help developing *HandyMOF*, and the reviewers for their comments.

References

1. Benoit Baudry, Sudipto Ghosh, Franck Fleurey, Robert B. France, Yves Le Traon, and Jean-Marie Mottu. Barriers to Systematic Model Transformation Testing. *Communications of the ACM*, 53(6):139–143, 2010.
2. Boris Bezier. *Software Testing Techniques.* New York : Van Nostrand Reinhold, 1990.
3. Erwan Brottier, Franck Fleurey, Jim Steel, Benoit Baudry, and Yves Le Traon. Metamodel-based Test Generation for Model Transformations: an Algorithm and a Tool. In *17th International Symposium on Software Reliability Engineering (ISSRE 2006),Raleigh, USA*, 2006.
4. Juan José Cadavid, Benoit Baudry, and Houari A. Sahraoui. Searching the Boundaries of a Modeling Space to Test Metamodels. In *5th International Conference on Software Testing, Verification and Validation (ICST 2012), Montreal, Canada*, 2012.
5. Franck Fleurey, Benoit Baudry, Pierre-Alain Muller, and Yves Le Traon. Qualifying Input Test Data for Model Transformations. *Software and System Modeling (SoSyM)*, 8(2):185–203, 2009.
6. Frank Fleurey, Jim Steel, and Benoit Baudry. Validation in Model-driven Engineering: Testing Model Transformations. In *1st International Workshop on Model, Design and Validation (SIVOES-MoDeVa 2004), Rennes, France*, 2004.
7. Martin Gogolla and Antonio Vallecillo. *Tract*able Model Transformation Testing. In *7th European Conference on Modelling Foundations and Applications (ECMFA 2011), Birmingham, UK*, 2011.
8. Carlos A. González and Jordi Cabot. ATLTest: A White-Box Test Generation Approach for ATL Transformations. In *15th International Conference Model Driven Engineering Languages and Systems (MODELS 2012), Innsbruck, Austria*, 2012.
9. Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A Model Transformation Tool. *Science of Computer Programming (SCP)*, 72(1-2):31–39, 2008.

10. Jochen M. Kuster and Mohamed Abd-El-Razik. Validation of Model Transformations - First Experiences using a White Box Approach. In *3rd International Workshop on Model Development, Validation and Verification (MoDeVa 2006), Genova, Italy*, 2006.
11. Jacqueline A. McQuillan and James F. Power. White-Box Coverage Criteria for Model Transformations. In *1st International Workshop on Model Transformation with ATL (MtATL 2009), Nantes, France*, 2009.
12. Gøran K. Olsen and Jon Oldevik. Scenarios of Traceability in Model to Text Transformations. In *3rd European Conference on Model Driven Architecture-Foundations and Applications (ECMDA-FA 2007), Haifa, Israel*, 2007.
13. OMG. MOF Model to Text Transformation Language, v1.0. Formal Specification, January 2008. Online at: http://www.omg.org/spec/MOFM2T/1.0/PDF.
14. OMG. Query/View/Transformation, v1.1. Formal Specification, January 2011. Online at: http://www.omg.org/spec/QVT/1.1/PDF/.
15. Gehan M. K. Selim, James R. Cordy, and Juergen Dingel. Model Transformation Testing: The State of the Art. In *1st Workshop on the Analysis of Model Transformations (AMT 2012), Innsbruck, Austria*, 2012.
16. Sagar Sen, Benoit Baudry, and Jean-Marie Mottu. On Combining Multi-formalism Knowledge to Select Models for Model Transformation Testing. In *1st International Conference on Software Testing, Verification, and Validation (ICST 2008), Lillehammer, Norway*, 2008.
17. Alessandro Tiso, Gianna Reggio, and Maurizio Leotta. Early Experiences on Model Transformation Testing. In *1st Workshop on the Analysis of Model Transformations (AMT 2012), Innsbruck, Austria*, 2012.
18. Manuel Wimmer and Loli Burgueño. Testing M2T/T2M Transformations. In *16th International Conference on Model-Driven Engineering Languages and Systems (MoDELS 2013), Miami, USA*, 2013.