White-Box Coverage Criteria for Model Transformations

Jacqueline A. McQuillan and James F. Power

Department of Computer Science,
National University of Ireland, Maynooth,
Co. Kildare, Ireland
{jpower@cs.nuim.ie}
http://www.cs.nuim.ie/research/pop/

Abstract. Model transformations are core to MDE, and one of the key aspects for all model transformations is that they are validated. In this paper we develop an approach to testing model transformations based on white-box coverage measures of the transformations. To demonstrate the use of this approach we apply it to some examples from the ATL metamodel zoo.

Keywords ATL, model transformations, software testing, coverage criteria, metamodels.

1 Introduction

Modelling is concerned with the construction and maintenance of models, but also the transformation from one model to another in the context of Model Driven Engineering (MDE). Model transformations are core to MDE and, similar to conventional software, it is vital to validate the transformations and ensure they are correct. One possible validation method is to systematically test the transformation. While much research has been conducted into transformation languages and tool support for conducting transformations, relatively little attention has focused on testing the model transformations.

In this paper we address the problem of testing model transformations by examining *white-box coverage measures* for model transformations. We report on some initial experiments of applying this approach to some transformations from the ATL transformation zoo.

The remainder of this paper is organised as follows. Section 2 outlines some of the background information related to testing model transformations. In Section 3 we consider what kind of white-box coverage measures can be derived from ATL transformations. In Sections 4 and 5 we illustrate the use of our test criteria using two model transformation. Finally, Section 6 concludes the paper and discusses future work.

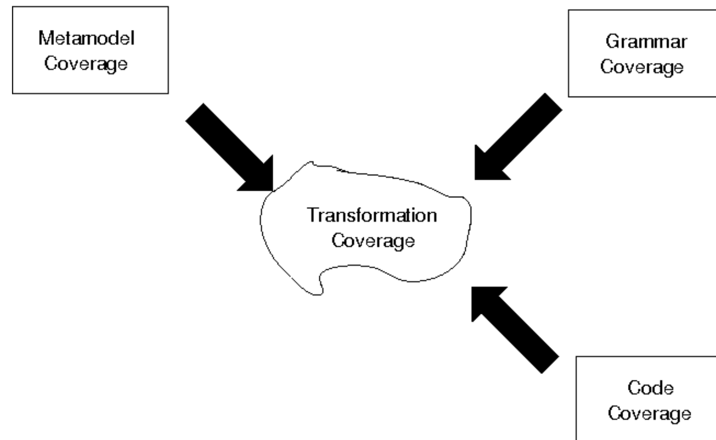


Fig. 1. Coverage measures for model transformations might be derived from metamodel coverage measures, or from research on grammar coverage, or by analogy with code coverage.

2 Background and Related Work

Research relevant to testing model transformations, particularly from a coverage perspective, may be divided into three main strands, as shown in Figure 1 and discussed in the following subsections.

2.1 Coverage of the input metamodel

The most obvious source for coverage data is gained from considering the degree to which the model transformations cover the input metamodel. In standard testing terminology this corresponds to testing using different inputs, most typically characterised as *black box* testing since it does not presuppose access to the transformation source code.

A range of coverage criteria have been suggested for the various UML diagrams [1]. Since a metamodel can be described using a UML class diagram, coverage criteria for class diagrams provide a basis for developing similar criteria for metamodels. For example, Andrews *et al.* define a number of coverage measures for class diagrams [2]. A parallel stream of research investigates the generation of test instances of models and metamodels. For example, Gogolla *et al.* [3] describe an approach to automatically generating model instances (snapshots) from UML class diagrams. Another approach is that of Ehrig *et al.* [4] which involves the automatic creation of an instance-generating graph grammar for the given metamodel. However, this work does not consider how to evaluate the adequacy of these test cases during the testing process.

The most important work related to coverage analyses of model transformations is that of Fleurey *et al.* [5–7]. Starting with the model coverage measures

of Andrews *et al.*, they extend these to the source metamodel for model transformations. They also use these coverage criteria to generate test cases for model transformation testing. Since they focus on generating inputs for the transformation, this approach may be characterised as *black-box* testing.

An important contribution of the work by Fleurey *et al.* is the identification of the *effective metamodel*. This is the section of the source metamodel, often a proper subset, that is relevant to the transformation. Clearly it is important that coverage data be calculated in terms of this effective metamodel, rather than the whole input metamodel, so as not to underestimate the level of coverage.

2.2 Grammar coverage

The use of measures based on input models relies on generating a test suite that adequately covers the input domain of a transformation, as defined by the input metamodel, or a relevant subset. However, there is little work on directly considering the coverage of the transformations themselves. One related area is that of grammar testing, since the process of transforming an input language using a grammar (or a generated parser) is analogous to a model transformation. Various coverage criteria have been proposed, the most simple being rule coverage, which requires that each rule in the grammar be used during testing, although there are many more complex variations [8, 9].

A model transformation consists of more than just rules to match the input, and so any consideration of coverage should also deal with model generation and any internal operations. To date there has been relatively little work on linking coverage of the "front end", as defined by a grammar or the input metamodel, with coverage of the "back end" as defined by transformation internals and generation code. Hennessy and Power show that applying test suite reduction using only grammar coverage as a criteria yielded poor results for the internals of a C++ parser [10], and thus would suggest that coverage of transformation internals should be considered further.

2.3 Code coverage

One of the standard ways of determining the adequacy of a test suite is to determine the degree to which the test suite exercises the system under test using a coverage analysis [11]. For conventional programming languages the degree of coverage of elements such as statements, decisions, paths, functions etc. can be calculated for a test suite, with the goal of achieving 100% coverage of the chosen element. Given the widespread use and acceptance of such measures in the programming domain, it is natural to consider their use for modelling and model transformation.

Since metamodels can be implemented in program code, often automatically, applying coverage measures to this generated code provides one means of measuring metamodel coverage. This approach was taken in our earlier work on the measurement metamodel where line and branch coverage were used to evaluate coverage for an implementation generated by the Octopus tool [12]. While this

approach has the advantage of simplicity, it is rather indirect, and depends to some degree on decisions taken by the code generator.

3 White-Box Coverage Measures for Model Transformations

In order to calculate the coverage of the ATL rules during the transformation it is necessary to profile the operation of each ATL rule as the transformation takes place. Fortunately the design of the ATL system provides two useful features that facilitate this. First, the compiled ATL rules are actually executed on top of a special-purpose virtual machine [13]. Second, it is possible to run the ATL system in debug mode which prints out the step-by-step execution of instructions on this virtual machine.

The ATL virtual machine is similar in concept to the Java Virtual Machine (JVM) which greatly eases comprehension. It has instructions to access and create model elements, to manipulate data on the stack, and control instructions for selection, iteration and method calls.

3.1 Implementation

Thus, to measure coverage of ATL transformations we implemented a program that works in two phases:

1. First, we process the file of compiled ATL instructions (conveniently represented in XML format) to extract information about the operations, instructions and branch locations and targets.
2. Second, we run the transformation and process the resulting log file to record the actual coverage data for that transformation.

3.2 Relevant structures in the compiled ATL file

We can partition the code generated by the ATL compiler into three main categories:

1. **Scaffolding code**, such as internal routines to initiate the matching process and help with resolving references. This includes generated functions such as `main`, `__matcher__`, `__exec__`, `__resolve__` and `resolveTemp`.

2. **Code corresponding to the rules**, which is broken into two separate functions in the assembled code. For any user-defined rule R, the ATL compiler will generate a function `_match`R to handle the filtering of model elements, and a function `_apply`R to handle the instantiation of the target elements, assuming a successful match.

3. **Code corresponding to helpers**, which can be further subdivided into code for attribute helper initialisation, and code for operation helpers. Attribute helper initialisers will always be invoked by the internal ATL routines but, since they may contain conditional expressions, there is still possibility that they will not be fully covered for a test case. Operation helpers on the other hand must be invoked by the user's code, and thus there is a possibility that they may not be used at all for a given test case.

Since scaffolding code functions are largely transparent to the user, we do not consider them further in this study.

3.3 Possible coverage measures

Based on the structure of the compiled ATL file, we can immediately identify three kinds of coverage measures:

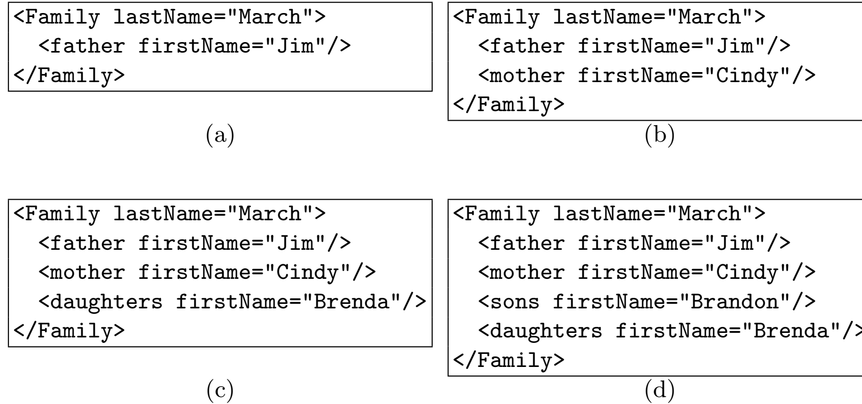
Rule Coverage is analogous to rule coverage in a grammar: it is simply the percentage of rules that were executed at least once during a transformation. Since each rule is represented as an operation on the ATL virtual machine, implementing rule coverage involves tracking and recording the calls to the operation corresponding to each rule.

Instruction Coverage is analogous to code coverage in a high-level language, with the additional benefit that formatting and layout do not effect the totals. The instruction coverage for a set of transformations is the percentage of instructions that were executed at least once during the transformation. The debug trace for the ATL virtual machine lists each instruction as it is executed, so it is relatively straightforward to measure this coverage.

Decision Coverage measures, for each decision in a program, whether the true and false paths were taken. In ATL transformations, branches are represented by IF instructions, and whether they evaluated to true or false can be determined from the trace file.

While *instruction coverage* corresponds to the most common kind of code coverage, it is not obvious that such a low-level measure is useful in the context of ATL transformation. First, it has the disadvantage of being linked directly to the ATL compiler and low-level VM implementation. Second, it is a measure more suited to an "imperative" style of programming, and can be difficult to relate back to ATL source code, which often contains large nested expressions.

In the remainder of this paper we focus on *decision coverage*, since this is relatively easy to relate to the ATL source code, and is not so directly tied to the underlying implementation. Since ATL transformation rules are implemented as functions in the compiled code, decision coverage will effectively subsume rule coverage which becomes a special case.



```
<Family lastName="March">
  <father firstName="Jim"/>
</Family>
```

(a)

```
<Family lastName="March">
  <father firstName="Jim"/>
  <mother firstName="Cindy"/>
</Family>
```

(b)

```
<Family lastName="March">
  <father firstName="Jim"/>
  <mother firstName="Cindy"/>
  <daughters firstName="Brenda"/>
</Family>
```

(c)

```
<Family lastName="March">
  <father firstName="Jim"/>
  <mother firstName="Cindy"/>
  <sons firstName="Brandon"/>
  <daughters firstName="Brenda"/>
</Family>
```

(d)

Fig. 2. Four simple possible input models, in increasing order of size, for the *Families2Persons* transformation given in Figure 3. These examples are based on the example distributed with the *Families2Persons* transformation.

4 A Simple Example: The Families2Persons transformation

As an example, consider the *Families2Persons* ATL transformation shown in Figure 3 at the end of this paper, taken directly from the ATL examples [14].

As well as the scaffolding code, the ATL compiler will generate the following methods:

Code for attribute helpers: The generated function `__initfamilyName` initialises the attribute helper `familyName` (lines 3-15)

Code for function helpers: `isFemale` is generated corresponding to the function helper of the same name (lines 16-25)

Code for Member2Male: Two functions called `__matchMember2Male` and `__applyMember2Male` are generated and correspond to lines 28 and 30-32 respectively of the rule `Member2Male`

Code for Member2Female: Again, two generated functions `__matchMember2Female` and `__applyMember2Female` corresponding to lines 36 and 38-19 respectively of the rule `Member2Female`

4.1 Deriving coverage data

Even with this simple example, it quickly becomes apparent that it is not trivial to define undisputed coverage measures. The logical place to seek a definition of the *effective metamodel* is in the two transformation rules, and this would appear to indicate that the source model element `Families!Member` should be covered. Thus a rather naive attempt might be the minimal family shown in Figure 2(a).

	No. of Calls			
Family:	Fig 2(a)	Fig 2(b)	Fig 2(c)	Fig 2(d)
`__initfamilyName`	1	2	3	4
`isFemale`	2	4	6	8
`__matchMember2Male`	1	1	1	1
`__applyMember2Male`	1	1	1	2
`__matchMember2Female`	1	1	1	1
`__applyMember2Female`	0	1	2	2

Table 1. The number of calls of each of the functions in the *Families2Persons* transformation for each of the four possible inputs in Figure 2. This provides a crude measure of the level of cover, but fails to adequately distinguish between the test cases.

Of course, it is clear that there is a rule each for female and male family members, so logically a better input model would contain at least one instance of each, such as shown in Figure 2(b). A quick analysis shows that all six generated functions are executed at least once for this test case: the number of calls to each function are shown in the data columns of Table 1. However, this simple statement masks some potential complexity. The filter that decides which rule is selected is defined in the helper operation `isFemale`, so, technically, constructing the effective metamodel correctly is contingent on being able to interpret this helper fully.

An analysis of the *decision coverage* data for the six functions is shown in Table 2. From this it can be seen that the test case of Figure 2(b) still covers only 75% of the decisions of the `isFemale` function, and even less of the initialiser for `familyName`. Adding a daughter to the family, as shown in Figure 2(c) completes the coverage for `isFemale`. Similarly adding a son `familyName`, as shown in Figure 2(d) completes the coverage for `familyName`. Comparing this data with that shown in Table 1 clearly shows that decision coverage is delivering a more complete picture than just counting the number of calls for each function.

	Decision Coverage			
Family:	Fig 2(a)	Fig 2(b)	Fig 2(c)	Fig 2(d)
`__initfamilyName`	17%	50%	83%	100%
`isFemale`	50%	75%	100%	100%
`__matchMember2Male`	50%	100%	100%	100%
`__applyMember2Male`	0/0	0/0	0/0	0/0
`__matchMember2Female`	50%	100%	100%	100%
`__applyMember2Female`	0/0	0/0	0/0	0/0

Table 2. The decision coverage percentage for each of the functions in the *Families2Persons* transformation for each of the four possible inputs in Figure 2. These data allow for a greater level of distinction between test inputs than the measures in Table 1.

4.2 Discussion

Even this simple example shows some of the difficulties involved in identifying the *effective metamodel* for a transformation. Since the code for `isFemale` could be substituted into the rule definitions, it should clearly be considered relevant to determining the effective metamodel. However, the defined attribute `familyName` is also relevant to the discussion, since it is defined entirely over the input metamodel, and thus a test suite would need to ensure that all of the possible permutations are exercised. Indeed, in this example there is very little of the code that *is not* relevant to fully defining the effective metamodel.

Even considering decision coverage, as above, does not quite give a full picture of metamodel coverage. For example, the coverage data for `isFemale` is the amalgamated coverage for each call to the function. In theory, a fuller picture would be given by considering the *context* of the call, so that we could distinguish between `isFemale` as used by `Member2Male` and as used by `Member2Female`. Only the example shown in Figure 2(d) exercises all these options, but this is not shown in the context-independent data of Table 2. There is an obvious potential for combinatorial explosion here, and further research would be required to see if adding context was justified in terms of the improvement in test suite analysis.

5 A larger study

In this section we apply the coverage measures to a larger example, the *UML2 to Measure* transformation from the ATL Transformations zoo [15]. This transformation calculates a set of metrics for UML class diagrams. As such, it has a well-known source metamodel, so plenty of test cases are available. It also has a computationally-intensive back-end that calculates 51 metrics over the input UML class diagram. It thus represents an extreme example of a transformation where the exact effective metamodel is not easily deducible.

The *UML2 to Measure* is composed of four modules: one main UML22Measure module, and three modules that calculate metrics called MOOD4UML2, EMOOSE4UML2 and QMOOD4UML2. In what follows, we abbreviate these as U2M, MOOD, EMOOSE and QMOOD respectively.

5.1 A test suite of UML class diagrams

The test cases used in our study were taken from the Eclipse UML2 Tools project [16]. This project includes, among other examples, 19 UML class diagrams from chapter 7 of the UML Superstructure Specification. The 19 class diagrams are described briefly in Table 3, mainly to provide reference to the original source. While there was no coverage data or analysis provided with these models, they were selected as they presumably covered a wide range of features in class diagrams.

The relevant coverage measures were calculated for each of the 19 test cases individually, and then calculated for the test suite as a whole.

Test Case	Description used in UML2 Tools project
7.19	Graphic notation indicating exactly one association end owned by the association
7.20	Combining line path graphics
7.21	Binary and ternary associations
7.22	Association ends with various adornments
7.23	Examples of navigable ends
7.24	Example of attribute notation for navigable end owned by an end class
7.25	Derived supersets (union)
7.26	Composite aggregation is depicted as a black diamond
7.27a	An AssociationClass is depicted by an association symbol (a line) and a class symbol (a box)
7.27b	Association Class
7.28	Class notation - details suppressed, analysis-level details, implementation-level details
7.30	Examples of attributes
7.32	Comment notation
7.33	Constraint attached to an attribute
7.34	{xor} constraint
7.39	Example of element import
7.40	Example of element import with aliasing
7.48	Multiple ways of dividing subtypes (generalization sets) and constraint examples
7.54	Instance specifications representing two objects connected by a link

Table 3. A summary of the "Chapter 7" class diagrams from the UML2 Tools project (release 0.8.1 (2008/09/23) [16]. In future tables we refer to these models by number only; this table can be used to refer back to the models in the UML2 distribution.

5.2 Decision coverage results

Since there are 75 IF instructions, this makes a total of 150 possible decisions, and over half of these (88) in the U2M module. The results from the coverage analysis are summarised in Table 4 on a per-model basis. This table has one row for each of the UML models described previously in Table 3. The data in each row represent the fraction of decisions covered for each module in the ATL transformation, summed over all functions in that module.

For example, the value "28/88" in the first column of the first data row of Table 4 measures the decision coverage for the U2M ATL module when run with class diagram #19 as input. There were 44 IF instructions in the module, so the total possible decision coverage would have been 88. In this case, only 28 of the possible true/false decisions were taken.

As can be seen from the data in Table 4, the test cases are relatively similar in terms of their module-by-module coverage. This possibly reflects the nature of the transformation itself: since all metrics are calculated for each test case, the level of coverage is quite similar for each. It is notable that decision coverage

UML Model	Proportion of Decisions Covered				
	U2M	MOOD	EMOOSE	QMOOD	Total
19	28/88	12/24	1/8	15/30	56/150
20	28/88	12/24	1/8	15/30	56/150
21	27/88	12/24	1/8	15/30	55/150
22	31/88	12/24	3/8	18/30	64/150
23	28/88	12/24	1/8	15/30	56/150
24	28/88	12/24	1/8	15/30	56/150
25	26/88	12/24	3/8	15/30	56/150
26	22/88	12/24	1/8	12/30	47/150
27a	22/88	12/24	1/8	12/30	47/150
27b	27/88	12/24	1/8	16/30	56/150
28	36/88	12/24	1/8	19/30	68/150
30	44/88	12/24	3/8	22/30	81/150
32	20/88	10/24	1/8	12/30	43/150
33	25/88	10/24	1/8	13/30	49/150
34	22/88	12/24	1/8	12/30	47/150
39	24/88	10/24	1/8	12/30	47/150
40	30/88	10/24	1/8	14/30	55/150
48	26/88	12/24	3/8	15/30	56/150
54	27/88	10/24	1/8	14/30	52/150
Cum. Tot.	55/88	24/24	3/8	25/30	107/150

Table 4. The decision coverage of each of the four modules for each of the UML models of Table 3. The coverage is given as a fraction of the total number of decisions in the helper functions.

for any individual test case rarely exceeds 50% in a module, and is particularly low for the EMOOSE module.

The final row of Table 4 shows the *cumulative* decision coverage when all 19 UML models were transformed, and thus represents the total coverage for all models considered as a test suite. This is important to consider since, even though the individual totals are similar, the decisions being covered may not be the same in each case. For example, all of the test cases cover either 10 or 12 of the 24 decisions in the MOOD module, but the cumulative total of 24 shows that these must be *different* decisions in at least some of the cases.

The overall coverage of 107/150, or 71% seems quite a reasonable level of coverage for a suite that was not designed with such coverage in mind. Nonetheless, it would clearly be advisable to augment the suite to achieve full coverage.

5.3 Detailed analysis of decision evaluation

As a further example of the type of information available from decision coverage analysis, Table 5 analysis the 75 decisions in the four modules for the test cases. This table has one row for each test case, and shows the number of IF statements that were never evaluated, that evaluated just to false or true respectively, or

UML Model	No. of if statements				
	Neither	False	True	Both	Total
19	26	24	18	7	75
20	26	24	18	7	75
21	27	24	17	7	75
22	23	25	15	12	75
23	26	24	18	7	75
24	26	24	18	7	75
25	29	17	19	10	75
26	33	16	21	5	75
27a	33	16	21	5	75
27b	25	23	21	6	75
28	21	30	10	14	75
30	14	33	8	20	75
32	35	13	24	3	75
33	30	21	20	4	75
34	32	15	24	4	75
39	34	11	24	6	75
40	28	17	22	8	75
48	28	16	22	9	75
54	27	21	23	4	75
Cum. Total	12	13	6	44	75

Table 5. A breakdown of the overall cumulative decision coverage data for all 19 UML models. This table splits the decision instructions into four categories based on the degree to which they were covered during the transformations.

that evaluated to both false and true. This gives a deeper insight as to the overall cause of low coverage, since this could be due either to decisions not being covered at all, or not being evaluated to all possibilities.

For example, the first data row of Table 5 shows the coverage details for class diagram #19. From this we can see that 26 of the decisions in the ATL transformation were never executed, 24 were executed and only ever evaluated to FALSE, 18 only ever evaluated to TRUE, and just 7 were fully tested, being evaluated to both FALSE and TRUE during the transformation. For comparison with Table 4, we can calculate the total decision coverage for this test case as $24 + 18 + (7 * 2) = 56$, as shown in the final column of the first row of Table 4.

For individual test cases, relatively few of the IF statements evaluate to both true and false: only three test cases cause more than 10 of the 75 IF statements to be fully evaluated. Happily, the final row of Table 5 shows that in total 44 of the 75 IF statements are fully evaluated, and efforts to augment the test suite need only concentrate on the remaining 31 statements.

5.4 Comparison with instruction coverage

The ATL transformation contains 151 functions in total (excluding scaffolding code), and these contain a total of 2988 ATL byte-code instructions. The total cumulative instruction coverage for all 19 test cases is 2626 instructions, or 88% of the total. Thus the instruction coverage runs well ahead of the decision coverage figure of 71%.

Just 20 of the 151 functions were never called during any of the 19 transformation runs, and these account for a total of 201 instructions. Since 362 instructions were not covered in total, this means that the remainder, 161/362, or 44% of the uncovered instructions are directly attributable to decisions not taken. Indeed, since the non-coverage of these instructions might have resulted in functions not being called, the figure of 44% is actually a lower-bound on the influence of decision coverage. This demonstrates that improving decision coverage can have a significant impact on improving the coverage of the transformation as a whole.

In the previous subsection we noted that the deficiencies in decision coverage resulted form the incomplete coverage of just 44 IF statements. In fact, we can make a further attempt to estimate the ease of localising the lack of coverage. Of the 131 functions that were called at least once during the 19 transformation runs, 104 of these have 100% decision coverage. In fact, of these, 73 contain no decision instructions, and so the function call covers all decisions by default. This means that locating the decisions not taken is localised to just 27 of the 151 functions in the ATL transformation. This suggests that tracking down incomplete decision coverage is at least feasible, even in a relatively large transformation.

6 Conclusion and Future Work

In this paper we have noted the dual nature of a model transformation: part input-recognition, like a grammar, and part generation, like program code. Thus it is possible to extend the notion of rule coverage from grammars to model transformations, and use instruction and decision coverage to evaluate the remaining elements. We have developed tool support to measure decision coverage for the transformation language ATL. Finally, we have shown how these criteria were used in the process of testing a specific model transformation.

The work presented in this paper takes place in the overall context of developing a framework for calculating metrics from various kinds of models. Our approach is based on designing a single metamodel, called the *measurement metamodel* that describes the quantifiable elements used in software metrics [17, 12]. We are in the process of developing a set of model transformations from other artifacts, such as UML class diagrams and Java programs, into this measurement metamodel. Hence, in order to ensure the correctness of the resulting metrics it is important that the model transformations faithfully represent the source models in each case.

This paper is a first step in identifying suitable white-box coverage measures. In future work we plan to validate the utility of these coverage measures primarily

by examining their correlation with coverage of the *effective metamodel*. We also intend to compare the fault-detection effectiveness of the coverage criteria presented in this paper with other test adequacy criteria in the literature such as that in [5, 2]. Using this information we plan to establish a full set of test adequacy criteria for testing model transformations and use these criteria for automating the generation of test cases for model transformation testing.

References

1. McQuillan, J.A., Power, J.F.: A survey of UML-based coverage criteria for software testing. Technical Report NUIM-CS-TR-2005-08, Dept. of Computer Science, National University of Ireland, Maynooth (September 2005)
2. Andrews, A., France, R., Ghosh, S., Craig, G.: Test adequacy criteria for UML design models. Software Testing, Verification and Reliability **13** (2003) 95–127
3. Gogolla, M., Bohling, J., Richters, M.: Validating UML and OCL models in USE by automatic snapshot generation. Journal on Software and System Modeling **4**(4) (2005) 386–398
4. Ehrig, K., Küster, J., Taentzer, G., Winkelmann, J.: Generating Instance Models from Meta Models. In: 8th IFIP Intl. Conf. on Formal Methods for Open Object-Based Distributed Systems. (June 2006) 156–170
5. Fleurey, F., Steel, J., Baudry, B.: Validation in model-driven engineering: Testing model transformations. In: Workshop on Model Design and Validation. (2004)
6. Brottier, E., Fleurey, F., Steel, J., Baudry, B., Traon, Y.L.: Metamodel-based test generation for model transformations: an algorithm and a tool. In: International Symposium on Software Reliability Engineering, Raleigh, NC (November 2006) 85–94
7. Fleurey, F., Baudry, B., Muller, P., Traon, Y.L.: Qualifying input test data for model transformations. Software and Systems Modeling (2009) (to appear)
8. Lämmel, R.: Grammar testing. In: Fundamental Approaches to Software Engineering, Genova, Italy, Springer Verlag (April 2-6 2001) 201–216
9. Lämmel, R., Schulte, W.: Controllable combinatorial coverage in grammar-based testing. In: 18th IFIP TC6/WG6.1 International Conference on Testing of Communicating Systems, New York, USA (May 2006) 19–38
10. Hennessy, M., Power, J.F.: Analysing the effectiveness of rule-coverage as a reduction criterion for test suites of grammar-based software. Empirical Software Engineering **13**(4) (2008) 343–368
11. Binder, R.: Testing Object Oriented Systems: Models, Patterns and Tools. Addison-Wesley (October 1999)
12. McQuillan, J.A., Power, J.F.: A metamodel for the measurement of object-oriented systems: An analysis using alloy. In: IEEE International Conference on Software Testing Verification and Validation, Lillehammer, Norway (April 9-11 2008) 288–297
13. Jouault, F., Allilaire, F.: The ATL virtual machine. Available on-line as `http://www.eclipse.org/m2m/atl/doc/ATL_VMPresentation[v00.01].pdf` (2006)
14. Allilaire, F., Jouault, F.: Presentation families to persons. `http://www.eclipse.org/m2m/atl/basicExamples_Patterns/` (February 2007)
15. Vépa, E.: ATL transformation example: UML2 to measure. `http://www.eclipse.org/m2m/atl/atlTransformations/`

16. The Eclipse Foundation: Eclipse modeling - model development tools: UML2 tools. http://www.eclipse.org/modeling/mdt/ (2009)
17. McQuillan, J.A., Power, J.F.: On the application of software metrics to UML models. In: Models in Software Engineering. Volume 4364 of Lecture Notes in Computer Science., Springer (2007) 217–226

```
1   module Families2Persons;
2   create OUT : Persons from IN : Families;

3   helper context Families!Member def: familyName : String =
4     if not self.familyFather.oclIsUndefined() then
5       self.familyFather.lastName
6     else
7       if not self.familyMother.oclIsUndefined() then
8         self.familyMother.lastName
9       else
10        if not self.familySon.oclIsUndefined() then
11          self.familySon.lastName
12        else self.familyDaughter.lastName
13        endif
14      endif
15    endif;

16  helper context Families!Member def: isFemale() : Boolean =
17    if not self.familyMother.oclIsUndefined() then
18      true
19    else
20      if not self.familyDaughter.oclIsUndefined() then
21        true
22      else
23        false
24      endif
25    endif;

26  rule Member2Male {
27    from
28      s : Families!Member (not s.isFemale())
29    to
30      t : Persons!Male (
31        fullName <- s.firstName + ' ' + s.familyName
32      )
33  }

34  rule Member2Female {
35    from
36      s : Families!Member (s.isFemale())
37    to
38      t : Persons!Female (
39        fullName <- s.firstName + ' ' + s.familyName
40      )
41  }
```

Fig. 3. The ATL transformation *Families2Persons*, taken directly from the examples in the ATL transformation zoo. This transformation is used as an example in Section sec:example.