ATLTest: A White-Box Test Generation Approach for ATL Transformations

Carlos Alberto González Pérez, Jordi Cabot

ATLTest: A White-Box Test Generation Approach for ATL Transformations

Carlos A. González and Jordi Cabot

École des Mines de Nantes - INRIA - LINA, Nantes, (France)
{carlos.gonzalez,jordi.cabot}@mines-nantes.fr

Abstract. MDE is being applied to the development of increasingly complex systems that require larger model transformations. Given that the specification of such transformations is an error-prone task, techniques to guarantee their quality must be provided. Testing is a well-known technique for finding errors in programs. In this sense, adoption of testing techniques in the model transformation domain would be helpful to improve their quality. So far, testing of model transformations has focused on black-box testing techniques. Instead, in this paper we provide a white-box test model generation approach for ATL model transformations.

1 Introduction

Model-Driven Engineering (MDE) is a software engineering paradigm where models play a fundamental role. They are used to specify, simulate, test, verify and generate code for the application to be built. Most of these activities are model manipulations, thus, model transformation becomes a crucial activity. Nevertheless, writing model transformations is a complex and error-prone task, specially when using MDE to develop complex systems that usually involve chains of large transformations. Therefore, having mechanisms to make model transformations more reliable has become a matter of utmost importance.

Among the possible strategies to improve the quality of model transformations, several testing techniques for model transformations have been recently proposed (see [4] for a recent survey). So far, most of the techniques follow a black-box approach (i.e. transformations are regarded as a black-box so the generation of test input models does not take into account the internals of the transformation) while only a few attempt partial white-box testing strategies (but oriented towards the coverage of the input metamodel and not that of the transformation).

In this sense, the contribution of this paper is to define a new white-box testing mechanism to generate test input models out of ATL model transformations. Our goal is to optimize the generation of the tests by maximizing the coverage of the internal transformation structure. We have chosen ATL [14] as target transformation language due to its popularity (both in academia and industry). However, many of the ideas presented herein could be applied to other

transformation languages following the rule-based paradigm in which the OCL is broadly used such as the QVT transformation language family. Our approach can be used in isolation or could be integrated with black-box testing techniques to provide an hybrid test transformation framework.

This paper is structured as follows: Section 2 provides some background on model transformation testing and motivates our approach. Section 3 describes at a high level the foundations of our approach and introduces the running example used throughout the paper. Section 4 goes into detail on how to analyze an ATL transformation to extract the information needed to generate the test input models. Section 5 describes the generation of test input models using the information extracted in Section 4. Finally, Section 6 reviews the related work, and some conclusions and further work are drawn in Section 7.

2 Background and Motivation

Software testing, also known as program testing, can be viewed as the destructive process of trying to find the errors (whose presence is assumed) in a program or piece of software, of course, with the intent of establishing some degree of confidence that the program does what it is expected to do [19]. A common methodology to test a piece of software generally comprises a number of well known steps, namely the creation of input test cases, running the software with the test cases, and finally, using an oracle to analyze the results yielded to determine whether errors came up or not. An oracle is any program, process or body of data that specifies the expected outcome for a set of test cases as applied to a tested object [5] and it can be as simple as a manual inspection or as complex as a separate piece of software.

It is generally accepted that the more input tests are created and the more time is spent running the software, the higher is the probability of finding errors and therefore end up with a more reliable software. However, since finding all the errors presented in a piece of software is impossible [5], and the number of test cases that can be created to test a piece of software can be potentially infinite, it is necessary to establish some strategy to carry out testing in an effective way. Two of the most prevalent strategies are black-box testing and white-box testing. The main difference betweeen them is that in black-box testing only the program specification is taken into account at the time of designing test cases, whereas in white-box testing test cases are created out of the analysis of program internals. Mixed strategies combining both approaches are usually encouraged, though, in order the get a better testing experience.

The methodology to test a model transformation is essentially the same as for software testing and, therefore the same conclusions can be applied. However, compared to program testing, model transformation testing must face an additional challenge [3]: the complex nature of model transformation inputs and outputs. Models can be large structures and must conform to a meta-model (possibly extended with OCL well-formedness rules) thus making even harder the generation of test models and the analysis of the results. Fig. 1 shows what

a mixed strategy to test model transformations looks like, in which black-box testing approaches derive test cases from the transformation specification and white-box approaches do it out of its implementation.

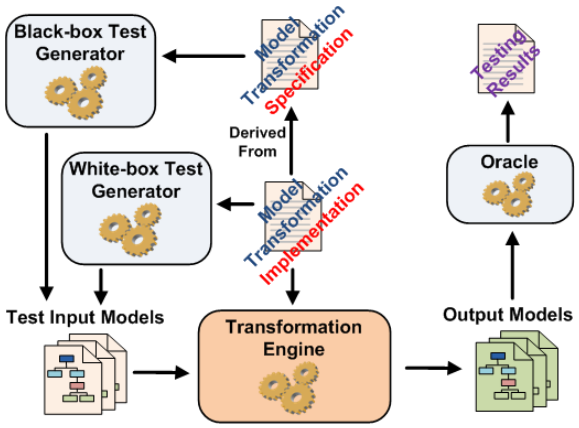

Fig. 1: Mixed approach to model transformation testing

So far, the generation of input test models by means of black-box techniques has become more popular since, unlike white-box approaches, they do not need to deal with the technology or transformation language employed in the implementation of the model transformation. In relation to this, our motivations to present a white-box testing approach are twofold: On the one hand, our approach could be combined with black-box approaches to facilitate the creation of mixed testing strategies. On the other hand, at the time of implementing a model transformation, a formal specification is not always available, thus making difficult or even impossible the application of black-box approaches to generate input test models. In these scenarios, white-box testing techniques can be of special relevance.

3 ATLTest: Test Input Models for ATL Transformations

3.1 Overall Picture

ATLTest is a white-box test generation approach for ATL transformations. In traditional white-box testing, test generation is a 2-step process in which, typically a control flow graph or a data flow graph is generated in the first place, out of an analysis of the source code, and then, a set of test cases is obtained from traversing the graph a specific number of times, usually determined by some coverage criteria, like for example decision coverage. Although essentially the same, compared to traditional white-box approaches, the test generation process in ATLTest exhibits some differences, basically due to the mixed declarative and

imperative model transformation language constructs of ATL and the complex nature of model transformation inputs.

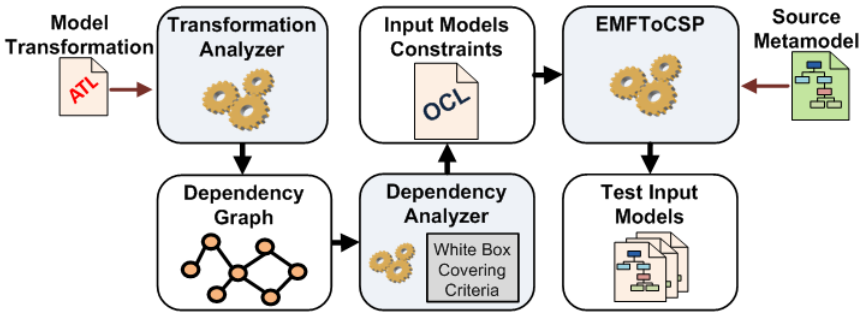

Fig. 2: ATLTest: Overall picture

More specifically, the test generation process in ATLTest, depicted in Fig. 2, consists of three separate steps. In the first one, the ATL transformation is analyzed and a graph abstracting the relevant information for the test generation phase is produced. This graph, called "dependency graph", plays the same role in ATLTest that control flow graphs or data flow graphs play in other traditional approaches, although it is substantially different in nature. For now it suffices to say that the dependency graph represents groups of interrelated conditions expressed in the OCL, that must be hold (totally or partially) by the test input models.

Once the analysis of the ATL transformation is done, the second step is to traverse the dependency graph a number of times which, as for traditional approaches, is determined by some coverage criteria. Traversing the dependency graph implies setting truth values for the different conditions in the graph and, therefore, each traversal will yield a set of constraints that symbolizes a family of relevant test cases for the transformation (i.e. the constraints characterize the structure/values of possible sample input models corresponding to that test case).

In the last step, the actual test cases (i.e. the test input models to be used when executing the transformation) are created by computing models conforming to the source metamodel and satisfying the constraints for the test case. This computation can be performed using any of the SAT-based or CSP-based solvers available. In particular, we use EMFtoCSP[1] [12] to generate the input test models. EMFtoCSP is an Eclipse[2]-integrated tool for the automatic verification of UML models and EMF models annotated with OCL constraints by means of reexpressing them as a constraint satisfaction problem. In the context of model transformation testing, EMFtoCSP will generate solutions (i.e. sample

[1] http://code.google.com/a/eclipselabs.org/p/emftocsp/
[2] http://www.eclipse.org/

models) that satisfy both the source metamodel and the additional OCL expressions resulting from the graph traversal. A single sample model suffices to cover the corresponding test case.

In the next sections we will describe in more detail the foundations and rationale behind ATLTest.

3.2 Running Example

To illustrate our approach we will be using as a running example the following transformation that converts *publications* into *books* (see Fig. 3). In a nutshell, the model transformation contains two rules (*Publication2Book* and *PubSection2Chapter*) to respectively transform "Publication" and "PubSection" input elements into "Book" and "Chapter" output elements. Those elements are only transformed if the respective flags "isBook" and "isChapter" are activated.

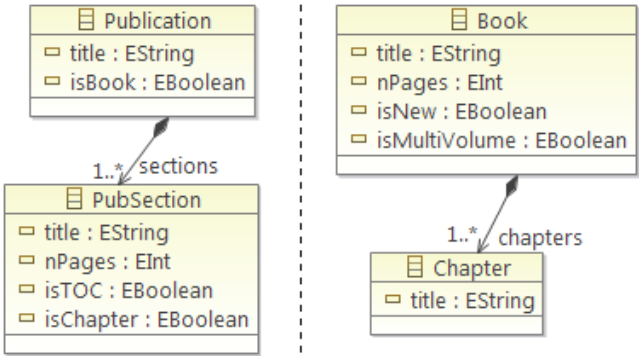

Fig. 3: Source (left) and target (right) metamodels for the running example

```
module Publication2Book;
create OUT : Book from IN : Publication;
rule Publication2Book {
  from p: Publication!Publication (p.isBook)
  to   b: Book!Book (
          title<-p.title,
          isMultiVolume<-p.sections->select(s| s.isChapter)->
          size()>25 and p.sections->select(s| s.isTOC)->size()>2,
          chapters<-p.sections->select(s| s.isChapter),
          nPages<-p.sections->collect(s| s.nPages)->sum()   )
}
rule PubSection2Chapter {
  from ps: Publication!PubSection (ps.isChapter)
  to    c: Book!Chapter ( title<-ps.title )
}
```

4 Dependency Graph Generation

The ATL language includes a variety of constructs (matched rules, lazy rules, helpers, etc) but in most of them OCL plays a key role. Therefore any white-box testing approach for ATL must devote a special attention to the OCL expressions appearing in the transformation.

In fact, OCL expressions are at the heart of the mechanism to create the dependency graph. In a nutshell, the majority of nodes and arcs are generated out of the analysis of certain OCL expressions found in the rules and helpers making up the ATL transformation, thus forming the building blocks of the dependency graph. The analysis of the rules and helpers containing those OCL expressions extends and interconnects those building blocks. The process is described in more detail in the following subsections.

4.1 Analysis of OCL Expressions

OCL expressions have a clear impact on the number and structure of interesting input models to use as tests for the model transformation. To ensure the coverage of the model transformation we should make sure that the test models evaluate to a different result the several OCL expressions in the transformation.

Let's consider the OCL expression `p.sections->select(s|s.isChapter)` extracted from the running example. The expression is part of a binding in the first rule, aimed at generating as many "Chapter" elements in the output model as "PubSection" elements with the flag "isChapter" set to "True" are present in the input model. Clearly, when looking at this expression we immediately think of different situations that should be tested, e.g. "What happens if there are no "PubSection" elements in the input model?" or "What happens if none of the "PubSection" elements are flagged as chapters?". Therefore, input models that test each situation (i.e. an input model with no "PubSections", a model with "PubSections", a model with "PubSections" in which none of them is flagged as a chapter,...) should be generated by our method.

Each question above can be characterized by means of a boolean OCL expression (for the former example `PubSection::allInstances()->notEmpty()` and `PubSection::allInstances()->select(s|s.ischapter)->notEmpty()` could be those expressions). Each expression would constitute a node in the dependency graph (meaning that the generated tests may include the condition in the node depending on how the graph is traversed as explained in the next section). It is also worth noting that it does not make much sense to check the second condition if the first one does not hold (we cannot create at the same time a model with no "PubSection" elements and a non-empty list of "PubSection" elements, some of them flagged as chapters), which means that the two conditions are somehow interrelated. This interrelation is the reason why we call the graph, dependency graph. There is a dependency between the two conditions, expressed as an arc between the two nodes. Obviously, these arcs play a key role in the traversal of the graph during the test generation phase.

In the rest of the section we generalize this discussion to arbitrary OCL expressions. We have identified three different big groups of OCL expressions relevant to the process sketched above, namely, expressions in the context of collections (Table 1), iterative operations (Table 2) and boolean expressions (Table 3). Each row in the tables show how the dependency graph is extended when finding an expression of that type in an ATL construct. The dependency graph is expressed as two ordered sets that contain the nodes (V) and the arcs (E) in the order they are created, where nodes are described with an OCL expression, and arcs are expressed as "(x,y)", "x" and "y" being the positions of the source and target nodes in the corresponding set. In this regard, "last" is used to make reference to the last position in a set, and in the case of complex OCL expressions, "$G_x(V)$" and "$G_x(E)$" make reference to the respective sets of nodes and arcs obtained from the analysis of the source expression "x". Similar for "$G_{body}(V)$" and "$G_{body}(E)$" in table 2.

One important remark is that, in order to be considered for analysis, all these OCL expressions must reference at least one element of the input metamodel, since these are the most relevant for test generation. The identification of the OCL expressions suitable for analysis can be done by traversing the abstract syntax tree of the OCL expressions in the ATL transformation.

To finish this subsection, we illustrate how to create nodes 3, 4, 5, 6 and 7 of the dependency graph in Fig. 6 by applying the information in the tables to the following expression from the running example:

```
isMultiVolume<-p.sections->select(s| s.isChapter)->size() > 25
and p.sections->select(s| s.isTOC)->size() > 2                (exp1)
```

To begin with, the OCL expression at the right of "<-", matches entry 10 in table 3 using "and" as "Op". According to this entry, the 2-step process depicted in Fig. 4 must be carried out. That is, subexpressions at the left and at the right of "and" must be analyzed, thus yielding several nodes and arcs, and then some of those nodes are merged. Finally all the nodes are interconnected.

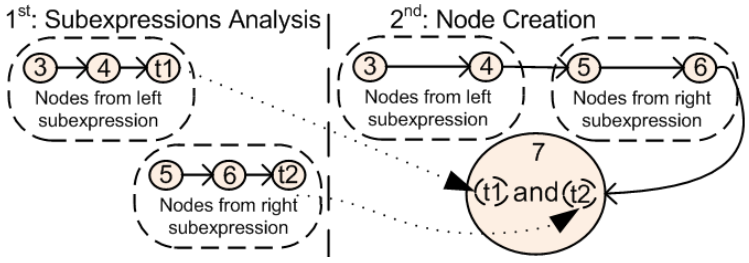

Fig. 4: Actions to carry out when applying entry 10 in table 3 to the running example

The expression at the left of "and" in (exp1) is

```
p.sections->select(s| s.isChapter)->size() > 25               (exp2)
```

Table 1: Nodes and arcs generated out of OCL operations in the context of a collection

	OCL Expression	**G=(V,E)**
1	$Obj_c.[nav \mid nav \rightarrow notEmpty()]$	$V = \{C :: allInstances() \rightarrow select(c \mid c.nav \rightarrow notEmpty()) \rightarrow notEmpty()\}$
2	$C :: allInstances()[\rightarrow notEmpty()]$	$V = \{C :: allInstances() \rightarrow notEmpty()\}$
3	$Obj_c.nav \rightarrow isEmpty()$	$V = \{C :: allInstances() \rightarrow select(c \mid c.nav \rightarrow isEmpty()) \rightarrow notEmpty()\}$
4	$C :: allInstances() \rightarrow isEmpty()$	$V = \{C :: allInstances() \rightarrow isEmpty()\}$
5	$c \rightarrow isEmpty()$	$V = \{c \rightarrow isEmpty()\} \cup G_c(V)$, $E = \{(G_c(V)[last], 1)\} \cup G_c(E)$
6	$c \rightarrow notEmpty()$	$V = \{c \rightarrow notEmpty()\} \cup G_c(V)$, $E = \{(G_c(V)[last], 1)\} \cup G_c(E)$
7	$c \rightarrow [size() \mid last() \mid sum() \mid append(o) \mid flatten() \mid first() \mid including(o) \mid prepend(o)]$	$V = G_c(V)$, $E = G_c(E)$
8	$c \rightarrow [includes(o) \mid count(o) \mid indexOf(o) \mid excluding(o)]$	$V = \{c \rightarrow includes(o)\} \cup G_c(V)$, $E = \{(G_c(V)[last], 1)\} \cup G_c(E)$
9	$c \rightarrow excludes(o)$	$V = \{c \rightarrow excludes(o)\} \cup G_c(V)$, $E = \{(G_c(V)[last], 1)\} \cup G_c(E)$
10	$c \rightarrow includesAll(cl)$	$V = \{c \rightarrow includesAll(cl)\} \cup G_c(V) \cup G_{cl}(V)$, $E = \{(G_c(V)[last], G_{cl}(V)[1]), (G_{cl}(V)[last], 1)\} \cup G_c(E) \cup G_{cl}(E)$
11	$c \rightarrow excludesAll(cl)$	$V = \{c \rightarrow excludesAll(cl)\} \cup G_c(V) \cup G_{cl}(V)$ $E = \{(G_c(V)[last], G_{cl}(V)[1]), (G_{cl}(V)[last], 1)\} \cup G_c(E) \cup G_{cl}(E)$
12	$c \rightarrow union(cl)$	$V = G_c(V) \cup G_{cl}(V)$, $E = \{(G_c(V)[last], G_{cl}(V)[1])\} \cup G_c(E) \cup G_{cl}(E)$
13	$c \rightarrow [insertAt(n, o) \mid at(n)]$	$V = \{c \rightarrow size() \geq n\} \cup G_c(V)$ $E = \{(G_c(V)[last], 1)\} \cup G_c(E)$
14	$c \rightarrow subSequence(l, u)$	$V = \{c \rightarrow size() \geq u\} \cup G_c(V)$ $E = \{(G_c(V)[last], 1)\} \cup G_c(E)$
15	$c \rightarrow [intersection(cl) \mid symetricDifference(cl)]$	$V = \{c \rightarrow includesAll(cl)$ *or* $cl \rightarrow includesAll(c)\} \cup G_c(V) \cup G_{cl}(V)$, $E = \{(G_c(V)[last], G_{cl}(V)[1]), (G_{cl}(V)[last], 1)\} \cup G_c(E) \cup G_{cl}(E)$

that matches entry 11 in table 3 where "CompOp" is ">" and "LitValue" is "25". Fig. 5 illustrates the process to be carried out when instructions in this entry are followed. The subexpression on the left side is analyzed in the first place, this way yielding nodes 3 and 4, and then, the node "t1" is created.

Now let's see in detail how nodes 3 and 4 are generated. The expression on the left of (exp2) is

```
p.sections->select(s| s.isChapter)->size()                    (exp3)
```

Table 2: Generation of nodes and arcs out of OCL iterative operations

	OCL Expression	**G=(V,E)**
1	$c \rightarrow exists(body)$	$V = \{c \rightarrow exists(body)\} \cup G_c(V) \cup G_{body}(V)$, $E = \{(G_c(V)[last], G_{body}(V)[1]), (G_{body}(V)[last], 1)\} \cup G_c(E) \cup G_{body}(E)$
2	$c \rightarrow forAll(body)$	$V = \{c \rightarrow forAll(body)\} \cup G_c(V) \cup G_{body}(V)$, $E = \{(G_c(V)[last], G_{body}(V)[1]), (G_{body}(V)[last], 1)\} \cup G_c(E) \cup G_{body}(E)$
3	$c \rightarrow isUnique(body)$	$V = \{c \rightarrow isUnique(body)\} \cup G_c(V) \cup G_{body}(V)$, $E = \{(G_c(V)[last], G_{body}(V)[1]), (G_{body}(V)[last], 1)\} \cup G_c(E) \cup G_{body}(E)$
4	$c \rightarrow one(body)$	$V = \{c \rightarrow one(body)\} \cup G_c(V) \cup G_{body}(V)$, $E = \{(G_c(V)[last], G_{body}(V)[1]), (G_{body}(V)[last], 1)\} \cup G_c(E) \cup G_{body}(E)$
5	$c \rightarrow [collect(body) \mid sortedBy(body)]$	$V = G_c(V)$, $E = G_c(E)$
6	$c \rightarrow [reject(body) \mid any(body) \mid select(body)]$	$V = G_c(V) \cup G_{body}(V)$, $E = \{(G_c(V)[last], G_{body}(V)[1])\} \cup G_c(E) \cup G_{body}(E)$

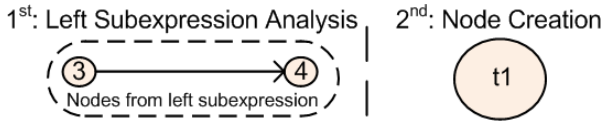

Fig. 5: Actions to carry out when applying entry 11 in table 3 to the running example

that matches entry 7 in table 1. According to this entry, it is necessary to analyze the source collection of (exp3), that is:

```
p.sections->select(s| s.isChapter)                    (exp4)
```

It matches entry 6 in table 2. This entry indicates that (exp4) has the form `c->select(body)` and therefore "c" and "body" expressions must be analyzed. In (exp4), "c" is `p.sections` and "body" is `s.isChapter`. They match respectively entry 1 in table 1 and entry 2 in table 3. This way, we finally obtain nodes 3 and 4 that can be seen in Fig. 6. It is important to remember that the creation of nodes 3 and 4 is just the first step in the analysis of (exp2), as exposed in Fig. 5. Now it is time to complete the analysis of this expression by creating the node "t1". This node is made up by the following OCL expression:

```
p.sections->select(s| s.isChapter)->size() > 25       (t1)
```

It is the time to remember that the analysis of (exp2) is just the analysis of the left subexpression of (exp1). As can be seen in Fig. 4 the analysis of (exp1) continues with the analysis of its right subexpression. We omit a detailed description of this analysis, though, since it is very similar to the one just described. It suffices to say that the analysis of the right subexpression of (exp1) yields nodes 5

Table 3: Generation of nodes and arcs out of boolean OCL operations

	Boolean OCL Expression	**G=(V,E)**
1	$[True\|False]$	$V = \varnothing, E = \varnothing$
2	$[not]Obj_A.boolAttr$	$V = \{A :: allInstances() \rightarrow$ $select(a\|[not]a.boolAttr) \rightarrow notEmpty()\}$
3	$Obj_A.[attr\|nav].oclIsUndefined()$	$V = \{A :: allInstances() \rightarrow$ $select(a\|a.[attr\|nav].oclIsUndefined()) \rightarrow$ $notEmpty()\}$
4	$expr.oclIsUndefined()$	$V = \{expr \rightarrow oclIsUndefined()\} \cup G_{expr}(V),$ $E = \{(G_{expr}(V)[last], 1)\} \cup G_{expr}(E)$
5	$expr.oclIsKindOf(t)$	$V = \{expr \rightarrow oclIsKindOf(t)\} \cup G_{expr}(V),$ $E = \{(G_{expr}(V)[last], 1)\} \cup G_{expr}(E)$
6	$expr.oclIsTypeOf(t)$	$V = \{expr \rightarrow oclIsTypeOf(t)\} \cup G_{expr}(V),$ $E = \{(G_{expr}(V)[last], 1)\} \cup G_{expr}(E)$
7	$Obj_A.attr\ CompOp\ LitValue$	$V = \{A :: allInstances() \rightarrow select(a\|$ $a.attr\ CompOp\ LitValue) \rightarrow notEmpty()\}$
8	$Obj_A.attr\ Op\ Obj_B.attr$	$V = \{A :: allInstances() \rightarrow select(a\|$ $B :: allInstances() \rightarrow exists(b\|$ $a.attr\ CompOp\ b.attr)) \rightarrow notEmpty()\}$
9	$Obj_A.attr\ Op\ Obj_B.attr$ $Op\ ...\ Op\ Obj_N.attr$	$V = \{A :: allInstances() \rightarrow$ $select(a\|B :: allInstances() \rightarrow exists(b\|... \rightarrow$ $exists(n\|a.attr\ Op\ b.attr\ Op\ ...\ Op\ n.attr)...))$ $\rightarrow notEmpty()\}$
10	$expr_1\ Op\ expr_2$	$V = \{G_{expr_1}(V)[last]\ Op\ G_{expr_2}(V)[last]\} \cup$ $\{G_{expr_1}(V)[1], ..., G_{expr_1}(V)[last-1]\} \cup$ $\{G_{expr_2}(V)[1], ..., G_{expr_2}(V)[last-1]\},$ $E = \{(G_{expr_1}(V)[last-1], G_{expr_2}(V)[1]),$ $(G_{expr_2}(V)[last], 1)\} \cup$ $\{G_{expr_1}(E)[1], ..., G_{expr_1}(E)[last-1]\} \cup$ $\{G_{expr_2}(E)[1], ..., G_{expr_2}(E)[last-1]\}$
11	$expr\ CompOp\ LitValue$	$V = \{expr\ CompOp\ LitValue\} \cup G_{expr}(V),$ $E = \{(G_{expr}(V)[last], 1)\} \cup G_{expr}(E)$

and 6 that can be seen in Fig. 6, as well as node "t2", made up by the following OCL expression:

```
p.sections->select(s| s.isTOC)->size() > 2                    (t2)
```

Finally, applying last step shown in Fig. 4, node 7 is created out of the union of nodes "t1" and "t2", expressed in terms of the "allInstances()" operator, and the different nodes created during the process are interconnected. The final result can be seen in Fig. 6.

The analysis of the rest of OCL expressions in the sample model transformation can be carried out in the same way.

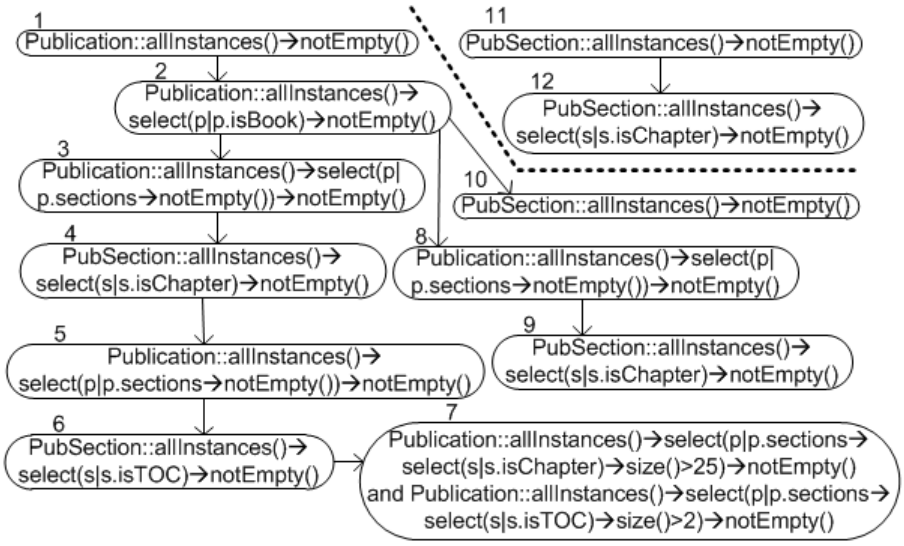

Fig. 6: Dependency graph of the example, made up by two connected components

4.2 Analysis of Rules and Helpers

As we have seen, the analysis of OCL expressions yields the building blocks of the dependency graph. In this subsection we cover the analysis of rules and helpers, coarse-grained elements of ATL transformations.

There are different types of rules in ATL, namely, matched rules, lazy rules and called rules. The first two are declarative rules while the last one is an imperative type of rule.

The analysis of a declarative rule focuses on the `from` section of the rule, that indicates the conditions that trigger the rule, the `to` section of the rule, that describes how elements of the target model are created, and the optional `do` section of the rule, used to enable the specification of imperative statements.

The analysis of the `from` section produces a node with the OCL expression `in_type::allInstances()->notEmpty()`, where "in_type" refers to the model element that will be matched by the rule. Optionally, this section can include a boolean OCL expression, as a filter to limit the "in_type" elements that can trigger the rule. When present, this filter is analyzed according to the instructions of subsection 4.1 and, in this case, the node created in the first place is connected to the first node rendered by the filter analysis.

Returning to the running example, the analysis of the `from` section of the rule "Publication2Book", that includes the condition `p.isBook`, produces nodes 1 and 2 in Fig. 6. Analogously, the `from` section of the rule "PubSection2Chapter" generates nodes 11 and 12 that made up the second connected component of the dependency graph.

The `to` section of a declarative rule is, essentially, a collection of bindings describing how elements of the target metamodel are created. Each binding has

the form `feature-name <- exp`, being "exp" an OCL expression. The result of analyzing this section is a number of interconnected nodes, obtained from the analysis of each "exp" element as explained in subsection 4.1. Finally, the first node in each of the groups of nodes rendered is connected to the last node in the group of nodes obtained from the analysis of the `from` section of the rule.

The `do` section of a declarative rule allows the specification of imperative statements. This section is analyzed by looking for OCL expressions suitable for analysis. When found, those expressions are analyzed according to the directions of subsection 4.1. This approach is also applied at the time of analyzing called rules.

To finish the description of the dependency graph generation process, one word about ATL helpers. Helpers can be viewed as the ATL equivalent to methods and can be called from different points in an ATL transformation. Each helper has a body, specified as an OCL expression. If during the analysis of the elements described above and in subsection 4.1, a call to a helper is found, then its body is analyzed like any other OCL expression and the rendered nodes are included as resulting from the analysis of the element where the call was found.

One last remark that is worth mentioning is that depending on the complexity of the ATL transformation under analysis (number of rules, presence of imperative sections, etc.), the resulting dependency graph can be made up by more than one connected component.

5 Test Input Models Generation

Once the dependency graph is created, the next step consists in traversing it a number of times, each time determining the set of constraints a new test case must fulfill. The process is directed by a coverage criterion, which eventually determines the number of traversals, and consequently, the number of test cases to be generated.

In white-box testing, coverage criteria help designers to select the structural elements of the software (model transformations in this case) that will be the focus of the testing and to determine the desired intensity of the testing efforts. The coverage criteria drive the creation of the tests to make sure the tests cover the selected parts of the transformation and do it enough to gain the desired confidence on their correctness. The fact that a test suite covers an element means that it exists at least one test case that exercises that element. This is known as coverage analysis.

Branches in the program logic are elements typically selected as object of coverage analysis in white-box testing. There are a number of classical white-box coverage criteria that follow this approach, like for example, "condition coverage" or "multiple-condition coverage" [19]. Both focus on making sure that all branches in the program are covered, but they differ on how they exercise conditional branches where the condition is not atomic. In the case of "condition coverage", complete coverage is achieved by simply ensuring that the test cases exercise each branch with all possible outcomes at least once (i.e. for a boolean

branch, the test suite must include a test case where the branch evaluates to "False" and one where it evaluates to "True"). However, "multi-condition coverage" requires the test suite to include a test case for each individual combination of truth values of the subconditions conforming the branch condition.

These and other similar criteria can be easily adapted to our approach. Since in the dependency graph each node contains a boolean expression, condition coverage and multi-condition coverage can be applied by considering each node as a branch, with the particularity that every time the condition in the node evaluates to "False" the traversal of the actual connected component ends and goes on with the next one. In other case, a neighbour node is visited and the traversal continues.

This way, the application of the two coverage criteria consists on traversing the dependency graph a number of times, each time asigning either different output values to each OCL expression (condition coverage), or different combinations of truth values to each component of a complex OCL expression (multi-condition coverage). After "n" traversals, "n" sets of constraints to characterize "n" test cases will have been obtained.

Eventually, once the sets of constraints have been obtained, the execution of EMFtoCSP over each set will yield the set of input models to test the model transformation[3].

Retaking our example, we are going to show what the results of one traversal of the graph shown in Fig. 6 would be in every approach. Let's suppose that the sequence of truth values assigned to the nodes of the first connected component is <1,True>, <2,True>, <3,True>, <4,True>, <5,True>, <6,True>, and then, in the case of "condition coverage" node 7 is set to <7,True>, and in the case of "multi-condition coverage" is set to <7,(False,True)>. In the second connected component, the expressions will be set as <11,True>, <12,True>, for both approaches.

Applying "condition coverage", the constraints obtained are:

```
Publication::allInstances()->notEmpty()=true
Publication::allInstances()->select(p|p.isBook)->notEmpty()=true
Publication::allInstances()->select(p|p.sections->notEmpty())
   ->notEmpty()=true
PubSection::allInstances()->select(s|s.isChapter)->notEmpty()=true
Publication::allInstances()->select(p|p.sections->notEmpty())
   ->notEmpty()=true
PubSection::allInstances()->select(s|s.isTOC)->notEmpty()=true
Publication::allInstances()->select(p|p.sections->
   select(s|s.isChapter)->size()>25)->notEmpty() and
   Publication::allInstances()->select(p|p.sections->
   select(s|s.isTOC)->size()>2)->notEmpty()=true
```

[3] Some assignments can cause contradictory sets of OCL expressions (e.g. if the same subexpressions are used in two connected components and they are assigned different truth values in the same iteration). In those situations, EMFtoCSP will return an empty result and the test case will be discarded.

```
PubSection::allInstances()->notEmpty()=true
PubSection::allInstances()->select(s|s.isChapter)->notEmpty()=true
```

Running EMFtoCSP over the input metamodel constrained with the expressions above yields the model that can be seen in Figure 7 a).

For "multi-condition coverage", only the expression of node 7 changes:

```
Publication::allInstances()->select(p|p.sections->
   select(s|s.isChapter)->size()> 25)->notEmpty()=false and
   Publication::allInstances()->select(p|p.sections->
   select(s|s.isTOC)->size()>2)->notEmpty()=true
```

Running again EMFtoCSP, we obtain the model of Figure 7 b).

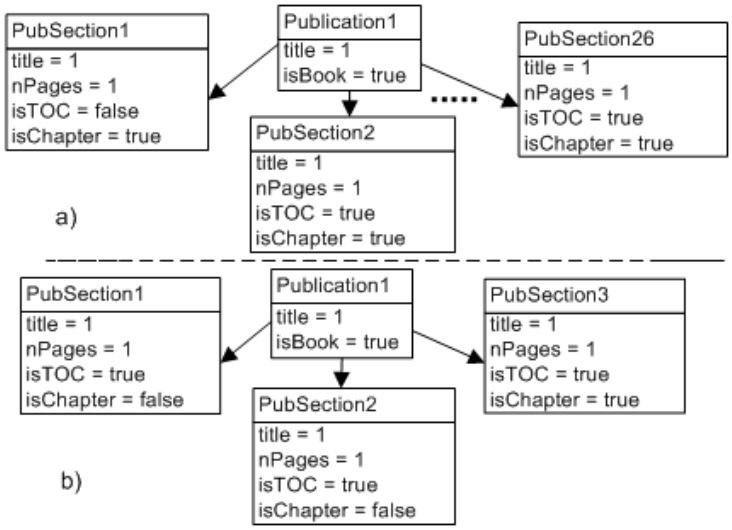

Fig. 7: Results of the example

6 Related Work

One of the most important tasks when testing a model transformation is the creation of an adequate set of test input models. Currently, the majority of approaches facing this challenge are based on black-box techniques [11, 9, 10, 16, 21, 22, 3, 6, 20, 8, 13].

As far as we know only two white-box approaches for transformation testing have been proposed [9, 15]. Both address the identification of the relevant parts of the input metamodel to be exercised by the tests: by looking at the transformation definition they detect the subset of the metamodel (and possible relevant

values for the metamodel attributes) that is accessed during the transformation and thus focus the generation of tests on that subset. In our case, the coverage of the input metamodel is derived from the test cases generated when addressing the coverage of the model transformation internal structure. This analysis of the internal transformation structure also guarantees that our tests exercise all branches in the transformation, this way maximizing their effectiveness.

White-box techniques can also be used in coverage analysis, to measure the quality of the generated test models. Regarding this, [17] proposes a number of white-box coverage measures for ATL transformations, namely rule coverage, instruction coverage and decision coverage, that are used to check how a number of test cases cover ATL transformations. This could be useful to check the quality of the tests generated with our approach, especially for model transformations where the designer may want to limit the number of tests generated.

It is worth noting that the generation of test cases out of OCL expressions is not exclusive of model transformation testing, on the contrary, it is also an important method for the verification and validation of UML/OCL specifications. Regarding this, [7] and [2] propose approaches to generate test data from OCL specifications, based on the utilisation of Higher-Order Logic and constraint solving techniques, respectively. Another approach based on the utilisation of constraint solving techniques is proposed in [1] to generate test cases out of UML specifications, although in this case only a limited subset of the OCL is supported. Finally, [23] proposes an approach to evaluate the quality of test cases generated from OCL expressions based on the utilization of several coverage criteria.

7 Conclusions and Future Work

We have presented ATLTest, a white-box testing approach for the generation of test input models for ATL transformations. Our approach tries to optimize the effectiveness of the generated tests by maximizing the coverage of the internal structure of the model transformation under analysis. ATLTest could be combined with black-box testing techniques to create mixed test generation approaches. In ATLTest, each test case is characterized by a set of OCL expressions that define the possible structure of the test input models for that test case. Sample test models satisfying the OCL constraints are created automatically using the EMFtoCSP tool.

As further work, we plan to extend our to approach to cover other transformation languages like QVT. We would also like to study complexity metrics like cyclomatic complexity [18] to establish a limit on the number of test cases that need to be created, something that can be specially useful when testing large transformations. Finally, ATLTest is a first step in the development of a full model transformation testing framework called ATLUnit, where different test cases generation approaches could be combined.

References

1. Aertryck, L.V., Jensen, T.: Uml-casting: Test synthesis from uml models using constraint resolution. In: Proceedings of AFADL2003 (Approches Formelles Dans L'Assistance Au Dévelopment De Logiciel) (2003)
2. Aichernig, B.K., Salas, P.A.P.: Test case generation by ocl mutation and constraint solving. In: QSIC. pp. 64–71. IEEE Computer Society (2005)
3. Baudry, B., Dinh-Trong, T., Mottu, J.M., Simmonds, D., France, R., Ghosh, S., Fleurey, F., Le Traon, Y.: Model transformation testing challenges. In: Proceedings of IMDT Workshop in conjunction with ECMDA06 (2006)
4. Baudry, B., Ghosh, S., Fleurey, F., France, R.B., Traon, Y.L., Mottu, J.M.: Barriers to systematic model transformation testing. Commun. ACM 53(6), 139–143 (2010)
5. Beizer, B.: Software Testing Techniques, 2nd Ed. Int. Thomson Computer Press (1990)
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: ISSRE. pp. 85–94. IEEE Computer Society (2006)
7. Brucker, A.D., Krieger, M.P., Longuet, D., Wolff, B.: A specification-based test case generation method for uml/ocl. In: MoDELS Workshops. LNCS, vol. 6627, pp. 334–348. Springer (2010)
8. Fiorentini, C., Momigliano, A., Ornaghi, M., Poernomo, I.: A constructive approach to testing model transformations. In: ICMT. LNCS, vol. 6142, pp. 77–92. Springer (2010)
9. Fleurey, F., Steel, J., Baudry, B.: Validation in model-driven engineering: testing model transformations. In: Proceedings of first Int. Workshop on Model, Design and Validation. pp. 29 – 40 (nov 2004)
10. Fleurey, F., Baudry, B., Muller, P.A., Traon, Y.L.: Qualifying input test data for model transformations. Software and System Modeling 8(2), 185–203 (2009)
11. Gogolla, M., Vallecillo, A.: *Tract*able model transformation testing. In: ECMFA. LNCS, vol. 6698, pp. 221–235. Springer (2011)
12. González, C.A., Büttner, F., Clarisó, R., Cabot, J.: Emftocsp: A tool for the lightweight verification of emf models. In: FormSERA - Formal Methods in Software Engineering: Rigorous and Agile Approaches. ICSE Satellite Events (2012), [To appear]
13. Guerra, E.: Specification-driven test generation for model transformations. In: ICMT. LNCS, vol. 7307, pp. 40–55. Springer (2012)
14. Jouault, F., Kurtev, I.: Transforming models with atl. In: MoDELS Satellite Events. LNCS, vol. 3844, pp. 128–138. Springer (2005)
15. Küster, J.M., Abd-El-Razik, M.: Validation of model transformations - first experiences using a white box approach. In: MoDELS Workshops. LNCS, vol. 4364, pp. 193–204. Springer (2006)
16. Lamari, M.: Towards an automated test generation for the verification of model transformations. In: SAC. pp. 998–1005. ACM (2007)
17. Mc Quillan, J.A., Power, J.F.: White-box coverage criteria for model transformations. Department of Computer Science, National University of Ireland (Jul 2009)
18. McCabe, T.J.: A complexity measure. IEEE Trans. Software Eng. 2(4), 308–320 (1976)
19. Myers, G.J.: The Art of Software Testing, 2nd Ed. John Wiley & Sons, Inc. (2004)
20. Sen, S., Baudry, B., Mottu, J.M.: On combining multi-formalism knowledge to select models for model transformation testing. In: ICST. pp. 328–337. IEEE Computer Society (2008)

21. Sen, S., Baudry, B., Mottu, J.M.: Automatic model generation strategies for model transformation testing. In: ICMT. LNCS, vol. 5563, pp. 148–164. Springer (2009)
22. Wang, J., Kim, S.K., Carrington, D.A.: Automatic generation of test models for model transformations. In: Australian Software Engineering Conference. pp. 432–440. IEEE Computer Society (2008)
23. Weißleder, S., Schlingloff, B.H.: Quality of automatically generated test cases based on ocl expressions. In: ICST. pp. 517–520. IEEE Computer Society (2008)