Towards effective mutation testing for ATL

Esther Guerra
Universidad Autónoma de Madrid (Spain)
Esther.Guerra@uam.es

Jesús Sánchez Cuadrado
Universidad de Murcia (Spain)
jesusc@um.es

Juan de Lara
Universidad Autónoma de Madrid (Spain)
Juan.deLara@uam.es

***Abstract*—The correctness of model transformations is crucial to obtain high-quality solutions in model-driven engineering. Testing is a common approach to detect errors in transformations, which requires having methods to assess the effectiveness of the test cases and improve their quality.**

Mutation testing permits assessing the quality of a test suite by injecting artificial faults in the system under test. These emulate common errors made by competent developers and are modelled using mutation operators. Some researchers have proposed sets of mutation operators for transformation languages like ATL. However, their suitability for an effective mutation testing process has not been investigated, and there is no automated mechanism to generate test models that increase the quality of the tests.

In this paper, we use transformations created by third parties to evaluate the effectiveness ATL mutation operators proposed in the literature, and other operators that we have devised based on empirical evidence on real errors made by developers. Likewise, we evaluate the effectiveness of commonly used test model generation techniques. For the cases in which a test suite does not detect an injected fault, we synthesize test models able to detect it. As a technical contribution, we make available a framework that automates this process for ATL.

***Index Terms*—Model transformations, Mutation testing, ATL.**

I. Introduction

Model transformations enable the model-driven engineering (MDE) [1] vision of software production by supporting automated model manipulation. Hence, their correctness is crucial for the success of MDE projects. Many efforts have been devoted to the validation and verification of model transformations [2], including static analysis [3]–[5], spectrum based techniques [6], backward analysis [7], [8], and results based on graph transformation [9], among others.

In practice, transformations are often verified by testing [2]. Input test models are created either manually in ad-hoc ways, or automatically using random instantiators or coverage criteria over some artefact, like the meta-model [10], [11], the transformation [12] or transformation contracts [13]–[15]. Similarly, we can check the correctness of the transformation results either using partial oracles specified as contracts [13], [15], or full oracles consisting of the expected target model.

Several researchers have reported that transformations are error-prone [4], [16], while [4] showed that most transformations in the ATL zoo repository have errors. This evinces the need for mechanisms to evaluate and improve the quality of transformation test cases. In this respect, mutation testing [17], [18] is a well-known technique to evaluate the quality of a test suite. It is based on the use of mutation operators that inject artificial flaws in the system under test, emulating errors made by competent developers. The result is a set of mutants of the original system, which are executed against the test suite. If the test suite detects the injected errors, the mutant is killed; or else it remains alive. A good test suite should kill most mutants, as this indicates that it will likely discover real errors; otherwise, the suite needs to be improved with additional test cases. Conversely, a mutation operator producing mutants that are always killed is called *trivial* and is not useful.

Some works propose the use of mutation testing to evaluate transformation test suites [19], [20], and define different sets of transformation mutation operators. However, these operators do not consider real errors made by competent programmers, and their efficacy is unknown. Moreover, there are no proposals on how to automatically augment the test suite with models able to kill live mutants, and there is a lack of publicly available tools for transformation mutation testing.

This paper tackles these problems for ATL [21], one of the most widely used transformation languages, making the following contributions: (i) a set of mutation operators mimicking the most recurrent typing errors made by ATL developers; (ii) an evaluation of the efficacy of ATL mutation operators and test case generation techniques, based on six transformations developed by third parties; (iii) an automated technique to synthesize input models able to kill live mutants, improving the quality of a test suite; and (iv) an open-source framework to automate the mutation testing process for ATL.

Paper organization. Section II introduces mutation testing, and Section III customizes this technique for ATL. Section IV describes how to generate mutant-killing models. Section V reviews mutation operators for ATL, and proposes new operators mimicking ATL typing errors. Section VII evaluates the efficacy of the operators, several test generation approaches, and our technique to improve the quality of test cases. Section VIII discusses related works, and Section IX concludes.

II. Mutation Testing

Mutation testing [17], [18] is a technique to evaluate and improve the quality of test suites. It is based on seeding artificial flaws into the system under test founded on the belief that, if a test suite is able to detect the injected flaws, it will probably be good at detecting real errors.

Fig. 1 shows a typical mutation testing process. It assumes the availability of a set of mutation operators (label 1) that modify the program under test to emulate programming errors. The result is a set of mutants of the original program (label 2). Each mutant is executed against the test suite, and the result is

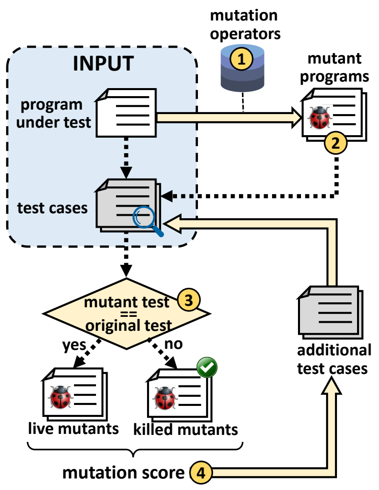


Fig. 1. Mutation testing process.

compared with the result of executing the test suite against the original program (label 3). If the results differ, the mutant is *killed* as the test suite has detected the error; otherwise, the mutant is *alive*. In languages like Java, the input to the mutation testing process is a Java program and a JUnit test suite that the program passes. This way, a mutant is killed upon finding some non-passing test case.

Some mutants can be indistinguishable from the original program. These are called *equivalent*. For instance, a mutant that adds "+0" to an arithmetical expression is an equivalent mutant because even though it is syntactically different from the original program, no test case can distinguish them.

The quality of a test suite is given by its *mutation score*. This is the fraction of the killed mutants divided by the total number of mutants, disregarding the equivalent ones. The higher the mutation score, the better the quality of the test suite. If the mutation score is deemed too low, the tester can add new test cases to kill some of the live mutants, increasing the mutation score and therefore the test suite quality. Hence, the automated synthesis of new test cases that kill live mutants is a challenge.

Mutation testing is computationally expensive due to the large number of potential mutants. Hence, good sets of mutation operators are required, producing mutants that are not easy to kill. For example, Offutt and Pan [22] apply selective mutation techniques that omit the most prevailing mutation operators, obtaining similar effectiveness with up to a 75% decrease in execution cost.

Determining mutant equivalence is non-computable in general. Since checking equivalence by hand is expensive, several heuristics to automate this process exist, e.g., based on constraints [23], on comparing the binaries of the original and mutant programs [24], or on automata language equivalence [25].

Another challenge in mutation testing is test data selection: identifying a set of test data that maximizes the number of mutants killed [26].

III. Testing ATL Transformations

In this section, we introduce ATL using a running example (Section III-A), and then adapt the mutation testing process described in Section II to ATL (Section III-B).

A. ATL by example

Model-to-model transformations are programs which take one or more input models and produce one or more output models. One of the most widely used model-to-model transformation languages is the ATLAS transformation language

```
1  create OUT : Relational from IN : Class
2
3  helper context Class!Attribute def: multiValued : Boolean =
4    if self.upperBound = −1 then true
5    else self.upperBound > 1 endif;
6  helper def: defaultType : String = 'Integer';
7
8  rule Class2Table {
9    from c : Class!Class ( not c.isAbstract )
10   to out : Relational!Table (
11        name <− c.name,
12        col <− Sequence{key}→union(c.att→select(e | not e.multiValued)),
13        key <− Set{key} ) ,
14      key : Relational!Column (
15        name <− 'objectId',
16        type <− Class!DataType.allInstances()→any(e |
17          e.name = thisModule.defaultType))
18 }
19
20 rule MultiValuedDataTypeAttribute2Column {
21   from a : Class!Attribute (
22        a.type.oclIsKindOf(Class!DataType) and a.multiValued )
23   to out : Relational!Table (
24        name <− a.owner.nameOrEmpty + '_' + a.name,
25        col <− Sequence {thisModule.createIdColumn(a.owner), value} ),
26      value : Relational!Column (
27        name <− a.name,
28        type <− a.type )
29 }
30
31 lazy rule createIdColumn {
32   from ne : Class!NamedElt
33   to key : Relational!Column (
34        name <− ne.name,
35        type <− Class!DataType.allInstances()→any(e |
36          e.name = thisModule.defaultType))
37 }
```

Listing 1. Excerpt of ATL program.

(ATL) [21]. Listing 1 shows an excerpt of an ATL program that we will use as a running example. It is based on a transformation available in the ATL zoo repository[1], which transforms object-oriented class models into relational ones.

An ATL program declares rules that select objects from the source model and create objects in the target model. Lines 8–18 in Listing 1 define the rule Class2Table which, given an object c of type Class (line 9), creates objects out of type Table and key of type Column. The type of the objects is prefixed by the meta-model names declared in line 2 (Class and Relational).

Rules can have *filters*, which are boolean expressions constraining the set of source objects to be transformed. For example, the filter in line 9 (not c.isAbstract) makes the rule applicable only to Class objects when isAbstract is false. The created target objects can initialize their attributes through *bindings* (e.g., lines 11–13 and 15–17). These are OCL expressions that may refer to objects read or created by the rule. Bindings for references (e.g., line 12) can assign objects of the source model to references in the target one. In such cases, an implicit binding resolution mechanism determines the target objects that were created from the given source objects, and assigns the target objects to the reference. For example, the binding in line 12 makes the union of the object key (of type Column) and the Attribute objects in collection c.att; the latter objects are resolved to Column objects (produced by MultiValuedDataTypeAttribute2Column) and assigned to col.

Rules Class2Table and MultiValuedDataTypeAttribute2Column are

[1]https://www.eclipse.org/atl/atlTransformations/#Class2Relational

matched rules. ATL also supports other types of rules, like *lazy rules*, which are executed only when explicitly called from other rules. For example, the lazy rule createIdColumn (lines 31–37) is called from line 25. Finally, *helpers* are auxiliary operations written in OCL. In the listing, the helper multiValuedOrFalse (lines 3–5) is defined on the context of class Attribute, while defaultType (line 6) is a global helper.

B. Mutation testing for ATL

Testing transformations is hard due to the difficulty of generating test models conformant to the input meta-model and providing an oracle [27]. ATL transformations are especially challenging to test because ATL is a dynamic language, so a transformation may crash at runtime even when the compiler does not signal any typing or syntactical error [4].

Mutation testing can help in detecting weak transformation test cases. However, its realization for a particular transformation language (e.g., ATL) requires providing support for the different steps of the process depicted in Fig. 1.

Mutation operators. There are several proposals of mutation operators for transformations. These include the ones by Troya et al. [20], Mottu et al. [19], [28], Khan et al. [29] and Sánchez et al. [4]. However, their effectiveness (i.e., their potential to produce hard-to-kill mutants) has not been assessed, and they do not mimic real developer errors.

Test suite. There are different ways to create input test models: (i) by hand, (ii) using a random instantiator, (iii) using meta-model coverage criteria [10], [11], (iv) using transformation coverage criteria [12], or (v) using coverage of transformation properties or contracts [13]–[15].

Testing oracle. An oracle must determine whether the transformation output is correct. This can be: (i) a total oracle consisting of the expected target model, or (ii) a partial oracle stating properties from the expected target model, e.g., in the form of contracts [13]–[15]. Most works that apply mutation testing to transformations use the transformation under test as a total oracle. Hence, given an input test model, they compare the target models produced by the original transformation and the mutant transformation to decide whether the mutant is killed. However, total oracles may give unrealistically high mutation scores, and may not be practical as it implies providing the output model for every possible input model.

Killing cause. A mutant transformation can be killed in three ways: (i) the mutant crashes when executed on a test model, (ii) the output model does not conform to the target meta-model, or (iii) the output model is not the expected one, which can be checked using a total or a partial oracle.

Example. Fig. 2 illustrates some mutation testing steps for ATL. It shows an application of the mutation operator “Collection filtering change with perturbation” (CFCP) [19], which modifies a collection or rule filter, to the example transformation. While operators are exhaustively applied to every possible location, the figure shows just one of such applications to the filter of rule Class2Table. The figure also shows the execution of the original and mutant transformations over a test model M_{in}. The original rule is applied to the non-abstract class named Female, but the mutant rule is not as it only applies to unnamed classes; hence, the executions yield different output models and the mutant is killed.

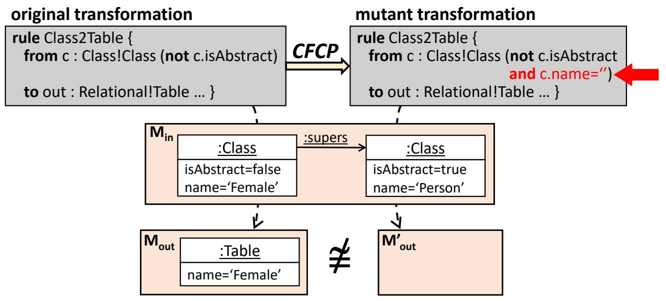


Fig. 2. Example of mutation testing process for ATL.

In the remainder of the paper, we first present a mechanism to help improving the quality of ATL transformation test suites. Then, we evaluate the effectiveness of different proposals (from the literature and this paper) for the realization of the main steps in the ATL mutation testing process.

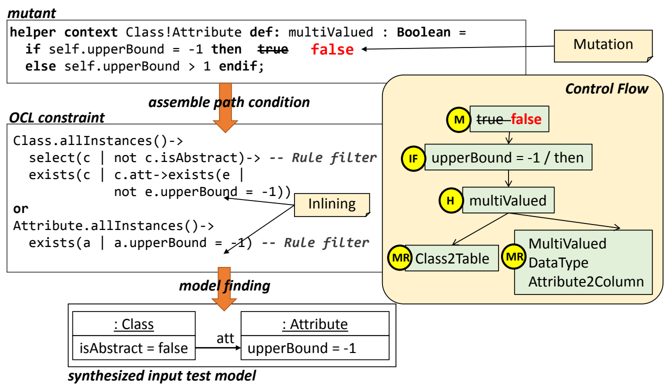
IV. Synthesizing Mutant-Killing Models

Transformation mutation operators perform a change in the AST of a transformation. If the test suite does not contain a model exercising the modified part, the mutant will not be detected and will remain alive, having a negative impact in the mutation score. In such cases, we propose enhancing the test suite with an input model that forces the execution of the mutated code, to increase the probability to kill the mutant.

For this purpose, we build an OCL path condition stating the constraints that an input model must satisfy to reach (i.e., to exercise) the modified part of the transformation. This OCL path condition and the input meta-model of the transformation are fed into a model finder, which outputs a model that conforms to the meta-model and satisfies the OCL condition, if such a model exists within the search bounds. Several model finders exist, like Alloy [30] or the USE Validator [31]. We use the latter in practice (see Section VI).

Example. Fig. 3 illustrates the synthesis of a test model to kill a mutant that has replaced a true literal by false. The location of the change is recorded to build the path condition of the modified expression.

To calculate the path condition, we use the Algorithm 1. This is similar to the algorithm presented in [4], but the way to assemble the condition is different (cf. Fig. 3). The input of the algorithm is the AST element modified by the mutation operator. The output is a slice of the transformation control flow graph, representing all the paths that start in matched rules and lead to the modified location (see Control Flow in Fig. 3, which contains any rule and helper dependency that is relevant to reach the modified location [4]). The algorithm recursively analyses the container of each AST element in the path to determine the kind of control flow that leads to it and create the corresponding node. In the example, the input of the algorithm *elem* is the modified boolean literal which will



```
helper context Class!Attribute def: multiValued : Boolean =
  if self.upperBound = -1 then  ~~true~~  false
  else self.upperBound > 1 endif;
```

```
Class.allInstances()->
  select(c | not c.isAbstract)-> -- Rule filter
  exists(c | c.att->exists(e |
           not e.upperBound = -1))
or
Attribute.allInstances()->
  exists(a | a.upperBound = -1) -- Rule filter
```

Fig. 3. Finding a test model that kills a mutant.

be the root node (line 2). Its parent is checked in function pathToControlFlow (lines 6–13); as it is an IfExp expression, the creation of the corresponding node is forwarded to pathToIfExp. In this case, the path flows to the *then* branch (line 16). The next container is a helper, which is handled by pathToHelper (lines 29–35). Hence, we obtain the *call sites* for the helper (line 32) to generate the corresponding *call nodes*. In the example of Fig. 3, there are two call sites for the multiValued helper which effectively split the path.

```
   Input: AST node of the element of interest ($elem$)
   Output: root node of the path condition tree ($node$)
1  def computePath (elem):
2      node = createPathRootNode(elem)
3      pathToControlFlow(elem, node)
4      return node
5  end
6  def pathToControlFlow (elem, node):
7      parent = getParentControlFlowStmt(elem)
8      switch parent do
9          case IfExp do pathToIfExp(parent, elem, node);
10         case Helper do pathToHelper(parent, node);
11         // ... similarly LazyRule, MatchedRule, IteratorExp
12     end
13 end
14 def pathToIfExp (ifExp, child, node):
15     branch = nil
16     if ifExp.thenExpr = child then
17         branch = true
18     else if ifExp.elseExpr = child then
19         branch = false
20     else
21         // the path comes from the condition
22         pathToControlFlow(ifExp, node)
23         return
24     end
25     ifNode = new IfNode(ifExp, branch)
26     node.addChildren(ifNode)
27     pathToControlFlow(ifExp, ifNode)
28 end
29 def pathToHelper (helper, node):
30     hNode = new HelperInvocationNode(helper)
31     node.addChildren(hNode)
32     foreach $pcall$ in helper.invocations do
33         pathToCall($pcall$, hNode)
34     end
35 end
36 def pathToCall (pcall, node):
37     cNode = new CallExpNode(pcall)
38     node.addChildren(cNode)
39     pathToControlFlow(pcall, cNode)
40 end
```

Algorithm 1: Computation of a path condition (from [4]).

The OCL path condition is built by traversing the path tree starting from the leaves, which in ATL represent matched rules. Each matched rule is mapped to an OCL expression that follows the pattern InputType.allInstances()→select(<filter>)→exists(<parent>).

The mapping for a helper call consists of inlining the helper nodes by replacing the helper parameters with the actual parameters in the call. In our example, we need to inline the *IfNode* condition used in the filters of the matched rules (self.upperBound = -1), and because the filter of MultiValuedDataTypeAttribute2Column calls the helper, we remove the last exists. Finally, we or-catenate the OCL expressions generated for each path, in this case corresponding to the paths starting in rules Class2Table and MultiValuedDataTypeAttribute2Column.

Our approach ensures that executing the mutant transformation with the generated model will exercise the mutated code, which is a necessary condition to kill the mutant. However, it does not guarantee that the original and mutant transformations will produce different output models, that is, our method is not sufficient. Hence, in some cases, executing the original and mutant transformations with the synthesized model yields indistinguishable output models and lets the mutant alive. Extending our approach to tackle this case is future work. Anyhow, Section VII-D will show that, in practice, our method frequently synthesizes test models that kill the mutants.

V. Mutation Operators for ATL

In this section, we revise mutation operators for model transformations proposed in the literature (Section V-A), and then propose a new set of operators emulating the most frequent typing errors in ATL transformations (Section V-B).

A. Mutation operators in the literature

a) Syntactic operators: There are different strategies to design mutation operators for a language. The first one is covering the language meta-model using *create-update-delete* (CUD) operations on its elements. Troya et al. [20] follow this approach to propose the 18 operators in Table I, which create, delete or update the main elements of the ATL meta-model (matched rules, in/out rule pattern elements, rule filters and bindings). The ATL meta-model coverage is not complete though, as for instance, helpers are not considered. We call this set of operators *syntactic* as they only consider the language abstract syntax.

TABLE I
Troya's operators [20]
(*syntactic set*).

Concept	Operator
Matched rule	addition
	deletion
	name change
In and out pattern element	addition
	deletion
	class change
	name change
Filter	addition
	deletion
	condition change
Binding	addition
	deletion
	value change
	feature change

b) Semantic operators: Another way to design operators is to mimic common faults a developer may incur. In this line, Mottu et al. [19], [28] propose the 10 mutation operators in Table II. These emulate errors in OCL navigation expressions (e.g., removing the last step in a navigation sequence), in filters of rules and collections (e.g., deleting a filter), and in creation operations (e.g., replacing the type of object created by a rule).

These operators are generic for any transformation language. They were designed based on the authors' experience in developing transformations, but not on empirical evidence of real errors. We call this set of operators *semantic* as they modify the transformation semantics.

TABLE II
MOTTU'S MUTATION OPERATORS [19] (*semantic set*).

Operator	**Explanation**
Navigation operators	
Relation to the same class change (RSCC)	Replaces the navigation through one association by another to the same class.
Relation to another class change (ROCC)	Replaces the navigation through one association by another to a different class.
Relation sequence modification with deletion (RSMD)	Removes the last step of a navigation sequence.
Relation sequence modification with addition (RSMA)	Adds a navigation step to a navigation sequence.
Filter operators	
Collection filtering change with perturbation (CFCP)	Modifies an existing filter, e.g., acting on a property or type of a class.
Collection filtering change with deletion (CFCD)	Deletes a filter.
Collection filtering change with addition (CFCA)	Adds a filter, e.g., returning an arbitrary element of a collection.
Creation operators	
Class compatible creation replacement (CCCR)	Replaces the creation of an object by the creation of an object of a compatible type.
Classes association creation deletion (CACD)	Omits the creation of a relation between two objects.
Classes association creation addition (CACA)	Adds the creation of a relation between two objects.

c) Typing operators: Sánchez et al. [4] define the 27 operators in Table III, whose aim is testing the anATLyzer ATL static analyser [4]. The operators inject typing errors (e.g., changing the return type of a helper) or faults causing runtime errors (e.g., deleting a parameter in a called rule). The design of these operators is guided by the ATL meta-model (though the set is more complete than Troya's) and gets inspiration from mutation operators for other languages (e.g., changing forAll by select). The operators introduce arbitrary typing errors, not common developer errors. Moreover, some operators are not exhaustive but yield different mutant classes. For example, the "binding creation" operator produces at most five mutants for each rule, adding to the rule either a duplicate binding, an attribute or a reference binding with correct value, a binding with an incorrect value, or a binding for a non-existing feature. We call them *typing* operators as they target typing errors.

d) Others: Khan et al. [29] propose 10 ATL operators, such as adding and removing bindings (present in the previous operator sets); changing a lazy rule to matched and vice versa; and some radical mutations like changing the mode of a transformation to refining, so that modifications to the input model are done in-place instead of producing an output model.

B. Mutation operators derived from errors in the ATL zoo

Mutation operators should emulate errors made by competent developers; however, the previous proposals are not based on empirical evidence of the errors that ATL developers do in practice. Hence, we next apply that principle to propose a set of mutation operators based on the most frequent typing errors found in the ATL zoo – a repository with 101 unique

TABLE III
SÁNCHEZ'S MUTATION OPERATORS FOR ATL [4] (*typing set*).

Type	**Targets**
Creation	binding source/target pattern element rule inheritance relation
Deletion	rule, helper binding source/target pattern element rule filter rule inheritance relation operation context formal/actual parameter in operation or called rule argument in operation invocation parameter in operation or called rule definition variable definition
Type modification	type of source/target pattern element helper context type, helper return type type of variable or collection parameter type of operation or called rule definition type parameter (e.g., oclIsKindOf(Type))
Feature name modification	navigation expression target of binding
Operation modification	predefined operator (e.g., and) or operation (e.g., size) collection operation (e.g., includes) iterator (e.g., exists, collect) operation/attribute helper invocation

ATL transformations – as reported in [4]. Most errors concern the creation of ill-formed models that do not conform to the target meta-model (44.9%), followed by navigation errors (23.8%), source typing errors (23.4%), rule conflicts (7.4%) and transformation integrity issues (0.5%).

Table IV shows the 5 most common typing errors in the zoo, its frequency, and the 7 mutation operators that we propose to emulate the errors. We call these operators *zoo operators*.

TABLE IV
ERRORS IN THE ATL ZOO [4] AND MUTATION OPERATORS (*zoo set*).

Error	**Freq.**	**Operator**
No binding for compulsory target feature	48.8%	Remove binding of compulsory feature (RBCF)
Invalid actual parameter type	11.9%	Replace helper call parameter (RHCP)
Feature access over possibly undefined receptor	11.22%	Remove enclosing conditional (REC), Add navigation after optional feature (ANAOF)
Feature found in subtype	3.75%	Replace feature access by subtype feature (RSF)
Binding possibly unresolved	3.7%	Restrict rule filter (RRF), Delete rule (DR)

The most common error, present in 44.8% of the transformations, is not giving a value to some mandatory field of a created target object, resulting in an ill-typed target model. To emulate this error, RBCF deletes a binding of a compulsory feature. This operator is a particular case of Troya and Sánchez's "binding deletion" operator (see Tables I and III).

The second most common error is invoking a helper passing a parameter whose type is incompatible with the type declared in the helper (11.9%). This error is due to the dynamic nature of ATL, which does not check type compatibility at compile-time. The corresponding operator (RHCP) changes an actual parameter by another of an incompatible type.

The most typical navigation error is accessing a feature over a possibly undefined receptor (11.22%). This occurs when a navigation expression traverses an optional reference without checking that the reference has a value (e.g., the expression

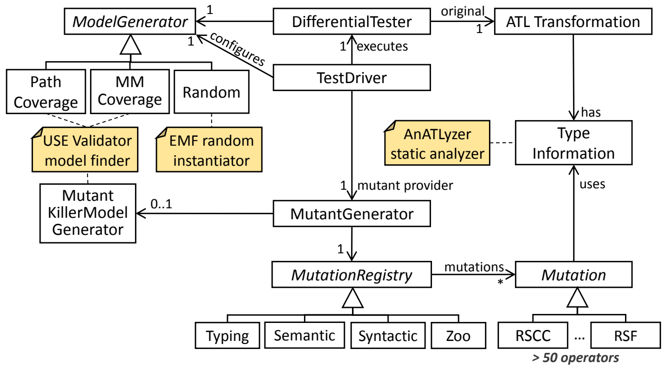


Fig. 4. Mutation testing framework for ATL.

obj.ref.feat, where ref is a monovalued reference with lower bound 0). If the reference is undefined, there may be a runtime error (a null pointer). We have designed two operators to emulate this error. The first one (REC) removes a possible enclosing "if" checking that the reference is defined. The second one (ANAOF) adds a navigation step to a navigation expression ending in an optional reference.

The last but one error (feature found in subtype, 3.75%) occurs when a feature is accessed from an object (e.g., obj.feat), and the feature is not defined in the type of the object (obj) but on a subtype, causing a runtime error. For this case, the operator RSF replaces a feature access from an object, by a feature declared in a subtype of the object's type.

Finally, the most common error related to rule conflicts is having bindings possibly unresolved (3.7%). This happens when a binding assigns source objects to a target reference, but no rule transforms e source objects. To emulate this error, the operator RRF modifies a rule filter to reduce the set of objects that the rule transforms, and the operator DR deletes a rule. Troya and Sánchez also consider the last operator.

VI. Tool Support

We have built a Java framework that automates mutation testing for ATL, available at https://github.com/jdelara/MDETesting. Fig. 4 shows a conceptual model with its elements. The framework includes all mutation operators introduced in Section V, organized in types (subclasses of MutationRegistry). New mutation operators can be added by extending the base class Mutation. The operators can use the transformation typing information (e.g., which rules resolve a binding) provided by the anATLyzer static analyser [4].

The framework supports three techniques for the generation of models: random instantiation using the EMF random instantiator[2] programmatically, meta-model coverage, and transformation path coverage. The latter two techniques, as well as a facility to generate mutant-killing test models (cf. Section IV), rely on the USE Validator model finder [31].

For mutation testing, the framework provides a test driver and implements a differential tester to compare the output of the original and mutant transformations. Metrics of the mutation process (e.g., mutation score, killed and live mutants, etc.) are persisted in an XML file.

[2]https://github.com/atlanmod/mondo-atlzoo-benchmark/tree/master/fr.inria.atlanmod.instantiator

VII. Evaluation

This section evaluates the effectiveness of the different steps in the mutation testing process for ATL. Since mutation testing is time consuming, an effective process must consider mutation operators that produce mutants difficult to kill and discard operators that yield trivial mutants. The quality of the input test suite is given by the mutation score (efficacy), and when two test suites have the same mutation score, the smaller one is preferred as it implies less testing time (efficiency). To analyse these issues, we address the following research questions:

RQ1 Which operators produce the hardest-to-kill mutants?
RQ1.1 Are mutants mimicking the errors found in the ATL zoo harder to kill than other mutants?
RQ2 Which test model generation technique produces higher quality test cases?
RQ3 How effective is our technique to generate models that kill live mutants?

For this purpose, Section VII-A first describes the evaluation setup. Then, Section VII-B analyses how hard to kill are the mutants produced by different mutation operators (**RQ1** and **RQ1.1**). Section VII-C tackles **RQ2** for three common test case generation techniques: random, based on meta-model coverage, and based on transformation path coverage. Finally, Section VII-D evaluates the effectiveness of our technique for generating models that kill live mutants (**RQ3**).

A. Evaluation setup

Mutation testing requires starting from programs syntactically correct and without typing errors. Hence, for the evaluation, we selected six ATL transformations from the ATL zoo having no or few typing errors, in the latter case fixing them by hand. These transformations provide a wide coverage of the ATL constructs, including matched and lazy rules, abstract rules, helpers, rules with several input or output elements, and so on. Table V shows some metrics for the transformations.

TABLE V
Transformations used in the evaluation.

transf.	in/out mm classes	matched/ lazy rules	abstract rules	helpers	avg in/out rule elems	bindings
class2table	6 / 5	7 / 1	0	3	1 / 1.3	20
uml2intalio	247 / 19	9 / 0	0	5	1 / 1.1	14
bt2db	21 / 8	9 / 0	0	4	1 / 2.2	25
cpl2spl	33 / 77	14 / 1	0	6	1 / 3.2	73
hsm2fsm	7 / 7	7 / 0	0	0	1.9 / 1	19
uml2er	4 / 8	8 / 0	3	0	1 / 1	5

We performed the experiments using the tool presented in Section VI, and considering all mutation operators of the *syntactic*, *semantic*, *typing* and *zoo* sets. We excluded the operators by Khan et al. as these are either included in other sets, their effects are covered by other operators (e.g., setting a rule to lazy is equivalent to deleting it), or always produce a runtime error (e.g., a refining transformation with different source and target meta-models crashes).

TABLE VI
KILLING STATISTICS OF MUTATION OPERATORS

Identification			Applicability			Resilience					Stubbornness		
Id	**Type**	**Operator name**	**#Mutants**	**#Trafo**	**Avg occ.**	**Total**	**%**	**Oracle**	**Crash**	**Orc. Norm.**	**#Models**	**Avg.**	**% Killing**
0	syn	InPatternElementDeletion	0	0	0.00	-	-	-	-	-	-	-	-
1	sem	CACA	14	2	7.00	14	100.00	78.57	21.43	100.00	553	98.80	17.87
2	zoo	ReplaceFeatureAccessBySubtypeFeature (RSF)	48	3	16.00	48	100.00	31.25	68.75	100.00	352	13.96	3.97
3	typ	ParentRuleDeletionMutator	21	1	21.00	21	100.00	61.90	38.10	100.00	289	10.76	3.72
4	typ	VariableModificationMutator	48	1	48.00	48	100.00	100.00	0.00	100.00	1812	37.75	2.08
5	sem	RSMD	72	3	24.00	72	100.00	84.72	15.28	100.00	1245	24.16	1.94
6	typ	CollectionModificationMutator	54	3	18.00	54	100.00	100.00	0.00	100.00	1951	37.21	1.91
7	typ	HelperContextModificationMutator	65	4	16.25	65	100.00	40.00	60.00	100.00	639	11.66	1.82
8	zoo	RemoveEnclosingConditional (REC)	109	1	109.00	109	100.00	100.00	0.00	100.00	3617	47.91	1.32
9	syn	FilterAddition	117	5	23.40	117	100.00	76.07	23.93	100.00	5418	66.94	1.24
10	sem	RSCC	75	2	37.50	75	100.00	80.00	20.00	100.00	4603	43.58	0.95
11	zoo	ReplaceHelperCallParameter (RHCP)	162	3	54.00	162	100.00	100.00	0.00	100.00	5648	52.38	0.93
12	typ	IteratorModificationMutator	179	4	44.75	179	100.00	60.34	39.66	100.00	2778	18.01	0.65
13	typ	PrimitiveValueModificationMutator	197	4	49.25	197	100.00	100.00	0.00	100.00	7138	36.50	0.51
14	typ	HelperOperationModificationMutator	380	4	95.00	380	100.00	79.21	20.79	100.00	8676	24.63	0.28
15	typ	CollectionOperationModificationMutator	407	4	101.75	407	100.00	76.17	23.83	100.00	9535	24.14	0.25
16	typ	ArgumentModificationMutator	486	5	97.20	486	100.00	88.07	11.93	100.00	13060	26.42	0.20
17	syn	MatchedRuleAddition	1367	6	227.83	1367	100.00	46.89	53.11	100.00	18218	15.99	0.09
18	typ,syn	InElementCreationMutator	3818	6	365.67	3818	100.00	82.48	17.52	100.00	133931	43.29	0.03
19	syn	BindingValueChange	279	6	46.50	278	99.64	92.09	7.91	99.61	16426	43.28	0.26
20	syn	BindingFeatureChange	941	6	156.83	938	99.68	71.22	28.78	99.55	28653	44.98	0.16
21	typ,syn	BindingDeletionMutator	724	6	72.67	720	99.45	100.00	0.00	99.45	36256	46.92	0.13
22	sem	ROCC	861	5	172.20	858	99.65	58.62	41.38	99.41	14129	26.15	0.19
23	sem	CACD	268	6	44.67	266	99.25	100.00	0.00	99.25	12626	58.13	0.46
24	zoo	RemoveBindingOfCompulsoryFeature (RBCF)	260	5	52.00	258	99.23	100.00	0.00	99.23	17192	62.08	0.36
25	syn	InPatternElementClassChange	1626	6	271.00	1615	99.32	62.85	37.15	98.93	48096	26.54	0.06
26	syn	BindingAddition	598	6	99.67	591	98.83	93.06	6.94	98.74	30769	85.34	0.28
27	sem	CFCP	1001	6	166.83	991	99.00	77.70	22.30	98.72	36744	58.56	0.16
28	zoo	RestrictRuleFilter (RRF)	730	6	121.67	720	98.63	78.89	21.11	98.27	33976	52.72	0.16
29	zoo	DeleteRule (DR)	179	6	29.83	176	98.32	93.18	6.82	98.20	10694	74.33	0.70
30	typ	RuleDeletionMutator	178	6	29.67	175	98.31	93.14	6.86	98.19	5322	29.19	0.55
31	typ,syn	OutElementCreationMutator	3327	6	343.00	3269	98.26	93.51	6.49	98.14	149822	56.26	0.04
32	syn	MatchedRuleDeletion	113	6	18.83	111	98.23	91.89	8.11	98.08	7670	55.77	0.73
33	typ	ArgumentDeletionMutator	306	5	61.20	302	98.69	67.55	32.45	98.08	4712	17.39	0.37
34	syn	InPatternElementNameChange	145	6	24.17	142	97.93	83.10	16.90	97.52	14044	51.86	0.37
35	typ	BindingModificationMutator	913	6	152.17	893	97.81	84.32	15.68	97.41	15831	16.73	0.11
36	typ	PredefinedOperationModificationMutator	1016	6	169.33	995	97.93	75.68	24.32	97.29	19265	18.50	0.10
37	syn	OutPatternElementClassChange	1244	6	207.33	1222	98.23	63.91	36.09	97.26	29111	28.50	0.10
38	sem	RSMA	822	5	164.40	810	98.54	48.77	51.23	97.05	11404	18.85	0.17
39	syn	OutPatternElementNameChange	207	6	34.50	201	97.10	92.54	7.46	96.88	11472	41.11	0.36
40	typ	NavigationModificationMutator	859	6	143.17	836	97.32	80.14	19.86	96.68	19547	20.63	0.11
41	sem	CCCR	1194	6	199.00	1165	97.57	68.76	31.24	96.51	22265	27.08	0.12
42	typ	OperatorModificationMutator	1092	5	218.40	1059	96.98	67.80	32.20	95.61	20667	18.87	0.09
43	typ	OutElementModificationMutator	439	6	73.17	423	96.36	78.01	21.99	95.38	7675	15.55	0.20
44	typ	BindingCreationMutator	570	6	95.00	543	95.26	90.61	9.39	94.80	10488	19.65	0.19
45	typ	ParentRuleModificationMutator	1424	6	237.33	1368	96.07	70.83	29.17	94.54	32272	22.19	0.07
46	syn	FilterConditionChange	206	6	34.33	196	95.15	83.67	16.33	94.25	13677	48.57	0.36
47	sem	CFCD	100	6	16.67	96	96.00	67.71	32.29	94.20	2980	33.10	1.11
48	typ	InElementModificationMutator	266	6	44.33	251	94.36	68.13	31.87	91.94	3705	11.53	0.31
49	typ	HelperReturnModificationMutator	966	6	161.00	882	91.30	100.00	0.00	91.30	32304	26.93	0.08
50	typ,syn	FilterDeletionMutator	167	6	15.50	157	94.01	65.61	34.39	91.15	5091	28.84	0.57
51	syn	MatchedRuleNameChange	120	6	20.00	109	90.83	100.00	0.00	90.83	8319	31.58	0.38
52	typ	HelperDeletionMutator	780	6	130.00	704	90.26	95.74	4.26	89.87	24592	29.10	0.12
53	typ	ParameterDeletionMutator	513	6	85.50	459	89.47	100.00	0.00	89.47	16902	24.76	0.15
54	typ	ParameterModificationMutator	570	6	95.00	510	89.47	100.00	0.00	89.47	18780	24.76	0.13
55	zoo	AddNavigationAfterOptionalFeature (ANAOF)	44	3	14.67	39	88.64	64.10	35.90	83.33	560	5.83	1.04

A repository with all models, transformations, and the raw data of the results is available at http://miso.es/ATLmut.

B. Evaluation of mutation operators (RQ1, RQ1.1)

To evaluate the ease to killing the mutants produced by each operator, we created a test suite per transformation with test models generated using three techniques: random instantiation, meta-model coverage, and transformation path coverage. For random instantiation, we fixed a number of 50 test models per transformation, while the number of test models generated with the other techniques ranged from 1 to 710. The size of the resulting test suites ranged from 82 to 784 test models.

Table VI shows the statistics obtained for each mutation operator, structured in four blocks:

- The first block includes the id, type (*syn*tactic, *sem*antic, *typ*ing, *zoo*) and name of each operator. Some operators (18, 21, 31, 50) belong to two different sets (*typ*, *syn*). Others are similar (e.g., 29, 30, 32) but have slightly different application conditions.
- The second block shows *applicability* metrics: the number of mutants generated by each operator; the number of transformations that produced some mutant; and the average number of mutants generated per transformation.

- The third block gives metrics related to mutant *resilience*: the number and percentage of killed mutants; the way in which the mutants were killed, either because the original and mutant transformations produced different output models (oracle) or by a runtime error of the mutant execution (crash); and the percentage of killed mutants considering only those killed by the oracle (orc. norm.).
- A last block details metrics about mutant *stubbornness*: the total and average number of models executed with the mutants produced by each operator; and the percentage of models that killed the mutants. We consider that a mutant is stubborn if it is killed by very few tests [32].

Overall, we tested more than 32 000 mutant transformations, performing over one million transformation executions.

Regarding applicability, many of the operators (>61%) were applicable to all transformations. However, three applied only to one transformation: ParentRuleDeletionMutator with id 3 (as only one transformation had rule inheritance), VariableModificationMutator with id 4, and RemoveEnclosingConditional with id 8; this means that these operators may be useless in many scenarios. Operator InPatternElementDeletion with id 0 generated no valid mutants because even though one transformation had rules with several input elements, deleting any of them produced non-compiling mutants due to dangling references from the rule filter or some binding to the deleted elements.

Looking at block Resilience, we observe that the percentage of killed mutants is high (>88% in column %). The table also distinguishes the percentage of killed mutants due to discrepancies in the oracle function or due to runtime errors. Crash-prone mutants are the least helpful as they can be detected more easily, e.g., using a static analyser like anATLyzer [4][3]. Thus, we focus on mutants killed by the oracle, and order the table in descending order of the percentage of mutants killed in this way. Hence, the mutants on top of the table are the easiest to kill, and those at the bottom are the most resilient. Specifically, the mutants generated by the first 18 operators (excluding InPatternElementDeletion) were killed by some test suite. In contrast, only 83.33% of the mutants produced by AddNavigationAfterOptionalFeature were killed using the oracle.

An interesting case concerns operators BindingDeletionMutator (id 21), which deletes any arbitrary binding, and RemoveBindingOfCompulsoryFeature (id 24), which only deletes bindings of target mandatory features. While both have similar resilience (99.45% and 99.23%), the latter produces one third less mutants (724 vs 260); therefore, using the latter is more efficient without loss of efficacy. Similarly, MatchedRuleDeletion (id 32) should be slightly preferable to the other two rule deletion operators (ids 29 and 30).

To analyse mutant stubbornness, the graph in Fig. 5 depicts the average of oracle-killed mutants of each kind, against the percentage of test models killing them. Each point in the graph represents a type of mutant with its id on top, and a different shape and colour depending on the set where it belongs to. Mutants on the left-bottom are harder to kill

[3]anATLyzer is reported to have $> 96\%$ precision and $> 98\%$ recall.

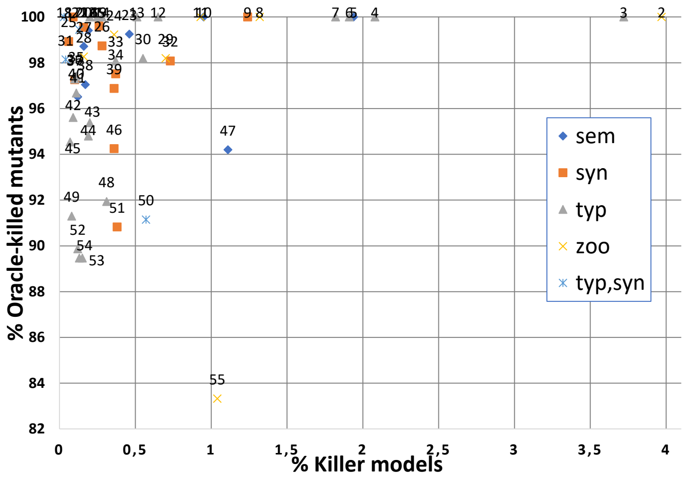


Fig. 5. Mutant stubbornness: % of killed mutants vs % of killer test models.

(lower ratio of killed mutants and killer test models), while those on the upper-right are easier to kill. The graph excludes CACA because the percentage of test models that kill its mutants is much higher than the rest (>17%). In general, most operators generate stubborn mutants as few test models kill them. Consistently with the last block in Table VI, the mutants in the left-bottom have ids 52, 53, 54 and 55 (typing and zoo operators). Operator 55 produces a low percentage of oracle-killed mutants, but the percentage of killer models is somewhat high. Conversely, the easiest-to-kill mutants have ids 1–7 (two semantic, four typing and one zoo operators).

For space constraints, Table VI shows the aggregated results obtained for the three test model generation techniques. If we expand the results, we observe that oracle-killed resilience is 100% for all operators when using path-based models, 80% of the operators when using meta-model coverage, and 36% of the operators for random models. Operators with similar resilience in all test generation techniques, like CACA, have generally lower stubbornness when using path-based models. The disaggregated data are available in the online repository.

Overall, we can answer **RQ1** saying that some typing operators (especially 52, 53, 54) produce the hardest-to-kill mutants. The zoo operator 55 is hard but crash-prone, and when killed, it is killed by many models. Moreover, its applicability is average. On the other hand, we answer **Q1.1** negatively. If we look at the zoo operators, AddNavigationAfterOptionalFeature (id 55) is one of the hardest to kill, but three other zoo operators (ids 2, 8 and 11) are among the easiest to kill. The remaining three (ids 24, 28 and 29) have intermediate values.

C. *Evaluation of test case generation techniques (RQ2)*

We used three approaches to generate the test models: random, meta-model coverage, and transformation path coverage. In the first case, we generated 50 random models of size 100 for each transformation. For meta-model coverage, we synthesized models covering all input meta-model classes and features as described in [33]. This approach ensures that every meta-model element appears in some model. For the last approach, we automatically extracted the path condition

TABLE VII
EVALUATION OF THE TEST CASE GENERATION TECHNIQUES USING THE MUTATION SCORE

	Meta-model					**Path**					**Random**				
Test case	#M.	Typ	Sem	Syn	Zoo	#M.	Typ	Sem	Syn	Zoo	#M.	Typ	Sem	Syn	Zoo
class2table	62	69.98	80.73	100.00	97.14	27	77.53	75.23	100.00	97.14	50	67.15	58.33	59.86	97.14
uml2intalio	200	87.44	31.68	45.54	30.33	18	82.47	71.66	89.31	95.08	50	65.64	56.71	48.36	23.58
bt2db	50	81.21	81.44	84.50	97.04	1	84.85	87.37	90.56	99.26	50	64.18	81.05	69.01	95.56
cpl2spl	50	98.71	68.52	86.17	100.00	92	98.14	98.60	96.33	100.00	50	99.22	98.33	95.74	100.00
hsm2fsm	710	73.48	82.05	73.37	92.16	24	76.83	84.62	81.41	92.16	50	15.67	65.81	22.22	78.00
uml2er	26	75.91	72.84	60.96	87.88	6	82.70	70.37	60.56	87.88	50	21.60	19.28	23.51	45.45

of each independent transformation path, similar to Section IV, and used a model finder for model generation.

We need to make a few precisions. First, since the uml2intalio transformation only deals with a small part of the UML meta-model, we generated its test models using the effective meta-model, i.e., the slice of the UML meta-model touched by the transformation. Second, to avoid excessive execution times, we restricted the number of test models generated using meta-model coverage to 50 for transformations bt2db and cpl2spl.

In order to analyse the quality of the test suites obtained by the different approaches, we calculated their mutation score for each transformation. Table VII summarizes the results. We disregard the mutants killed by crashes because, as abovementioned, these can be detected using a static analyser.

At a first glance, it is easy to see that transformation path coverage produced fewer models, most of the times, an order of magnitude less. Still, this technique yielded the highest mutation score more frequently (19 out of 24 cases). The test suites made of random models were the least performant, even if those models were much larger than those populated by the other approaches (random produced models with 108.13 objects on average, meta-model coverage with 7.22 objects and path coverage with 2.55 objects). Hence, having larger test models does not ensure higher-quality test suites. Actually, as the test model size has an impact in the transformation execution time, smaller models are preferable whenever possible.

Overall, from these results, we can answer **RQ2** by saying that the most effective test model generation technique among the three evaluated is transformation path coverage.

D. Evaluation of test case improvement technique (RQ3)

The last experiment explores the efficacy of our algorithm to generate test models that kill an intended mutant. For this purpose, we instrumented the mutant generator to synthesize, together with each mutant, an associated test model targeting the modified location, as explained in Section IV. Then, we executed the mutant and the original transformation against the test model, and compared the output.

Table VIII reports the results. Overall, our technique generated models that killed the mutant in 85% of the cases. The percentage of success (i.e., of killed mutants) was higher than 90% in 12 cases out of 24, and higher than 80% in 17 cases. In two cases, the test models killed all mutants. The worst results were obtained for the uml2er transformation, as the synthesized models killed between 43.75% and 59.48% of the mutants.

TABLE VIII
EFFICACY OF OUR TECHNIQUE TO GENERATE MUTANT-KILLER MODELS

	Typ		**Sem**		**Syn**		**Zoo**	
Test case	#M.	%	#M.	%	#M.	%	#M.	%
class2table	266	93.94	58	96.55	187	87.57	15	100.00
uml2intalio	2200	97.14	550	83.82	4045	97.11	60	81.67
bt2db	151	93.10	32	100.00	175	94.08	21	95.24
cpl2spl	3161	82.76	649	71.19	5993	78.66	65	73.85
hsm2fsm	382	92.25	75	94.67	363	86.76	41	92.68
uml2er	153	59.48	42	54.76	143	58.04	16	43.75

Hence, we answer **RQ3** by saying that, while our method only considers reachability constraints, it provides good results. In the future, we will improve the method by considering other conditions which may depend on the mutation operator.

E. Discussion and threats to validity

For ATL, a general recommendation derived from our evaluation is using test models generated by transformation path coverage, as they produce smaller test suites without losing efficacy. For mutation testing, none of the analysed operator sets is more effective than the others, but all of them contain operators easy to kill, and others which are harder to kill. Interestingly, the hardest-to-kill operator belongs to the zoo set, and it emulates a recurrent error committed by ATL developers, which is accessing a feature over an undefined receptor object. However, two operators from the same set are among the easiest to kill, one of them emulating the same error in another way. As many transformations in the ATL zoo contain these typing errors, we conclude that transformations are not tested up to standards for other kinds of software.

There are several threats to validity. First, we did not discard equivalent mutants for computing the mutation score, as we are not aware of any method for their automated detection, and their manual identification was not feasible due to the high number of generated mutants ($>$32 000). Retaining equivalent mutants may result in slightly lower mutation scores. To mitigate this effect, we used six different transformations.

Another issue with the mutation score is that we used a total oracle due to the difficulty of finding publicly available partial oracles. However, total oracles may be unfeasible in practice, and may produce a mutation score higher than partial oracles.

We were able to reuse an existing implementation of the typing operators. However, we had to encode all syntactic and semantic operators based on their descriptions in the literature, and so, another threat is that we may have misinterpreted some details producing wrong implementations.

Finally, we have used six third-party transformations. This number is higher than other works on transformation mutation analysis [28], [34], which typically use one, but considering more transformations would provide stronger evidence.

VIII. Related Work

Next, we survey works on analysis of mutation operators, automated test case generation, and mutation testing in MDE.

Analysing efficacy of operators. Mutation testing is suited to measure test efficacy, but its computational cost is high [35]. This is why some works aim to identify *sufficient operator sets* [22], [36], [37]. These are subsets of operators that provide a sufficiently accurate measurement of overall mutation adequacy while reducing the computational cost. Given a test suite, a subset of the operators is sufficient if a mutation score of 1 on this subset is likely to produce a high mutation score on the whole set. This subset can be heuristically computed based on statistics [37]. Instead, in this work, our interest is in analysing the ease of killing mutants created by different operators, and to evaluate test model generation techniques.

Several works analyse which mutation operators are hard to kill [38], [39]. Visser [38] identifies three factors: reachability, the operator itself, and the oracle. Using symbolic execution for simple programs and selected mutation operators, he was able to assert the percentage of all possible inputs that kill a mutant. He argues that, once the fault is reached, most mutants seem easy to kill, hence confirming the rationale of our mutant-killing model generation strategy.

Mutation-based test case generation. Some researchers have exploited mutation analysis to generate test cases. Similar to us, Godzilla [40] relies on constraint solving. This work was later optimized with dynamic domain reduction techniques to cover complex input data [41]. Dynamic symbolic execution overcomes the limitations of symbolic execution, and can be used in directed automated random testing [42].

Instead of symbolic execution, other methods rely on search. Botaci [43] uses constraints derived from program mutants to guide metaheuristic search for test data generation. This is used in [44] to derive test data using the ant colony optimization method. μTEST [45] uses genetic algorithms to construct method call sequences that are effective in detecting mutants, using synthesized oracles. Instead of targeting one mutant at a time, EvoSuite [46] uses a search-based approach to generate a test for all mutants at the same time. Finally, SHOM [47] combines dynamic symbolic execution and search-based software testing to generate input data from mutants generated using high-order operators.

All these works focus on imperative languages like C, Java and Fortran, and their test data have numeric domains. As a difference, we target ATL, which is a rule-based language, and our test data are models having a more complex structure (this is why we rely on model finders).

Mutation testing in MDE. Some works have proposed applying mutation testing to model transformations and define sets of mutation operators [4], [19], [20], [29]. However, none of them analyse the effectiveness of the operators.

Other works define systematic techniques for creating mutation operators for a specific language like ATL [20], [29] or for arbitrary languages defined by a meta-model [48], [49]. While we could have used these works to implement the operators, we resorted to our own framework as some operators required using the typing information from a static analyser.

Mutation techniques have been applied to the generation of input test models for testing model transformations [28], [50]. In [50], mutations (modelled as graph transformation rules) are used to create model variants. In [28], the authors devise heuristics to synthesize test models that have a good chance to kill live mutants generated by some of Mottu's operators. However, while that approach does not generalize to arbitrary operators, our approach based on path conditions does, and we empirically measure the effectiveness of the method. To our knowledge, our approach based on path condition constraints is novel in model transformation testing.

In [34], the authors generate test models based on the input meta-model footprint, and show that this is more effective than meta-model coverage. For this, they use mutation analysis over one transformation and Mottu's operators. In this work, we have more transformations, more operators, more test model generation techniques, and show that transformation coverage produces fewer models and achieves good mutation scores.

The iterative model generation technique proposed in [11] allows synthesizing a set of diverse models from different equivalent classes w.r.t. their graph shape. The technique can be used to generate test models for DSLs, and it produces more efficient test suites than using models not created for testing or created with Alloy for different symmetry breaking predicates. The technique only considers the DSL meta-model for model generation. Instead, our algorithm to generate mutant-killing models also needs to consider the semantics of the ATL program under test and the path to reach a mutated location.

IX. Conclusions and Future Work

In this paper, we have paved the way towards an effective mutation testing process for ATL: we have proposed a set of mutation operators emulating real errors, measured the efficacy of these and other operators proposed in the literature, and measured the efficacy of common test model generation techniques. We have also proposed a method to synthesize mutant-killing models, and developed tool support.

In the future, we plan to evaluate the test model generation methods and the mutation operators using partial oracles. We are working on methods to detect mutant equivalence and to improve the generation of mutant-killing models. Finally, we plan to derive sufficient mutant sets as in [37], and to generalize the approach to other transformation languages.

Acknowledgment

Work funded by the Spanish Ministry of Science (RTI2018-095255-B-I00, TIN2015-73968-JIN AEI/FEDER/UE), the Madrid Region (S2018/TCS-4314), and a Ramón y Cajal 2017 grant.

References

[1] S. Sendall and W. Kozaczynski, "Model transformation: The heart and soul of model-driven software development," *IEEE Software*, vol. 20, no. 5, pp. 42–45, 2003.

[2] L. A. Rahim and J. Whittle, "A survey of approaches for verifying model transformations," *SoSyM*, vol. 14, no. 2, pp. 1003–1028, 2015.

[3] L. Burgueño, J. Troya, M. Wimmer, and A. Vallecillo, "Static fault localization in model transformations," *IEEE Trans. Software Eng.*, vol. 41, no. 5, pp. 490–506, 2015.

[4] J. Sánchez Cuadrado, E. Guerra, and J. de Lara, "Static analysis of model transformations," *IEEE Trans. Software Eng.*, vol. 43, no. 9, pp. 868–897, 2017.

[5] F. Rabbi, L. M. Kristensen, and Y. Lamo, "Static analysis of conformance preserving model transformation rules," in *MODELSWARD*. SciTePress, 2018, pp. 152–162.

[6] J. Troya, S. Segura, J. A. Parejo, and A. R. Cortés, "Spectrum-based fault localization in model transformations," *ACM Trans. Softw. Eng. Methodol.*, vol. 27, no. 3, pp. 13:1–13:50, 2018.

[7] R. Clarisó, J. Cabot, E. Guerra, and J. de Lara, "Backwards reasoning for model transformations: Method and applications," *Journal of Systems and Software*, vol. 116, pp. 113–132, 2016.

[8] J. Sánchez Cuadrado, E. Guerra, J. de Lara, R. Clarisó, and J. Cabot, "Translating target to source constraints in model-to-model transformations," in *MODELS*. IEEE Computer Society, 2017, pp. 12–22.

[9] H. Ehrig, C. Ermel, U. Golas, and F. Hermann, *Graph and Model Transformation - General Framework and Applications*, ser. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2015.

[10] E. Brottier, F. Fleurey, J. Steel, B. Baudry, and Y. L. Traon, "Metamodel-based test generation for model transformations: an algorithm and a tool," in *ISSRE*. IEEE Computer Society, 2006, pp. 85–94.

[11] O. Semeráth and D. Varró, "Iterative generation of diverse models for testing specifications of DSL tools," in *FASE*, ser. LNCS, vol. 10802. Springer, 2018, pp. 227–245.

[12] C. A. González and J. Cabot, "Atltest: A white-box test generation approach for ATL transformations," in *MODELS*, ser. LNCS, vol. 7590. Springer, 2012, pp. 449–464.

[13] E. Guerra, J. de Lara, M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck, and W. Schwinger, "Automated verification of model transformations based on visual contracts," *Autom. Softw. Eng.*, vol. 20, no. 1, pp. 5–46, 2013.

[14] E. Guerra and M. Soeken, "Specification-driven model transformation testing," *SoSyM*, vol. 14, no. 2, pp. 623–644, 2015.

[15] M. Gogolla and A. Vallecillo, "*Tract*able model transformation testing," in *ECMFA*, ser. LNCS, vol. 6698. Springer, 2011, pp. 221–235.

[16] Z. Cheng and M. Tisi, "A deductive approach for fault localization in ATL model transformations," in *FASE*, ser. LNCS, vol. 10202. Springer, 2017, pp. 300–317.

[17] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *IEEE Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[18] R. G. Hamlet, "Testing programs with the aid of a compiler," *IEEE Trans. Software Eng.*, vol. 3, no. 4, pp. 279–290, 1977.

[19] J. Mottu, B. Baudry, and Y. L. Traon, "Mutation analysis testing for model transformations," in *ECMDA-FA*, ser. LNCS, vol. 4066. Springer, 2006, pp. 376–390.

[20] J. Troya, A. Bergmayr, L. Burgueno, and M. Wimmer, "Towards systematic mutations for and with ATL model transformations," in *ICST Workshops*, 2015, pp. 1–10.

[21] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "ATL: A model transformation tool," *Sci. Comput. Program.*, vol. 72, no. 1-2, pp. 31–39, 2008.

[22] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators," *ACM Trans. Softw. Eng. Methodol.*, vol. 5, no. 2, pp. 99–118, 1996.

[23] A. J. Offutt and J. Pan, "Automatically detecting equivalent mutants and infeasible paths," *Softw. Test., Verif. Reliab.*, vol. 7, no. 3, pp. 165–192, 1997.

[24] M. Kintis, M. Papadakis, Y. Jia, N. Malevris, Y. L. Traon, and M. Harman, "Detecting trivial mutant equivalences via compiler optimisations," *IEEE Trans. Software Eng.*, vol. 44, no. 4, pp. 308–333, 2018.

[25] X. Devroey, G. Perrouin, M. Papadakis, A. Legay, P. Schobbens, and P. Heymans, "Model-based mutant equivalence detection using automata language equivalence and simulations," *Journal of Systems and Software*, vol. 141, pp. 1–15, 2018.

[26] F. C. M. Souza, M. Papadakis, V. H. S. Durelli, and M. E. Delamaro, "Test data generation techniques for mutation testing: A systematic mapping," in *CIbSE*. Curran Associates, 2014, pp. 419–432.

[27] B. Baudry, S. Ghosh, F. Fleurey, R. B. France, Y. L. Traon, and J. Mottu, "Barriers to systematic model transformation testing," *Commun. ACM*, vol. 53, no. 6, pp. 139–143, 2010.

[28] V. Aranega, J.-M. Mottu, A. Etien, T. Degueule, B. Baudry, and J.-L. Dekeyser, "Towards an automation of the mutation analysis dedicated to model transformation," *Softw. Test., Verif. Reliab.*, vol. 25, no. 5-7, pp. 653–683, 2014.

[29] Y. Khan and J. Hassine, "Mutation operators for the atlas transformation language," in *ICST Workshops*. IEEE Computer Society, 2013, pp. 43–52.

[30] D. Jackson, *Software Abstractions - Logic, Language, and Analysis*. London, England: MIT Press, 2006, see also http://alloy.mit.edu/.

[31] M. Kuhlmann and M. Gogolla, "From UML and OCL to relational logic and back," in *MODELS*, ser. LNCS, vol. 7590. Springer, 2012, pp. 415–431.

[32] L. Gonzalez-Hernandez, B. Lindström, J. Offutt, S. F. Andler, P. Potena, and M. Bohlin, "Using mutant stubbornness to create minimal and prioritized test sets," in *QRS*. IEEE, 2018, pp. 446–457.

[33] F. Fleurey, B. Baudry, P. Muller, and Y. L. Traon, "Qualifying input test data for model transformations," *SoSyM*, vol. 8, no. 2, pp. 185–203, 2009.

[34] J. Mottu, S. Sen, M. Tisi, and J. Cabot, "Static analysis of model transformations for effective test generation," in *ISSRE*. IEEE Computer Society, 2012, pp. 291–300.

[35] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *ICSE*. ACM, 2005, pp. 402–411.

[36] E. F. Barbosa, J. C. Maldonado, and A. M. R. Vincenzi, "Toward the determination of sufficient mutant operators for C," *Softw. Test., Verif. Reliab.*, vol. 11, no. 2, pp. 113–136, 2001.

[37] A. S. Namin, J. H. Andrews, and D. J. Murdoch, "Sufficient mutation operators for measuring test effectiveness," in *ICSE*. ACM, 2008, pp. 351–360.

[38] W. Visser, "What makes killing a mutant hard," in *ASE*. ACM, 2016, pp. 39–44.

[39] X. Yao, M. Harman, and Y. Jia, "A study of equivalent and stubborn mutation operators using human analysis of equivalence," in *ICSE*. ACM, 2014, pp. 919–930.

[40] R. A. DeMillo and A. J. Offutt, "Constraint-based automatic test data generation," *IEEE Trans. Software Eng.*, vol. 17, no. 9, pp. 900–910, 1991.

[41] A. J. Offutt, Z. Jin, and J. Pan, "The dynamic domain reduction procedure for test data generation," *Softw., Pract. Exper.*, vol. 29, no. 2, pp. 167–193, 1999.

[42] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *PLDI*. ACM, 2005, pp. 213–223.

[43] L. Bottaci, "A genetic algorithm fitness function for mutation testing," in *SEMINAL*, 2001, pp. 1–5.

[44] K. Ayari, S. Bouktif, and G. Antoniol, "Automatic mutation test input data generation via ant colony," in *GECCO*. ACM, 2007, pp. 1074–1081.

[45] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," *IEEE Trans. Software Eng.*, vol. 38, no. 2, pp. 278–292, 2012.

[46] G. Fraser and A. Arcuri, "Achieving scalable mutation-based generation of whole test suites," *Empirical Software Engineering*, vol. 20, no. 3, pp. 783–812, 2015.

[47] M. Harman, Y. Jia, and W. B. Langdon, "Strong higher order mutation-based test data generation," in *FSE-ESEC*. ACM, 2011, pp. 212–222.

[48] F. Alhwikem, R. F. Paige, L. Rose, and R. Alexander, "A systematic approach for designing mutation operators for MDE languages," in *MoDeVVa*, 2016, pp. 54–59.

[49] P. Gómez-Abajo, E. Guerra, J. de Lara, and M. G. Merayo, "A tool for domain-independent model mutation," *Sci. Comput. Program.*, vol. 163, pp. 85–92, 2018.

[50] S. Sen and B. Baudry, "Mutation-based model synthesis in model driven engineering," in *Mutation Wokshop*, 2006.