# Applying Mutation Testing to ATL Specifications: An Experimental Case Study

Yasser Khan and Jameleddine Hassine
Department of Information and Computer Science
King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia
{yasera, jhassine}@kfupm.edu.sa

***Abstract*—Mutation testing is a well-established fault-based technique for assessing and improving the quality of test suites. In order to support mutation testing for model transformations, we define a set of eleven mutation operators for the Atlas Transformation Language (ATL). The effectiveness of the resulting operators, generated automatically using our prototype tool *MuATL*, is evaluated using a case study of an ATL program that refactors a given UML use case model. Our analysis shows that the proposed operators can successfully detect inadequacies in a given test suite.**

***Keywords-Model transformation*; *Model Driven Engineering*; *mutation testing*; *mutation operators*; *Atlas Transformation Language*;**

## I. INTRODUCTION

Model transformations aim to automatically convert a *source* model to a *target* model based on a set of transformation *rules* [1]. A rule defines how attributes of a source object map to attributes of a target object. The source and target models must each conform to well defined *metamodels*, which specifies the language (syntax and semantics) of the models [2]. Apart from model refinement, model transformation can greatly improve several software development activities; including model refactoring, reverse engineering of models, and applying design patterns [3].

Faults in model transformations may result in defective models, and eventually defective code. Many approaches to test model transformations have been proposed in the literature. Lamari [4] used a functional testing approach based on a data partitioning technique that focuses on the structure of models in order to take into account the structural aspect of models when generating input test models. González and Cabot [5] and McQuillan and Power [6] have proposed white-box test model generation approaches for ATL model transformations. Fleurey et al. [7] investigated the problem of test data generation for model transformations and proposed the use of partition testing to define test criteria to cover the input metamodels. Fiorentini et al. [8] have proposed a uniform framework for treating metamodels, model transformation specifications and the automation of test case generation. Their proposed technique [8] is based on a black-box testing approach of model transformations to validate their adherence to given specifications. A gray-box testing technique has also been used by Bauer and Küster [9] for model transformations. Mottu et al. [10] have introduced the application of mutation testing to model transformations. The authors [10] have identified four semantic classes of faults (navigation, filtering, output model creation, and input model modification) for model transformations and they have defined a set of generic mutation operators to cover these class faults.

The widespread interest in testing model transformation programs provides the major motivation for this research. We, in particular, focus on investigating the applicability of fault-based testing to model transformations. To this end, this paper has the following purposes:

- It extends our previous work [11] on designing mutation operators for the ATL language [12], so that model transformation developers can gain the benefits of mutation testing.
- It evaluates the usefulness and the effectiveness of the proposed operators using a case study of a UML use case refactoring ATL specification.

The remainder of this paper is organized as follows. Our proposed ATL mutation testing approach is presented in Section II. Section III introduces a suite of 11 mutation operators for the ATL transformation language. In Section IV, we apply the defined mutation operators to an ATL program that refactors a given use case model. Finally, conclusions are drawn in Section V.

## II. ATL MUTATION TESTING APPROACH

*Mutation testing* is a well-established fault based testing technique, for assessing and improving the quality of test suites. An ATL *mutation operator* defines how a particular ATL artifact is altered in order to inject a single fault. The resulting ATL program is known as a *mutant*. If a mutant is syntactically incorrect, it is considered as an *invalid* mutant.

An ATL test suite consists of a synthesis of a number test cases consisting of input models and expected models. The original ATL program (i.e., ATL Spec S in Fig. 1) and the generated mutants run on the test cases and the results are compared using an oracle. Defining a test oracle for model transformations is a challenging task [13]. ATL Mutants are generated automatically using our prototype tool *MuATL* (Mutation Toolkit for ATL). *MuATL*, a Microsoft .NET C# based tool, is inspired by MuJava (Mutation System for Java) [14] . The execution of the test suite and the oracle function are performed manually. The automation of such activities is out of the scope of this paper.

 ISBN: 978-1-61208-304-9 

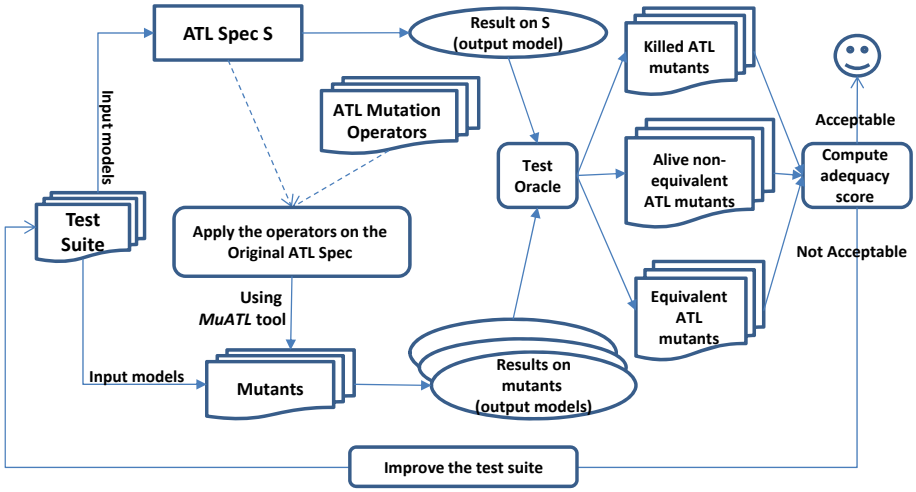



Fig. 1. ATL Mutation Process

A given test case, part of the test suite, is said to *kill* a mutant if the output model produced by the mutant is different from the expected model produced by the original ATL specification. Hence the test case is good enough to detect the change between the original and the mutant ATL program. A test case cannot distinguish between a mutant and the original ATL program if both produce the same output model(s) for the same input model. If a mutant is not killed (called *alive*) by a test suite, this usually means that the test suite is not adequate. However, it may also be that the mutant keeps the program's semantics unchanged-and thus cannot be detected by any test case. Such mutants are called *equivalent* mutants. Equivalent mutants detection is, in general, one of biggest obstacles for practical usage of mutation testing [15]. Fig. 1 illustrates our mutation testing process for the ATL language [12].

The effectiveness of a test suite is determined by running it on all mutants and computing its mutation *adequacy score*, that is the ratio of killed mutants to total number of non-equivalent mutants.

$$AdequacyScore = \frac{M_k}{M_t - M_e} \quad (1)$$

where $M_k$ is the number of killed ATL mutants, $M_t$ is the total number of generated ATL mutants, and $M_e$ is the number of ATL equivalent mutants. If the score is not acceptable, the test suite should be improved by adding additional test cases and/or modifying the existing ones.

## III. ATL Mutation Operators

In this section, we briefly present the eleven proposed ATL mutation operators.

### A. Matched to Lazy (M2L)

The M2L operator converts a matched rule to a lazy rule (which is an imperative rule). The consequence of applying the M2L operator is that a mutant rule will never be executed, since lazy rules must be explicitly invoked; thus, resulting in loss of information. If an input model contains at least one object on which the mutant rule is applicable, the corresponding M2L mutant will be killed. Otherwise, the mutant rule will not be exercised by the test case; therefore, resulting in an alive M2L mutant. An example of a mutation performed by applying the M2L operator is shown in Fig. 2(a). The M2L operator prepends the rule *AtoB* by the lazy modifier in the mutant rule *AtoB'*.

### B. Lazy to Matched (L2M)

The L2M operator does the opposite of the M2L operator; it converts a lazy rule into a matched rule. Matched rules cannot be explicitly invoked; therefore, a runtime failure will occur when a L2M mutant rule is called. However, a L2M mutation cannot be detected if the mutant rule is not invoked during the execution. An example of a mutation performed by applying the L2M operator is shown in Fig. 2(b). The L2M operator deletes the *lazy* modifier of rule *AtoB* in the mutant rule *AtoB'*.

### C. Delete Attribute Mapping (DAM)

The DAM operator deletes an attribute mapping from the definition of a particular rule. It is based on the CACD operator in [10]. The consequence of applying the DAM operator on a rule is that the attribute, whose mapping is deleted, will not participate in the transformation process, resulting in a loss of information. However, a DAM mutation will not be detected when the source attribute does not have a specified value. The DAM operator can be applied on matched, lazy and mapping called rules. An example of a mutation performed by applying the DAM operator is shown in Fig. 2(c). The DAM operator deletes the mapping of attribute *b2* in the mutant rule *AtoB'*.

### D. Add Attribute Mapping (AAM)

The AAM operator adds a useless attribute mapping from a source object to a target object in a given rule. It is based on the CACA operator in [10]. The consequence of applying the AAM operator on a rule is that unnecessary complexity is

 ISBN: 978-1-61208-304-9 

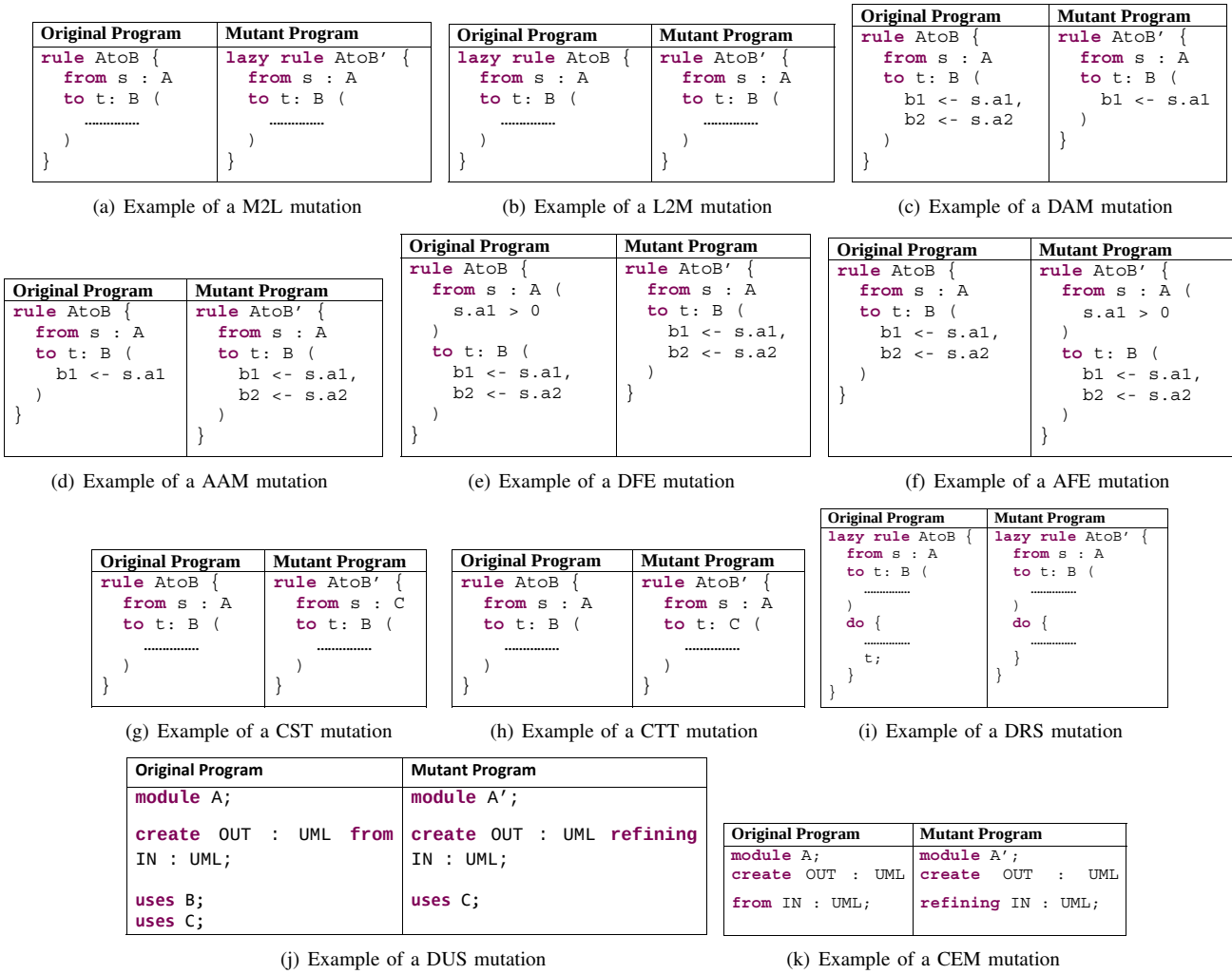



Fig. 2. Code examples of the proposed mutation operators

added to the output model. AAM mutants may also cause a runtime failure if the source and target attributes types are incompatible. An example of a mutation performed by applying the AAM operator is shown in Fig. 2(d). The AAM operator adds the useless mapping "*b2 <– s.a2*" in the mutant rule *AtoB'*.

### *E. Delete Filtering Expression (DFE)*

Filtering expressions constrain the input objects on which a particular rule can be applied. If a filtering statement evaluates to true for a given input object, its corresponding rule will be executed. This can only be applied on matched rules, as they allow filtering of input objects. The DFE operator deletes the filtering statement specified in the definition of a rule. It is based on the CFCD operator in [10]. The consequence of applying the DFE operator is that the mutant rule will be executed for incorrect objects of its source type. DFE operator may cause filtering expressions of multiple rules to evaluate to true for one source instance. In this case, a runtime failure will occur. An example of a mutation performed by applying the DFE operator is shown in Fig. 2(e). The DFE operator removes the filtering expression $s.a1 > 0$ in mutant rule *AtoB'*.

### *F. Add Filtering Expression (AFE)*

Based on the CFD operator in [10], we define the AFE operator which performs the opposite of the DFE operator. It adds an unnecessary filtering expression to a matched rule. The consequence of applying the AFE operator is that some objects of the input model will not participate in the transformation process, thus resulting in a loss of information. In order to apply the AFE operator on a rule, the source object must have at least one attribute. If this condition is satisfied, a numerous AFE mutants can be created for a given matched rule. Input Space Partitioning [16] can be applied on each source attribute to produce a set of mutant filtering expressions.

An example of a mutation performed by applying the AFE operator is shown in Fig. 2(f). The AFE operator adds the filtering expression $s.a1 > 0$ in mutant rule *AtoB'*.

### G. Change Source Type (CST)

The CST operator changes the source type of a given rule. It can be applied on matched and lazy rules. The consequence of applying the CST operator is that incorrect transformations may be performed. Indeed, the application of the CST operator on a rule may cause a runtime failure if the new source type does not contain the attributes which are specified to be mapped, or if multiple rules are associated with the new source type. An example of a mutation performed by applying the CST operator is shown in Fig. 2(g). The source type of rule *AtoB* is changed from *A* to *C* in the mutant rule *AtoB'*.

### H. Change Target Type (CTT)

The CTT operator changes the target type of a given rule. It can be applied on matched, lazy and mapping called rules. The consequence of applying the CTT operator is that the objects in the input model will be transformed into objects of incorrect type in the output model. An example of a mutation performed by applying the CTT operator is shown in Fig. 2(h). The target type of rule *AtoB* is changed to *C* in the mutant rule *AtoB'*.

It should be noted that CST and CTT do not produce syntacticly incorrect mutants.

### I. Delete Return Statement (DRS)

The last statement of a *do* block in a mapping called rule must return the target object. It is optional to specify a return statement in the *do* block of matched and lazy rules. The DRS mutation operator deletes the return statement of a *do* block. An example of a mutation performed by applying the DRS operator is shown in Fig. 2(i). The DRS operator deletes the return statement "*t;*" of the *do* block of rule *AtoB* in mutant rule *AtoB'*.

### J. Delete Use Statement (DUS)

An ATL module can import functions from a reusable library via the *uses* keyword. We define, the DUS operator which deletes an import statement from a given module. Since the ATL compiler does not check whether external functions are imported or not, the DUS operator does not produce an invalid mutant. If no external function is invoked by a test case, a DUS mutant will remain alive. An example of a mutation performed by applying the DUS operator is shown in Fig. 2(j). The DUS operator deletes the import statement of library B in mutant module A.

### K. Change Execution Mode (CEM)

ATL modules can execute in two modes, *default* and *refining*. Default mode is the default execution mode of ATL transformations and it is specified by the *from* keyword. The refining mode allows developer to specify rules only for those objects that need to be transformed; remaining objects will be implicitly copied into the output model. It should be added that refining mode applies only when the source and target models conform to the same metamodel. We define the CEM operator which switches the execution mode of an ATL module from *default* to *refining* mode, or vice versa. In default mode, a CEM mutation may cause useless objects to be copied into the output model; whereas, in refining mode, it will cause loss of information. If a module contains imperative code, which is not allowed in refining mode, application of the CEM operator will result in an invalid (i.e., syntactically incorrect) mutant. An example of a mutation performed by applying the CEM operator is shown in Fig. 2(k). The CEM operator changes the execution mode of module *A* to refining mode in the mutant module *A'*.

## IV. Case Study: UML Use Case Model Refactoring

The case study pertains to an ATL module, which implements a use case model refactoring. This refactoring is based on use case antipattern *a1*, which is introduced in [17]. Antipattern *a1* occurs when an actor is associated with a generalized use case in order to enable indirect access to a framework of services, which are implemented by specialized use cases. A generalized use case is often incomplete because it contains parts of common behavior required by the specialized use cases. Therefore, initiation of such a generalized use case will result in incomplete meaningless behavior. A given use case is involved in this antipattern if it:

- is a *concrete generalized* use case
- neither *includes* nor *extends* any use case
- not *extended* by any other use case
- is directly or indirectly associated with an actor

For a given input use case model, the transformation detects the model elements involved in *a1*, and performs the *ConcreteToAbstract* refactoring, which converts the *generalized* use case to an *abstract* use case. The semantics of *abstract* use cases are similar to the semantics of an abstract entity in the OO paradigm. Setting a use case as *abstract* indicates that it cannot be solely performed. Therefore, one of the *specialized* use cases will be performed. This guarantees that a complete and meaningful service will be delivered to the actor. If *a1* is not detected, the refactoring is not performed. Fig. 3 shows the subject ATL module, which is implemented in refining mode. It references three reusable libraries: *UseCase*, *Association*, and *Actor*. The filtering expression specified in the *from* clause of matched rule *AbstractGeneralizedUC* implements the detection conditions for *a1*. If a use case satisfies all of these detection conditions, its *isAbstract* property is set.

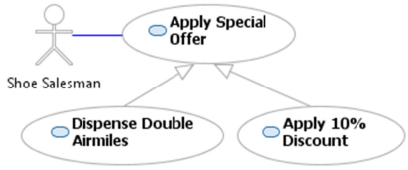
The case study contains 9 test cases which satisfy the Correlated Active Class Coverage (CACC) criteria [18], a logic coverage testing criteria that tests individual clauses in a logical expression. Each test case includes the input model and the expected output model. For instance, Fig. 4 and Fig. 5 illustrate the input model and the expected output model relative to test cases TC1 and TC2, respectively. In the input model of TC1, use case *Apply Special Offer* is involved in antipattern *a1*; therefore, it is set abstract in the output model. It should be noted that the antipattern *a1* is not detected in TC2; hence, no refactoring is performed.

The proposed mutation operators are automatically applied on the subject module using our prototype tool MuATL, and

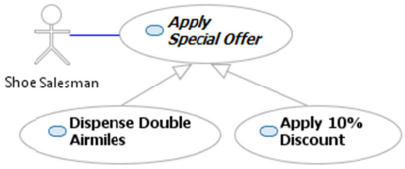
 ISBN: 978-1-61208-304-9 

```
module ConcreteToAbstract;
create OUT : UML refining IN : UML;

uses UseCase;
uses Association;
uses Actor;

rule AbstractGeneralizedUC {
  from s: UML!UseCase (
    s.isGeneralization() and
        s.isConcrete() and
        not (
      s.isIncluder() or
      s.isExtension() or
      s.isExtended()
        )
        and (
          (s.isAssociatedWithActor() and
           not s.isIncluded()) or
      s.isIndirectlyAssociatedWithActor()
        )
  )
  to t: UML!UseCase (
    isAbstract<-true
  )
}
```

Fig. 3. Excerpt of the Use Case refactoring model transformation




(a) Input Use Case Model




(b) Expected Output Use Case Model

Fig. 4. Input and expected output models of TC1

result in 47 mutant modules. In addition to the proposed operators, the Conditional Operator Replacement (COR) [16], Unary Operator Deletion (UOD) [16], and the Non-Void Method Call (NVMC) [19] operators are also applied. These additional operators are used because they will target the filtering expression of rule *AbstractGeneralizedUC*.

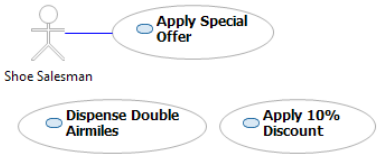
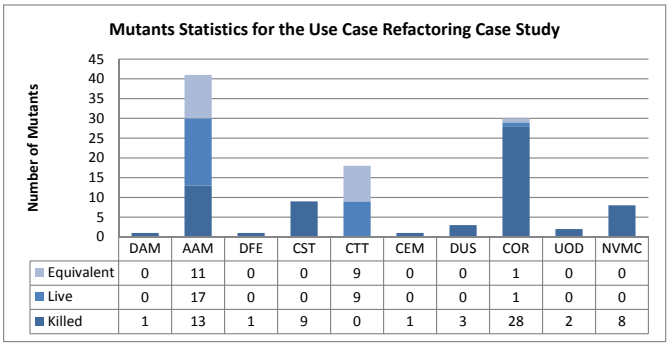


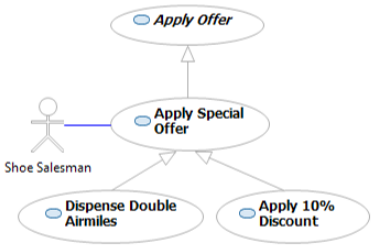

Fig. 5. Input and expected output models of TC2 (they are the same)

The rule *AbstractGeneralizedUC* contains 6 unmapped source attributes (*name*, *isAbstract*, *include*, *extend*, *generalization*, *subject*) and 5 unmapped target attributes (*name*,

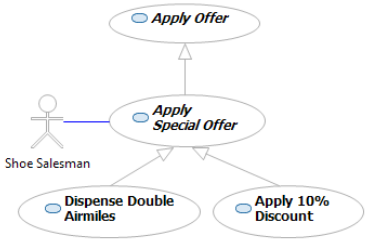



| | DAM | AAM | DFE | CST | CTT | CEM | DUS | COR | UOD | NVMC |
|---|---|---|---|---|---|---|---|---|---|---|
| Equivalent | 0 | 11 | 0 | 0 | 9 | 0 | 0 | 1 | 0 | 0 |
| Live | 0 | 17 | 0 | 0 | 9 | 0 | 0 | 1 | 0 | 0 |
| Killed | 1 | 13 | 1 | 9 | 0 | 1 | 3 | 28 | 2 | 8 |

Fig. 6. Live, killed, and equivalent mutants for the *ConcreteToAbstract* model transformation program

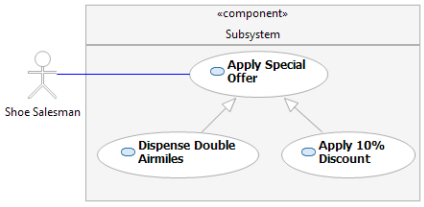



(a) Input Model for TC10

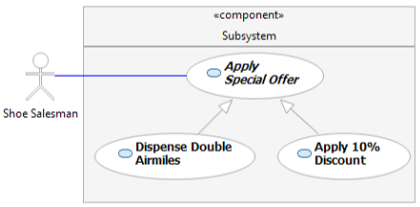



(b) Expected output Model for TC10

Fig. 7. Input and expected output models for TC10




(a) Input model for TC11




(b) Expected output model for TC11

Fig. 8. Input and expected output models for TC11

 ISBN: 978-1-61208-304-9

*include*, *extend*, *generalization*, *subject*); therefore, the application of the AAM operator resulted in 30 mutants. One DAM mutant was created for the mapping statement "*isAbstract <- true*". 10 source classes, and 10 target classes participate in the model transformation; therefore, 9 CST and 9 CTT mutants are created. It should be noted that these sets of source and target classes are the same. One DFE mutant corresponds to the filtering expression of *AbstractGeneralizedUC*. The AFE operator could not be applied on *AbstractGeneralizedUC* because it already contained a filtering expression. The M2L and L2M operators are also not applicable because the subject module is specified in refining mode. The module imports 3 reusable libraries; therefore, a DUS mutant is created for each import statement.

The results of the mutation analysis, presented in Fig. 6, reveal that 66 mutants are killed by the 9 test cases, and 27 mutants are kept alive. 1 DAM, 13 AAM, 5 CST, and 3 DUS mutants are killed as a result of runtime failures. 1 DFE, 4 CST, 1 CEM, 28 COR, 2 UOD, and 8 NVMC are killed because they produce incorrect output models. The 9 live CTT mutants are equivalent mutants; they cannot be killed by any test case. The single live COR mutant resulted in errors states for several test cases; however, these error states did not propagate into a failure. Moreover, for this mutant, no test case can be designed which will result in a failure; therefore, it was concluded as equivalent.

The nine test cases give an adequacy score of 91.67%. The obtained results show that the AAM operator determined inadequacies in the subject test suite. The 6 live non-equivalent AAM mutants (i.e., 17-11 = 6) can be killed by adding new test cases. We add TC10 and TC11, each of which kills 3 live AAM mutants, to the subject test suite. This enhanced test suite gives a 100% adequacy score. The input models of TC10 and TC11 are shown in Fig. 7(a) and Fig. 8(a), respectively.

## V. Conclusions

In order to support mutation testing for ATL language, we have defined a set of eleven mutation operators. Our approach has been validated using a use case model refactoring program. The results have shown that the operators successfully detected inadequacies in the subject test suite.

As a future work, we are planning to further enhance our prototype tool *MuATL* to include a test case execution engine and a test oracle. In addition, we aim at conducting an empirical study to better assess the usefulness and the effectiveness of the proposed ATL operators.

Furthermore, we will investigate the addition of mutation operators of traditional programming languages that are relevant to ATL. The idea of mutation testing will also be explored for other model transformation languages, such as QVT, Tefkat, and Epsilon.

## Acknowledgment

The authors would like to acknowledge the support provided by the Deanship of Scientific Research at King Fahd University of Petroleum & Minerals (KFUPM) for funding this work through project No. IN121009.

## References

[1] A. G. Kleppe, J. Warmer, and W. Bast, *MDA Explained: The Model Driven Architecture: Practice and Promise*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[2] J. M. Favre, "Towards a basic theory to model model driven engineering," in *In Workshop on Software Model Engineering, WISME 2004, joint event with UML2004*, 2004.

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

[4] M. Lamari, "Towards an automated test generation for the verification of model transformations," in *Proceedings of the 2007 ACM symposium on Applied computing*, ser. SAC '07. New York, NY, USA: ACM, 2007, pp. 998–1005.

[5] C. A. González and J. Cabot, "Atltest: a white-box test generation approach for ATL transformations," in *Proceedings of the 15th international conference on Model Driven Engineering Languages and Systems*, ser. MODELS'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 449–464.

[6] J. A. Mc Quillan and J. F. Power, "White-Box Coverage Criteria for Model Transformations." in *1st International Workshop on Model Transformation with ATL*, Jul 2009, p. 63.

[7] F. Fleurey, J. Steel, and B. Baudry, "Validation in model-driven engineering: testing model transformations," in *Model, Design and Validation (MoDeVa 2004), Rennes, France*, nov. 2004, pp. 29 – 40.

[8] C. Fiorentini, A. Momigliano, M. Ornaghi, and I. Poernomo, "A constructive approach to testing model transformations," in *Proceedings of the Third international conference on Theory and practice of model transformations*, ser. ICMT'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 77–92.

[9] E. Bauer and J. M. Küster, "Combining specification-based and code-based coverage for model transformation chains," in *Proceedings of the 4th international conference on Theory and practice of model transformations*, ser. ICMT'11. Springer-Verlag, 2011, pp. 78–92.

[10] J.-M. Mottu, B. Baudry, and Y. Le Traon, "Mutation analysis testing for model transformations," in *Proceedings of the Second European conference on Model Driven Architecture: foundations and Applications*, ser. ECMDA-FA'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 376–390.

[11] Y. Khan and J. Hassine, "Mutation operators for the atlas transformation language," in *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, ser. ICSTW '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 43–52. [Online]. Available: http://dx.doi.org/10.1109/ICSTW.2013.13

[12] 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, Jun. 2008.

[13] J.-M. Mottu, B. Baudry, and Y. Le Traon, "Model transformation testing: oracle issue," in *IEEE International Conference on Software Testing Verification and Validation Workshop (ICSTW)*, april 2008, pp. 105 – 112.

[14] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "Mujava: an automated class mutation system: Research articles," *Softw. Test. Verif. Reliab.*, vol. 15, no. 2, pp. 97–133, Jun. 2005.

[15] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Trans. Software Eng.*, vol. 37, no. 5, pp. 649–678, 2011.

[16] P. Ammann and J. Offutt, *Introduction to Software Testing*, 1st ed. New York, NY, USA: Cambridge University Press, 2008.

[17] M. El-Attar and J. Miller, "Improving the quality of use case models using antipatterns," *Software & Systems Modeling*, vol. 9, no. 2, pp. 141–160, 2010.

[18] P. Ammann, J. Offutt, and H. Huang, "Coverage criteria for logical expressions," in *14th International Symposium on Software Reliability Engineering), 17-20 November 2003, Denver, CO, USA*, ser. ISSRE '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 99–107.

[19] PIT, "PIT mutation testing," http://pitest.org/quickstart/mutators/, last accessed, August 2013.

 ISBN: 978-1-61208-304-9 