
Conference or Workshop Item:
Alhwikem, Faisal Haji M, Paige, Richard Freeman, Rose, Louis Matthew et al. (2016) A Systematic Approach for Designing Mutation Operators for MDE languages. In: UNSPECIFIED.

A Systematic Approach for Designing Mutation Operators for MDE languages

Faisal Alhwikem *, Richard Paige †, Louis Rose ‡, Rob Alexander §
Department of Computer Science, University of York, United Kingdom
Email: {*fhma500, †richard.paige, ‡louis.rose, §rob.alexander}@york.ac.uk

***Abstract*—Testing is an essential activity in software development, used to increase confidence in the quality of software. One testing approach that is used to evaluate the quality of testing inputs for a particular program is *mutation analysis*. The most important step in mutation analysis is the process of defining mutation operators that mimic typical errors of the users of a language. There is a wide variety of mutation operators that have been designed for a number of languages including C, Java, and SQL. However, the design of mutation operators is rarely systematic, which may result in passing over crucial operators for specific features of languages.**

This paper describes a way to apply mutation analysis in the context of Model Driven Engineering (MDE). In particular, the paper proposes a systematic approach for designing mutation operators for MDE languages. The systematic approach is demonstrated for the Atlas Transformation Language (ATL) and the result is a list of mutation operators that includes previously designed ones for ATL from the literature.

I. Introduction

Testing is an essential practice that can be used to assess the quality of software. It can involve running a program against a set of test cases in order to reveal defects. Hence, the quality of test cases is crucial in raising the confidence of the quality of a program. One evaluation approach that is used to measure the quality of the test cases is *mutation analysis*, where defects are seeded deliberately into a program using mutation operators to generate faulty versions known as mutants [1]. Mutants are then executed against a set of test cases in order to study their ability of detecting those introduced defects and to compute a *mutation score* (the number of detected mutants over the number of total mutants).

There are a number of mutation operators that have been defined for a number of languages such as Java, C#, and SQL. Operators are usually designed by examining elements of the languages, or motivated from other mutation operators designed for similar languages. Although there are systematic approaches for deriving mutation operators for Java in [2] and the Goal Agent Language in [3], the design of mutation operators is rarely systematic, and there are few principles and practices to explain how to generate operators from language definitions or previously harvested operators. As such, the design process for mutation operators may overlook crucial operators or may generate poor designs, which can result in ineffective tests. In particular, mutation operators are meant to be carefully designed such that they give realistic errors and to force the test developer to deal with such errors either by adding or enhancing existing testing inputs. Hence, a systematic approach for designing mutation operators can ensure the coverage of a language's features as well as designing operators that more likely to reflect realistic errors.

Model Driven Engineering can contribute to this challenge because languages in MDE can be models and structured according to some modelling concepts and therefore they are amenable to systematic and automated analysis. Furthermore, those modelling concepts are usually specified using a common metamodelling language (e.g. Ecore) and hence, we have the opportunity to define a set of mutation operators systematically that is applicable to a set of MDE languages that are derived from that common language.

This paper is structured as follows. Section II gives an overview of related work and section III presents our systematic approach for designing mutation operators. In section IV, we illustrate the application of the systematic approach to Ecore metamodelling language and discuss some findings in section V. Section VI presents our application of the systematic approach on ATL (Atlas Transformation Language) and compares our list of mutation operators with already defined ones found in the related literature. Section VII gives our conclusion and future work.

II. Related Work

There have been a number of attempts to define mutation operators to MDE languages. Mottu et al. [4] have proposed a set of generic mutation operators that can be applied to model transformations. They argue that their operators are based on the core activities of model transformation: 1) the navigation of models via relations between classes defined in input and output metamodels, 2) the filtering of a collection of objects, 3) the creation and modification of output models. Based on these generic mutation operators, they have proposed a set of 10 mutation operators that are defined specifically for model transformation. Although no systematic approach has been used for generating those mutation operators, they were widely used as a set of formalised mutation operators for much work as in [4], [5], [6], [7], [8], and [9].

A close work to ours is presented by Troya et al. [10]. They introduce a systematic approach for generating mutation

operators for ATL by examining its metamodel and manually applying three mutation operators; namely addition, deletion and modification to some concepts of the language. As a result of their application, they have constructed a set of 19 mutation operators. However, they only considered 5 modelling concepts of ATL, while one of our goals is to cover the entire language. Furthermore, their approach is only dedicated to ATL, whilst we are interested in a an approach that derives a set of mutation operators that can be used across different MDE languages.

III. Systematic Approach For Designing Mutation Operators

Our systematic approach of generating mutation operators in the context of MDE relies on examining a common meta-metamodel of a number of metamodels. In particular, we have chosen Ecore as a meta-metamodel for this, because it is used to express models of a number of popular metamodels such as ATL, Epsilon Object Language (EOL) [1], and Epsilon Transformation Language (ETL) [2] metamodels.

Ecore consists of a number of concepts that can be used to define a metamodel. A metamodel consists of a number of concepts that can be used to represent a particular model. Usually, models can be mutated by using the concepts of its metamodel. Hence, metamodels can be mutated by using the concepts of their meta-metamodel (i.e. Ecore). Thus, we believe that it is possible to generate a number of mutation operators that can be used to mutate arbitrary models by examining their metamodel, using the concepts provided by its meta-metamodel.

In this paper, we introduce the term *mutation actions* to refer to a set of common mutation actions derived from examining a list of mutation operators of a number of languages discussed thoroughly in the literature. These mutation actions are addition, deletion, and replacement. The addition action is used to introduce additional data to a particular feature of an object, while the deletion action is used to do the opposite. The replacement action is used to replace existing values.

The process of this approach is to examine each modelling concept of a metamodel, apply the *mutation actions* to it, and try to generate all applicable mutation operators for that particular concept. By applicable mutation operator, we mean an operator that when applied, does not generate a model that does not conform to its metamodel. For instance, the modelling concept *sections* in listing 1 is to represent *at least* one instance of a type *Section* for a specific *Chapter* as imposed by the multiplicity specified in line 7 (i.e. [1..*]). When the delete mutation action is applied, it involves deleting one instance of *Section* from a list of instances represented by it. In the case that there is only one instance of *Section* represented, it is possible when applying the action, an invalid mutant might be generated. Hence, defining mutation operator for a modelling concept should be governed by defining preconditions to each concept to minimise the generation of invalid mutants.

Listing 1. A metamodel of a *book* using Emfatic

```
1
class Book {                                    2
  attr String author;                           3
  val Chapter[1..*] chapters;                   4
}                                               5
class Chapter {                                 6
  attr String title;                            7
  val Section[1..*] sections;                   8
}                                               9
class Section {                                 10
  attr String content;                          11
  ref Section[0..*] refersTo;                   12
}                                               13
```

We have applied the systematic approach to Ecore generating a set of mutation operators that are abstract and thus applicable to various Ecore-based models (i.e. metamodels). This is describe in the following section.

IV. Applying the Systematic Approach to Ecore

A. Ecore Overview

Eclipse Modelling Framework (EMF) [3] is a framework created by the Eclipse Foundation [4] that brings support for MDE. It provides a number of facilities for modelling and models interchange [11]. One of these facilities is Ecore, a meta-metamodel language that provides modelling concepts that can be used to describe metamodels. Figure 1 shows the core modelling concepts of Ecore, as follows:

- *EClass* is used for modelling classes or types. A class may have a name, a number of structural features, and a number of super-types.
- *EDataType* is used to represent types that are not modelled as classes. For example, it is used to model primitive types such as integer and float or object types such as String and Map defined in Java. A data-type can have a name.
- *EAttribute* is used to model an attribute feature of a class. It is defined by a name and it has a data-type represented by *eAttributeType*
- *EReference* is used to model associations between classes. It is defined by a name and a type that must be an EClass represented by *eReferenceType*

Ecore has a number of core aspects that can be used in modelling. One of these aspect is inheritance, where the features of a parent (super) class can be obtained by its child class. Another aspect is that the structural features (i.e. attributes and references) of a class can be characterised by a number characteristics [12] (c.f. fig 1). For instance, the *lowerBound* and *upperBound* characteristics can be used to specify the multiplicity of a modelling feature. Another characteristic is

[1] http://www.eclipse.org/epsilon/doc/eol/
[2] http://www.eclipse.org/epsilon/doc/etl/
[3] https://eclipse.org/modeling/emf/
[4] http://www.eclipse.org

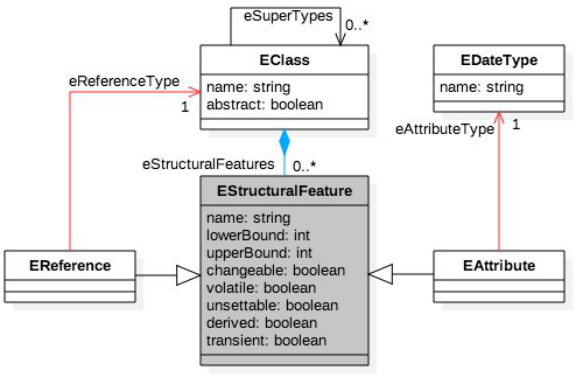

Fig. 1. The core elements of Ecore meta-metamodel adapted from [12]

changeable which indicates whether the value of a modelled feature is editable or read-only. The *derived* characteristic can specify whether the value of a modelled feature is to be computed from other, related data. Finally, the *transient* characteristic is used to specify whether a modelled feature is to be dropped at the serialisation of its containing object (i.e. class). All mentioned characteristics can be important when applying the systematic approach.

The three mutation actions have been applied upon certain concepts of Ecore to generate possible mutation operators for each one. In our application, we have ignored concepts that specified as unchangeable (i.e. read-only), derived, or transient. For instance, we ignored unchangeable concepts because their values are not modifiable and thus there is no benefits in defining mutation operators for such concepts. Derived modelling concepts as mentioned before are computed from other values. Hence, it makes sense to mutate only those values which they derived from and avoid defining redundant mutation operators. Finally, transient modelling concepts are eliminated during the serialisation of their contained model and hence any modification to those special concepts' values are trivial and serve no purpose as persisting of models (i.e. mutants) along with its modified data is essential task in mutation analysis. Specifically, in mutation analysis, mutants are usually saved for further analysis; for example run them against test cases, compare them with original models, and/or use them to evaluate the mutation operators that generated them.

For each considered modelling concept in Ecore, we define possible information based on the type of the modelling concept. For an EClass (c.f. fig 1), we define 1) name of the instance, 2) its super-types, and 3) list of its features. For an EDatatype instance, we define manually a set of mutation operators that are applicable to it when applying the mutation actions. For example, applying the mutation actions to Ecore Integer data-type would generate three mutation operators: adding an integer value to previously existing value, subtracting an integer value from an existing value, and replacing an existing integer value with a new one. For each EStructuralFeature of an EClass, we define 1) name of the feature, 2) its type (whether its a class or a data-type), and 3) its multiplicity (i.e. the lower and upper bounds constraints of the feature).

Furthermore, each considered modelling concept can inherit a set of mutation operators designed for its super-types. For instance, the *EClassifier* (see Ecore meta-class diagram figure at [5]) can acquire all sets of mutation operators defined for its super-type *ENamedElement*. The latter also inherits mutation operators defined for its super-type *EModelElement*. The benefit of applying this mechanism is to cover all features of a particular object including the inherited ones.

The downside of this process is the generation of a large set of mutation operators that may produce a great number of mutants; an issue that may lead to an expensive mutation analysis. However, the approach that we are proposing can produce a complete set of operators; including crucial operators that might have been overlooked using unsystematic design of mutation operators. Moreover, there are many cost reduction techniques (e.g. mutants sampling and mutation selection) that have been widely used in the context of programming languages to tackle the issue of expensive mutation analysis; which can be adapted and implemented in the context of MDE.

B. Example of Applying the Systematic Approach

In this example, we illustrate the application of the systematic approach to one of Ecore's modelling concept - *EPackage*. Since this particular concept is an EClass, we define the following:

1) Name: EPackage
2) Super-type: ENamedElement
3) EStructuralFeatures: EString nsURI, EString nsPrefix, EClassifier eClassifiers, EPackage eSubpackages, EString name, EAnnotation eAnnotations

In this section, we will only show the mutation operators for *nsURI* and *eClassifiers* features. The former feature is used to hold the namespace URI (Uniform Resource Identifiers) of a specific package while the latter feature represents the package classifiers (i.e. classes and data-types). The mutation operators for other features are quite similar. For example, the features *nsPrefix* and *name* have similar data-type and multiplicity of the feature *nsURI*. Likewise, the feature *eSubpackages* and *eAnnotations* have identical lower and upper bounds of the feature *eClassifiers*. Therefore, we list here the mutation operators that we manually designed for feature $nsURI$ and $eClassifiers$ as an examples of the implementation of the systematic approach.

- **EAttribute[EString nsURI], lowerBound=0, upperBound=1**: based on the multiplicity imposed by the lower and the upper bounds, this feature can hold a single value

[5] goo.gl/KjIAGR

and can take the following mutation operators defined for *EString* data-type.

- *ADD(EString nsURI, EString toAdd)*: Appends to the value of *nsURI* the value specified by *toAdd*.
  Preconditions:
  * $nsURI.isDefined()$ & $toAdd.isDefined()$
    Check that both $nsURI$ and $toAdd$ are valid and defined.
  * $toAdd.length \geq 1$
    For the changes to take place (or mutation), it is essential to ensure that this operator modifies existing value (i.e. $nsURI$) with at least one character.
- *DEL(EString nsURI, Integer toRemove)*: Removes randomly a number of *toRemove* characters from *nsURI*.
  Preconditions:
  * $nsURI.isDefined()$ & $toRemove.isDefined()$
    Verify that both values represented by $nsURI$ and $toRemove$ are valid and defined.
  * $nsURI.length \geq toRemove \geq 1$
    Ensure that the value represented by $toRemove$, which is the number of characters to be removed from $nsURI$, is less than or equal to the entire string size of $nsURI$ and greater than 0.
- *REP(EString nsURI, EString newValue)*: Replaces the value of *nsURI* with the value of *newValue*.
  Preconditions:
  * $newValue.isDefined()$ & $nsURI \neq newValue$ Check that $newValue$ is defined (i.e. not $null$) and its value is equal to *nsURI* in order to generate a valid mutation.

- **EReference[EClassifier eClassifiers], lowerBound=0, upperBound=***: based on the multiplicity introduced by the lower and the upper bounds, this feature can hold multiple values and can take the following mutation operators.
  - *ADD(EClassifier eClassifiers, Integer index, EClassifier extra)*: Inserts *extra* at the specific position in the list *eClassifiers*.
    Preconditions:
    * $eClassifiers.isDefined()$ & $extra.isDefined()$
      Ensure that both objects are valid (i.e. not $null$).
    * $NOTeClassifiers.include(extra)$
      Check whether the value represented by $extra$ is not already exist in the list of $eClassifiers$ because the addition is meant to add only a new instance to the list.
    * $extra.isKindOf(eClassifiers.getType())$
      Verify that $extra$ is a valid instance of the same type of $eClassifiers.getType()$.
    * $lowerBound \leq eClassifiers.size() + 1 \leq upperBound$
      Check the size of $eClassifiers$ after addition and does not violate the lower and upper bounds constraint.
    * $lowerBound \leq index < eClassifiers.size()$
      Make sure that $index$ is within the range of indices.
  - *DEL(EClassifier eClassifiers, Integer index)*: Deletes the element at the specific position in the list *eClassifiers*
    Preconditions:
    * $eClassifiers.isDefined()$
      Check whether $eClassifiers$ is valid.
    * $lowerBound \leq index < eClassifiers.size()$
      Ensure that $index$ is within the list range of indices.
    * $lowerBound \leq eClassifiers.size() - 1 \leq upperBound$
      Verify the size of $eClassifiers$ and check it does not violate the lower and upper bounds constraint.
  - *REP(EClassifier eClassifiers, Integer index, EClassifier newEClassifier)*: Replaces the value at the specific position in *eClassifiers* with *newEClassifier*.
    Preconditions:
    * $eClassifiers.isDefined()$ & $newEClassifier.isDefined()$
      Ensure that both $eClassifiers$ and $newEClassifier$ are valid and defined.
    * $newEClassifier.isKindOf(eClassifiers.getType())$
      Check whether $newEClassifier$ is of the type or one of the subtypes of $eClassifiers.getType()$
    * $lowerBound \leq index < eClassifiers.size()$
      Verify that the value given by $index$ is within the range of indices.
    * $eClassifiers(index) \neq newEClassifier$
      Verify that $newEClassifier$ does not equal to $eClassifiers(index)$. This would prevent the generation of equivalent mutation.

Two features have been discarded from the list of features of *EPackage*. Those are *EReference[EFactory eFactoryInstance]* and *EReference[EPackage eSuperPackage]*. The former is a transient feature and the latter is a transient and an unchangeable feature.

V. Abstract Mutation Operators

The application of the systematic approach to Ecore has led to an important conclusion: the similarities between Ecore modelling concepts that lead to the definition of nearly identical mutation operators. In the previous example, there were two identical mutation operators for different modelling concept: *EReference[EClassifier eClassifiers]* and *EReference[EPackage eSubpackages]*. This is because the mentioned features have similar multiplicity values. Furthermore, the modelling concepts of the same data-type can use similar mutation operators designed for that common data-type (as the case of *nsURI*, *nsPrefix*, and *name* from the example above). To overcome this redundancy of mutation operators, we construct a list of *Abstract Mutation Operators*, where mutation operators have been abstracted to generalise their implementation over similar modelling concepts. Those abstract operators can be then used to mutate Ecore-based models.

The abstraction of the mutation operators are based on two specific characteristics: multiplicity and type of the feature. The multiplicity characteristic can be used to determine whether a particular feature is single-valued or multi-valued. For single-valued features, the type of the feature matters to distinguish between attributes and references. If the type is of a particular data-type, then a set of mutation operators designed for that data-type is applied to this feature. However, if the feature is a reference feature, then there is another set of mutation operators designed for the single-valued reference features. Hence, we have defined three general *Abstract Mutation Operators (AMO)*: one for single-valued attributes,

and another one is for single-valued references. The last set of abstract mutation operators are for multi-valued features (whether attributes or references). These abstract mutation operators are given below.

A. EAttribute - Single-valued (AMO:single-attr)

The abstract mutation operators for attributes that are single-valued can take certain mutation operators based on the data-type of feature (e.g. *EString* and *EInt*). Because of the limit restriction of the paper, the list of mutation operators for Ecore data-types were not listed here; they can be found at [6].

B. EReference - Single-valued (AMO:single-ref)

The abstract mutation operators for this type of features can be:

- **ADD(Type subject, Type extra):** Assigns the value of *extra* to *subject*.
  Preconditions
  - $subject.isUndefined()$ & $extra.isDefined()$
    Ensure that $subject$ is not defined and only allowing new assignment. However, $extra$ must be valid.
  - $extra.isKindOf(subject.getType())$
    Check whether $extra$ is instance of $subject.getType()$ for valid assignment.
- **DEL(Type subject):** Deletes the value of *subject* (i.e. disjoint this feature from associated value).
  Preconditions
  - $subject.isDefined()$ In order to disjoint this feature from associated value, $subject$ is checked to to be valid.
- **:REP(Type subject, Type newValue):** Replaces the value of *subject* with the value of *newValue*.
  Preconditions
  - $subject.isDefined()$ & $newValue.isDefined()$
    Ensure that both $subject$ and $newValue$ are valid instances.
  - $newValue.isKindOf(subject.getType())$
    Verify that $newValue$ is of the type or one of the subtypes of $subject.getType()$

C. EFeature - Multi-valued (AMO:multi-feat)

The abstract mutation operators for features that are multi-valued (whether attributes or references) can be:

- **ADD(Type subjects, Integer index, Type extra):** Inserts *extra* at the specific position in *subjects*.
  Preconditions
  - $subjects.isDefined()$ & $extra.isDefined()$
    Ensure that both $subjects$ and $extra$ are both valid.
  - $NOTsubjects.include(extra)$
    Check whether $extra$ is already exist in the list of $subjects$ because the addition operator is meant to add only a new instance to the list.
  - $extra.isKindOf(subjects.getType())$
    Verify that $extra$ is of the type or one of the subtypes of $subjects.getType()$
  - $lowerBound \leq subjects.size() + 1 \leq upperBound$
    Check whether the size of $subjects$ after addition does not violate lower and upper bounds.
  - $lowerBound \leq index < subjects.size()$
    Check that $index$ is with range of indices.
- **DEL(Type subjects, Integer index):** Deletes the element at the specific position in *subjects*.
  Preconditions
  - $subjects.isDefined()$ Check that $subjects$ is valid and not $null$
  - $lowerBound \leq index < subjects.size()$
    Ensure that $index$ is within the list range of indices.
  - $lowerBound \leq subjects.size - 1 \leq upperBound$
    Verify that the size of $subjects$ after deletion does not violate lower and upper bounds.
- **REP(Type subjects, Integer index, Type newValue):** Replaces the value at the specific position in *subjects* with *newValue*.
  Preconditions
  - $subjects.isDefined()$ & $newValue.isDefined()$
    Ensure that both $subjects$ and $newValue$ are defined.
  - $newValue.isKindOf(subjects.getType())$
    Check whether $newValue$ is of the type or one of the subtypes of $subjects.getType()$
  - $lowerBound \leq index < subjects.size()$
    Check that $index$ is within the range of indices.
  - $subjects(index) \neq newValue$
    Verify that $newValue$ does not equal to the one that replacing with. This would prevent the generation of equivalent mutation.

VI. Usage of Abstract Mutation Operators

Our set of abstract mutation operators can be used to define specific mutation operators of a modelling language (i.e. metamodel), which can then be used to mutate models (or programs) described by that metamodel. In particular, we have used those abstract operators to generate specific operators for ATL and EOL (which are described by Ecore) by considering every language concept in each. Furthermore, we have investigated whether the set of ATL mutation operators designed in [5], [13], [10], and [14] can be generated using our set of abstract operators. In this paper in particular, we only give the investigation of some mutation operators designed by Khan and Hassine in [13] as an example. Those set of mutation operators are (see fig 2 for ATL modelling concepts):

1) **Matched to Lazy:** this mutation operator can be implemented using *AMO:multi-feature:REP* and applied to the ATL language feature *ModuleElement elements* of the containing object *Module* (c.f. fig 2). This should allow the replacement of a target *MatchedRule* with an instance of *LazyRule*. Elements of the target *MatchedRule* should be copied over (where applicable) to the instance of *LazyRule*.
2) **Delete Attribute Mapping:** this mutation operator can be implemented using *AMO:multi-feature:DEL* and applied to the feature *Binding bindings* of the containing object *OutPatternElement*. This should allow the deletion of one of the instances represented by *bindings*.
3) **Add Attribute Mapping:** this mutation operator can be implemented using *AMO:multi-feature:ADD* and applied to the feature *Binding bindings*. This should allow the addition of one more attribute mapping.
4) **Change Rule's Source Type:** this operator can be implemented using *AMO:single-ref:REP* and applied to

[6] goo.gl/oyKhQH

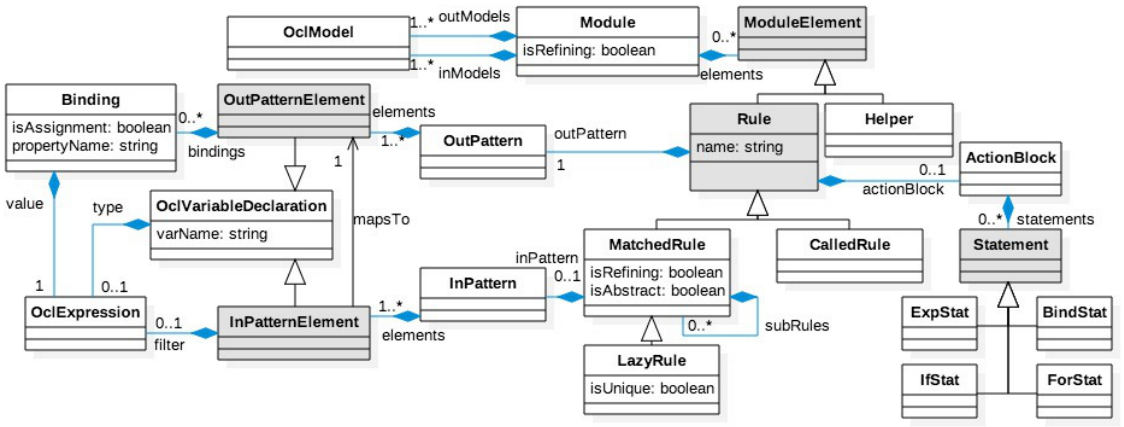

Fig. 2. ATL metamodel

feature *OCLExpression type* of the containing object *OclVariableDeclaration* of the *InPatternElement*. This should allow the replacement of an existing value of type with another one.

5) **Change Execution Mode: from default to refining:** this mutation operator can be implemented using *AMO:single-attr:Boolean* and negate the value of the feature *isRefining* of the containing object *Module*.

VII. Conclusion and Future Work

Our application of the abstract mutation operators over ATL and EOL has raised our confidence towards a comprehensive set of operators for both languages. We believe that these operators are complete since we considered all language concepts. Furthermore, we have compared our set of operators over previously designed ones that have been discussed in the related literature, and have studied whether our operators can cover them. Our initial analysis shows promising results towards the validation of our approach. Hence, we believe that our systematic approach can raise the confidence of covering all features of a language and can facilitate the generation of a set of complete mutation operators that will help the test developer in improving the test suite. One considerable issue here is that a large set of mutation operators can generate a great number of mutants that can grow significantly as the program scales up. Another issue that needs to be considered is the efficiency of the generated operators. Both these crucial issues can be addressed and resolved by studying the effectiveness of a list of concrete mutation operators upon the mutation score; a challenge that we are planning to tackle in our future work.

For tool support, we are planning to develop a tool to automate the generation of all specific mutation operators of a given metamodel based on our presented approach.

References

[1] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge University Press, 2008.

[2] S. Kim, J. A. Clark, and J. A. McDermid, "The rigorous generation of java mutation operators using hazop," in *12th International Conference on SOFTWARE and SYSTEMS ENGINEERING and their APPLICATIONS (ICSSEA'99)*, 1999.

[3] S. Savarimuthu and M. Winikoff, "Mutation operators for the goal agent language," in *Engineering Multi-Agent Systems: First International Workshop, EMAS 2013, St. Paul, MN, USA, May 6-7, 2013, Revised Selected Papers*, M. Cossentino, A. E. F. Seghrouchni, and M. Winikoff, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 255–273.

[4] J.-M. Mottu, B. Baudry, and Y. L. Traon, "Mutation analysis testing for model transformations," in *Model Driven Architecture - Foundations and Applications*, ser. LNCS, A. Rensink and J. Warmer, Eds. Springer Berlin Heidelberg, 2006, vol. 4066, pp. 376–390.

[5] P. Fraternali and M. Tisi, "Mutation analysis for model transformations in atl," in *Model Transformation with ATL, 1st International Workshop, MtATL 2009*, F. Jouault, Ed., July 2009, pp. 145–149.

[6] S. Sen, B. Baudry, and J.-M. Mottu, "Automatic model generation strategies for model transformation testing," in *Theory and Practice of Model Transformations*, ser. LNCS, R. F. Paige, Ed. Springer Berlin Heidelberg, 2009, vol. 5563, pp. 148–164.

[7] J.-M. Mottu, S. Sen, M. Tisi, and J. Cabot, "Static analysis of model transformations for effective test generation," in *Proceedings of the 2012 IEEE 23rd International Symposium on Software Reliability Engineering*. Washington, DC, USA: IEEE Computer Society, 2012, pp. 291–300.

[8] E. Guerra, "Specification-driven test generation for model transformations," in *Theory and Practice of Model Transformations*, ser. LNCS, Z. Hu and J. de Lara, Eds. Springer Berlin Heidelberg, 2012, vol. 7307.

[9] 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," *Software Testing, Verification and Reliability*, vol. 25, no. 5-7, pp. 653–683, 2015.

[10] J. Troya, A. Bergmayr, L. Burgueño, and M. Wimmer, "Towards systematic mutations for and with atl model transformations," in *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on*, 2015, pp. 1–10.

[11] D. Gasevic, D. Djuric, and V. Devedzic, *Model Driven Engineering and Ontology Development*, 2nd ed. Springer, 2009.

[12] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF Eclipse Modeling Framework*, 2nd ed., E. Gamma, L. Nackman, and J. Wiegand, Eds. Addison-Wesley, 2008.

[13] Y. Khan and J. Hassine, "Mutation operators for the atlas transformation language," in *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*, March 2013, pp. 43–52.

[14] P. Gómez-Abajo, E. Guerra, and J. de Lara, "Wodel: A domain-specific language for model mutation," in *31st ACM Symposium on Applied Computing (SAC 2016)*, 2016.