Analysis of Model Transformations via Alloy

Kyriakos Anastasakis[1] and Behzad Bordbar[1] and Jochen M. Küster[2]

[1] School of Computer Science, University of Birmingham, Edgbaston, Birmingham, UK
[K.Anastasakis,B.Bordbar]@cs.bham.ac.uk
[2] IBM Zurich Research Laboratory, Säumerstr. 4, 8803 Rüschlikon, Switzerland
jku@zurich.ibm.com

Abstract. The concept of model transformations is central to the domain of Model Driven Engineering (MDE). A model transformation automates the translation of models between a source and a target language. In order to reason about the correctness of the translation it is important to be able to analyse model transformations. A model transformation specification can be considered as a special kind of model and as such it can be subject to existing model analysis techniques. In this paper we present a systematic method of representing declarative model transformations in a formalism called Alloy. We demonstrate how the Alloy Analyzer can be used to conduct fully automated analysis of a model transformation specification represented in Alloy. The presented approach is explained with the help of an example model transformation in business processes.

1 Introduction

The Model Driven Architecture (MDA) [1] aims at promoting the role of models in the software development process. Typically, a model transformation is specified by a number of *transformation rules*, which define the mapping of constructs of the metamodel of a *source* language into constructs of the metamodel of a *target* language. The metamodels of the source and target languages are specified using a common meta language, the Meta Object Facility (MOF) [2]. A number of languages have been proposed for the definition of the transformation rules(see e. g. [3, 4]) and the Queries/Views/Transformations (QVT) [5] standard specifies the characteristics and capabilities model transformation languages should have.

As model transformations can be considered as programs that translate models expressed in one modeling language into models expressed in another modeling language, their quality becomes crucial for the success of model-driven engineering and needs to be ensured by applying suitable validation and analysis techniques. In this paper, we propose the use of Alloy [6] as the formalism used for the formal analysis of model transformations.

Alloy can be considered as a natural choice for the representation of model transformations. It is a declarative language based on first-order logic [6] and has strong foundations on relational logic. The idea of using relations for specifying model transformations has already received some attention [7] and the QVT standard [5] also contains a high-level relational language. As a consequence, using a relational approach such as Alloy for formal analysis seems to be promising. Another incentive for using the Alloy language is the fact that it comes with a tool, the Alloy Analyzer [8], which can be used to automatically analyse Alloy models.

The remainder of the paper is structured as follows: First, we briefly introduce key concepts of Alloy in Section 2. We then describe our approach of analysing model transformations with Alloy and apply it to an example in Section 3. We discuss limitations of our approach in Section 4 and end with a discussion of related work and conclusions.

2 Alloy

Alloy [6] is a textual, declarative modelling language based on first-order relational logic. An Alloy model consists of *Signatures*, *Relations*, *Facts* and *Predicates*. *Signatures* represent the entities of a system and *Relations* depict the relations between such entities. *Facts* and *Predicates* specify constraints, which apply on the *Signatures* and *Relations*.

Alloy comes with a tool, the Alloy Analyzer [8], which supports fully automated analysis of Alloy models. The analyser provides two main functionalities, *Simulation* and *Assertion* checking. *Simulation* produces a random instance of the model, which conforms to the specification. This ensures that the developed model is not inconsistent. *Assertions* are constraints, which the model needs to satisfy.

The Alloy Analyzer works by translating Alloy formulas to boolean expressions, which are analysed by SAT solvers embedded within the analyser. A user-specified *scope* on the model elements bounds the domain. If an instance that violates an assertion is found within the scope, the assertion is not valid. However, if no instance is found, the assertion might be invalid in a larger scope. For more details on the notion of scope, please refer to [6, Sect. 5].

3 Description of the Approach

Figure 1 depicts an outline of our approach, which is comprised of two steps. The first step is to convert the MDA compliant model transformation specification to an equivalent specification expressed in the Alloy language. The second step is to use the Alloy Analyzer to analyse the produced Alloy model. These two steps are explained in more detail in the following.

Step 1:Translate the Model Transformation Specification to Alloy. A model transformation specification consist of a MOF compliant representation of the source metamodel, a MOF representation of the target metamodel and the transformation rules, which define the mappings between the metamodels.

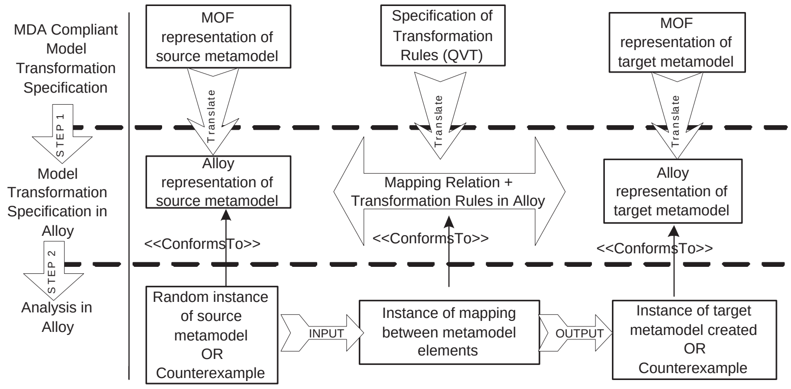
MDA Compliant Model Transformation Specification
STEP 1
Model Transformation Specification in Alloy
STEP 2
Analysis in Alloy
MOF representation of source metamodel
Specification of Transformation Rules (QVT)
MOF representation of target metamodel
Translate
Translate
Translate
Alloy representation of source metamodel
Mapping Relation + Transformation Rules in Alloy
Alloy representation of target metamodel
<<ConformsTo>>
<<ConformsTo>>
<<ConformsTo>>
Random instance of source metamodel OR Counterexample
INPUT
Instance of mapping between metamodel elements
OUTPUT
Instance of target metamodel created OR Counterexample

Fig. 1. An outline of our approach

MOF metamodels are usually accompanied by constraints, which define syntactic and semantic properties of the language. For example the UML standard specifies that an aggregation can only appear in binary associations [9, p.110]. Such invariants are often referred to as *well-formedness rules* [9]. Well-formedness rules are usually specified using the Object Constraint Language (OCL) [10] and are considered to be part of the metamodel specification.

The first step of our approach requires that the metamodels of the source and target language as well as the well-formedness rules of the source language are translated to Alloy. This procedure can be automated and a methodology has been developed [11] that translates MOF metamodels enriched with well-formedness rules expressed in OCL, to Alloy. We have implemented this method in a tool called UML2Alloy [11].

Additionally the transformation rules need to be converted to the Alloy language. Transformation rules express under which circumstances, elements of the source metamodel are mapped to elements of the target metamodel. The transformation rules in Alloy are expressed in first-order logic. In order to keep track which elements of the source metamodel are mapped to which elements of the target metamodel, we also introduce a *mapping relation* in Alloy. The notion of the mapping relation is similar to the notion of *trace* classes in the QVT specification [5].

Step 2: Analysis using the Alloy Analyzer. The procedure defined in the previous step results in the production of an Alloy model of the model transformation. The Alloy Analyzer can then be used to analyse the Alloy model to detect flaws in the specification of the model transformation.

The analyser can be used to simulate the transformation. This results in the production of a random instance of the source metamodel that conforms to the well-formedness rules, an instance of the mapping that transforms elements of the source model and the target model generated by the transformation. If the analyser can not produce an instance of the transformation, there is an inconsistency (i.e. conflicting statements) in the definition of the transformation rules. It is relatively straight forward to resolve inconsistencies in Alloy models using Alloy Analyzer. The tool provides an *UnSat Core* [12] functionality that highlights the statements which lead to logical inconsistencies. This functionality can be used to debug inconsistent model transformation specifications.

The Alloy Analyzer can also enumerate all possible instances that conform to the specification of the transformation. As a result it is feasible to explore the potential combinations of target models that can be generated by the given transformation rules. This is useful to identify whether there are more than one possible mappings between the elements of the source and target metamodels.

The Alloy Analyzer can also be used to check whether assertions, certain statements that should hold according to the specification, are satisfied. Assertions can be formulated to check if the target model conforms to the well-formedness rules of the target language. Assertions can also be used to check whether a model generated by the transformation rules satisfies certain properties. If a property is not satisfied, the analyser presents a counterexample, which is an instance of the target model that violates the property. The counterexample can be inspected to deduce the flaw in the definition of the transformation. Section 3.3 illustrates such a case in more detail. The next section briefly presents an example to demonstrate our approach.

3.1 Running Example

We have applied our method to the transformation presented in [13] by Küster et al. This transformation deals with a domain specific language used for business process modelling, utilised by IBM's WebSphere Business Modeler [14]. The language is similar to UML Activity Diagrams [15].

The transformation [13] removes control actions (i.e. *Decision*, *Fork*, *Join* and *Merge* nodes) and replaces them with implicit control actions expressed with the help of *pinsets*. Figure 2 depicts two of the conceptual rules of this transformation, called the Control Action to PinSet (*CA2PinSet*) transformation. For example rule *r2* removes the join control node, adds a new pin in the pinset of *B* and connects all edges incoming to the join, directly to the pins of *B*. For an extended study of the details of this transformation please refer to [13].

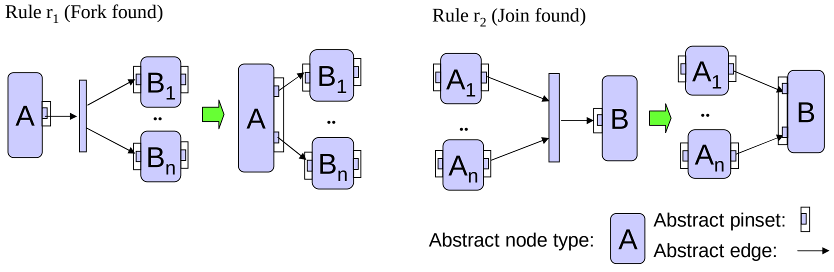

Fig. 2. Transformation rules of the CA2PinSet transformation [13]

Even though this transformation seems simple it can lead to the production of target models, which are not well-formed, as Küster et al. have discovered [13] and as our approach reveals. In the following section we demonstrate how our approach can be applied to the rules depicted in Fig. 2.

3.2 Applying our Approach to the Example

As described in Sect. 3, the first step of our method is to translate the MOF compliant metamodel of the source and target languages to Alloy. A simplified version of the metamodel of the source language, adapted from [13], is depicted in Fig. 3.

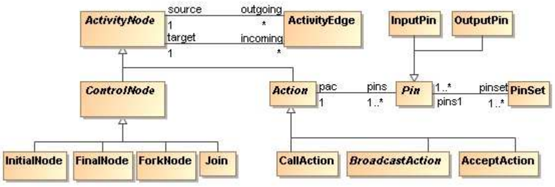

Fig. 3. A simplified metamodel, adapted from [13]

The language defines a number of *ActivityNodes*. Each ActivityNode can be related to a number of *incoming* or *outgoing ActivityEdges*. An ActivityNode can either be a *ControlNode* or an *Action*. A *ControlNode* can be in turn an *InitialNode*, a *FinalNode*, a *ForkNode* or a *Join*. An *Action* can be either a *CallAction*, or a *BroadcastAction*, or an *AcceptAction*. Finally an Action is related to a number of *Pins* and each Pin belongs to one or more *PinSets*.

The source metamodel has embedded well-formedness rules. For example the following rule in OCL expresses that an *IntialNode* has no *incoming* edges.

```
InitialNode.allInstances() -> forAll(i:InitialNode |
i.incoming -> size() =0 )
```

The translation of the metamodel and the well-formedness rules to Alloy is a straight forward procedure [11]. Classes are translated to Alloy *signatures*, while association ends are transformed to Alloy *fields*. Additional multiplicity facts are introduced in the Alloy model to reflect the multiplicity facts of the association ends in the original MOF metamodel. Using UML2Alloy [11], which implements these rules, we constructed an Alloy representation of the metamodel of the source language. Figure 4 depicts an excerpt of the metamodel, with inline comments. Next we need to transform the metamodel of the target language to Alloy.

In the transformation, the target metamodel is the same as the source metamodel, but without any ControlNodes, since all control nodes are represented using PinSets. As a result the transformation of the target metamodel to Alloy is similar to the transformation of the source metamodel to Alloy.

```
abstract sig ActivityEdge{ // Represents the ActivityEdge Class
target: one ActivityNode,  // Represents the target association end
source: oneActivityNode } // Represents the source association end

//Multiplicity Facts
fact{
//The outgoing relation maps one ActivityNode to 0 or more ActivityEdges
outgoing in ActivityNode one -> set ActivityEdge
}
// Well-formedness Rules
fact{
// All Initial nodes have 0 incoming edges.
all i:InitialNode | #i.incoming = 0
}
```

Fig. 4. Portion of the transformed metamodel of the source language in Alloy

Our approach also requires that the transformation rules are expressed in Alloy. Figure 5 depicts an excerpt of the Alloy representation of the transformation rule *r2*, which removes a Join node and replaces it with an implicit join [13]. The elements of Fig. 5, which start with a capital '*T*' represent elements of the target metamodel. For example, *InitialNode* represents the InitialNode of the source metamodel, while *TInitialNode* represents the *InitialNode* of the target metamodel. This convention was necessary to avoid name conflicts in the model (i.e. to distinguish which elements belong to the source metamodel and which elements belong to the target metamodel).

Lines 1-4 in Fig. 5 represent the Mapping relations. Line 2 specifies that an InitialNode of the source metamodel will be mapped to exactly one InitialNode in the target metamodel. Line 3 states that an ActivityEdge will be mapped to at most one ActivityEdge in the target metamodel (as shown by *r2* in Fig. 2 the outgoing ActivityEdge of a JoinNode does not map to any ActivityEdge after the transformation). Lines 5-14 depict an excerpt of the rule defined in Alloy. The comments provide an informal description of the rule in natural language. The full Alloy model of the transformation with a detailed description can be found in [16].

3.3 Analysis

To perform the analysis, the Alloy Analyzer was first used to simulate the transformation presented in Fig. 2. The Analyzer requires the user to specify a scope [6, Sect. 5] and then carries out the analysis by exhaustively searching the state space for the given scope. In our analysis we set a scope of 2 Actions,

```
sig Mapping{
in2tin: InitialNode one -> one TInitialNode,
ae2tae: ActivityEdge one -> lone TActivityEdge
}

//For all ActivityEdges of the source model, if the ActivityEdge
//relates an OutputPin with a JoinNode and the outgoing edge of the
//JoinNode arrives in the InputPin of another Node there is *one*
//ActivityEdge in the target model, where: (a) the ActivityEdge
//of the source model is mapped to the ActivityEdge of the
//target model through the Mapping relation and (b) the target
//ActivityEdge arrives at the InputPin of the CallAction, where the
//outgoing edge of the JoinNode arrived in the source model.
pred ruleR2(){ all m:Mapping | all ae:ActivityEdge |
ae.source in OutputPin && ae.target in JoinNode => //OutputPin2Join
one tae:TActivityEdge | (ae.target.outgoing.target in InputPin =>
   tae = m.ae2tae[ae] &&
       tae.ttarget = m.ip2tip[ae.target.outgoing.target] &&
        (
          all ca:ae.target.pac |
                   ca in CallAction => ( one tc:TCallAction |
          tc = m.ca2tca[ca] && tc = tae.ttarget .tpac
        )))}
```

Fig. 5. An excerpt of the mapping relation and the transformation rules

2 CallActions, 4 Pins, 1 JoinNodes, 10 ActivityNodes and a default scope of 10. This suggests that the analyser probes to find an instance that conforms to the transformation rules for any combination of 2 Actions, 2 CallActions, 4 Pins, 1 JoinNodes, 10 ActivityNodes.

The Alloy Analyzer produced an instance of a transformation where a well formed target model was generated. However this does not mean that a well-formed target model will be generated for every possible input model.

Assertions can be used to verify that a target model will always be well-formed given the transformation rules. For example the well-formedness rules of the language require that a FinalNode has only one incoming edge [13]. This can be formulated with the following Alloy statement:

```
// All FinalNodes of the target metamodel have exactly one
// incoming edge
 all tf:TargetFinalNode | #tf.incoming = 1
```

We checked this assertion using the same scope. This assertion produced a counterexample, which is represented in Fig. 6. The left hand side depicts an instance of the source model and the right hand side an instance of the target model that violates the well-formedness rules. It can be easily observed that when the JoinNode is removed, ActivityEdge 1 (*AE1*) and ActivityEdge 3 (*AE3*) are both connected to the FinalNode and thus produce a syntactically incorrect target model. In order to resolve this problem, the transformation rules need to be augmented so that such a target model is not created.

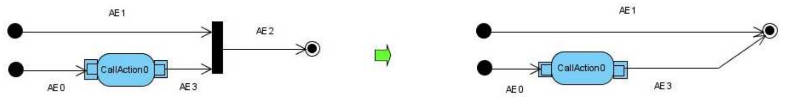

Fig. 6. Instance which violates the well-formedness rules of the target

4 Discussion

The Alloy Analyzer conducts bounded analysis by using the scope to restrict the state space. It is therefore expected that our approach will not scale well when large metamodels and complex transformation rules are involved. However in such cases, the method presented here might still be applicable. It is expected that the properties, which need to be checked, are related to certain elements of the metamodels and are affected by certain transformation rules. Therefore it might be possible to abstract complex transformation rules by removing the parts which are not related to the properties of interest.

A number of languages used for the definition of model transformations are imperative [3] and hybrid [4] (i.e. provide support for both imperative and declarative specifications). Alloy is a declarative language. As a result it is not possible to directly translate imperative model transformations to Alloy and analyse them. It might be possible to abstract an imperative transformation to a declarative one by removing the computational details of the transformation, which are not of interest.

The Alloy language has a simple type system. The only primitive types supported are Integers. As a consequence it is not possible to use Alloy to analyse properties involving certain types (i.e. String, Real numbers).

Models in Alloy are static, i.e. they capture the entities of a system, their relationships and constrains about the system. An Alloy model defines an instance of a system where the constraints are satisfied. More specifically, Alloy does not have any built in notion of statemachine [6, Ap. B.5.1]. As a result our approach can only be used to reason about static properties of the transformation. For example it is not possible to reason whether applying a rule r1 before a rule r2 in a model, will have the same effect as applying r2 before r1. It is however possible to model dynamic systems in Alloy [6]. Extending our approach to reason about dynamic properties of transformations remains for further research.

On the example transformation presented in this paper, we used Alloy to check whether the transformation produces well-formed target models. Applying our method to more case studies will provide us with more details on the range of properties that can be analysed.

5 Related Work

One of the most popular methods of checking model transformations is by using model based testing techniques. In particular Fleurey et al. [17] deduce metamodel coverage criteria and automatically produce test models. Küster et al. [13] suggest the use of a white box approach and propose three techniques for developing test cases. Baudry et al. [18] present various model transformation testing approaches and demonstrate the challenges involved. Unlike those methods, our approach does not require the construction of test cases. Instead the Alloy Analyzer exhaustively searches the state space for the given scope, in order to validate certain properties.

Massoni et al. [19] use Alloy to validate certain properties on UML class diagram refactorings [20]. Unlike their approach, our method is based on metamodelling and as a result it is not restricted to model refactorings.

Further approaches for verifying model transformations come from the domain of graph transformations: Baresi et al. [21] present a case study where they use graph transformations theories to verify that their transformation is semantic preserving. Graph transformation can also be used for verifying that a model transformation is confluent and terminates (see e.g. [22, 23]). It remains for future work to perform a detailed comparison of different approaches for the formal analysis of model transformation properties.

6 Conclusion

Model transformations, like any other piece of software, can be inconsistent and produce undesirable results under certain circumstances. Therefore the ability to analyse model transformations is of paramount importance. In this paper we demonstrated an approach of representing model transformations in Alloy and illustrated some of the capabilities of the method with the help of an example.

References

1. Kleppe, A., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture–Practice and Promise. The Addison-Wesley Object Technology Series. Addison-Wesley (2003)
2. OMG: MOF Core v. 2.0 Document Id: formal/06-01-01. http://www.omg.org.
3. Akehurst, D.H., Bordbar, B., Evans, M.J., Howells, W.G.J., McDonald-Maier, K.D.: SiTra: Simple transformations in java. In Nierstrasz, O., Whittle, J., Harel, D., Reggio, G., eds.: Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006. Volume 4199 of Lecture Notes in Computer Science., Genova, Italy, Springer (2006) 351–364
4. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Satellite Events at the MoDELS 2005 Conference. Volume 3844 of LNCS., Springer (2006) 128–138
5. http://www.omg.org: MOF QVT final adopted specification Document Id:ptc/2005-11-01. http://www.omg.org.

6. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press, London, England (2006)
7. Akehurst, D.H., Kent, S., Patrascoiu, O.: A relational approach to defining and implementing transformations between metamodels. Software and System Modeling **2**(4) (2003) 215–239
8. Jackson, D.: Alloy Analyzer website http://alloy.mit.edu/.
9. OMG: UML Infrastructure Document: formal/05-07-05. http://www.omg.org.
10. OMG: OCL Version 2.0 Document id: formal/06-05-01. http://www.omg.org.
11. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: UML2Alloy: A Challenging Model Transformation. In: ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2007). (2007) To Appear.
12. Shlyakhter, I., Seater, R., Jackson, D., Sridharan, M., Taghdiri, M.: Debugging overconstrained declarative models using unsatisfiable cores. In: Proceedings of the 18th IEEE International Conference on Automated Software Engineering, Montreal, Canada, IEEE Computer Society (2003) 94–105
13. Küster, J.M., Abd-El-Razik, M.: Validation of model transformations - first experiences using a white box approach. In Kühne, T., ed.: MoDELS Workshops. Volume 4364 of Lecture Notes in Computer Science., Springer (2006) 193–204
14. IBM: Websphere Business Modeller http://www-306.ibm.com/software/integration/wbimodeler/.
15. OMG: UML: Superstructure. Version 2.0 Document id: formal/05-07-04. http://www.omg.org.
16. Anastasakis, K., Bordbar, B.: Using Alloy for the Analysis of Model Transformations: A Case Study. Technical report, School of Computer Science, The University of Birmingham, UK (2007) (In Preperation). http://www.cs.bham.ac.uk/∼kxa/files/analysis/techrep07.pdf.
17. Fleurey, F., Steel, J., Baudry, B.: Validation in model-driven engineering: testing model transformations. In: First International Workshop on Model, Design and Validation. (2004) 29– 40
18. Baudry, B., Dinh-Trong, T., Mottu, J.M., Simmonds, D., France, R., Ghosh, S., Fleurey, F., Le Traon, Y.: Model transformation testing challenges. In: ECMDA workshop on Integration of Model Driven Development and Model Driven Testing. (2006)
19. Massoni, T., Gheyi, R., Borba, P.: Formal Refactoring for UML Class Diagrams. In: 19th Brazilian Symposium on Software Engineering (SBES), Uberlandia, Brazil (2005) 152–167
20. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: improving the design of existing code. Addison-Wesley, Boston, MA, USA (1999)
21. Baresi, L., Ehrig, K., Heckel, R.: Verification of model transformations: A case study with BPEL. In: Second Symposium on Trustworthy Global Computing, TGC'06. (2006)
22. Ehrig, H., Ehrig, K., Lara, J., Taentzer, G., Varró, D., Varró-Gyapay, S.: Termination Criteria for Model Transformation. In: FASE. (2005) 49–63
23. Küster, J.M.: Definition and validation of model transformations. Software and Systems Modeling **5**(3) (2006) 233–259